Unisys e-@ction ClearPath Enterprise Servers

Task Management
Programming Guide

ClearPath MCP Release 7.0 SSP1

March 2002

Printed in USA
8600 0494–506

UNISYS e-@ction CLEARPATH ENTERPRISE SERVERS

Task Management
Programming Guide

ClearPath MCP Release 7.0 SSP1

Printed in USA

March 2002 8600 0494–506

Unisys e-@ction
ClearPath Enterprise
Servers

Task Management

Programming Guide

**ClearPath MCP
Release 7.0 SSP1**

Unisys e-@ction
ClearPath
Enterprise
Servers

Task Management

**Programming
Guide**

**ClearPath MCP
Release 7.0
SSP1**

8600 0494–506

8600 0494–506

Bend here, peel upwards and apply to spine.

Contents

Figures

Tables

Section 1
Understanding Basic Tasking Concepts

Tasking features are inherent in the overall system architecture. Various programming languages and operations interfaces provide you with access to different subsets of the tasking capabilities of the system. This section presents an overview of tasking features and discusses the advantages and limitations of these features.

About This Guide

Purpose

This guide describes the following types of operating system features that are accessed using programming languages:

- Tasking features

  Features that enable processes to initiate, monitor, and control other processes include the CALL, PROCESS, and RUN statements, task variables, and task attributes. Features related to job restarting and process history also are this category.

- Interprocess communication features

  Features that enable user-defined information to be passed between processes, or that help regulate the timing of parallel processes include events, libraries, and parameter passing.

Audience

The audience for this guide consists of applications programmers familiar with at least one high-level programming language, such as ALGOL, C, COBOL74, COBOL85, FORTRAN77, Pascal, or WFL.

Terminology Conventions

Statements about ALGOL in this guide apply also to DCALGOL, DMALGOL, and BDMSALGOL unless otherwise specified.

Two different ANSI levels of COBOL are supported: ANSI-74 and ANSI-85. These implementations are referred to in this guide as COBOL74 and COBOL85, respectively.

Statements in this guide about *COBOL* are true of both COBOL implementations unless otherwise specified.

The term *library,* which was used in previous editions of this guide, has been replaced by the term *server library.* The term *user process* (when used in the context of libraries) has been replaced by the term *client process.* The library as it is declared in the client process is now referred to as the *client library.*

These changes resulted from the implementation of a new type of libraries, called *connection libraries.* The term *library* is now used as a general term referring to a server library, a client library, or a connection library.

Tasking Concepts

The following subsections discuss the relationships between programs and processes, and the methods you can use to monitor and control process behavior.

Programs and Processes

A *program* is a sequence of statements written in any of a number of languages, including ALGOL, C, COBOL74, COBOL85, FORTRAN77, Pascal, and Work Flow Language (WFL). The file in which you write and store these statements is referred to as a *source file*. By compiling the source file, you cause the creation of an *object code file*.

By using any of a number of commands or statements, you can cause a particular object code file to be *initiated*. That is to say, you cause the system to start performing the instructions in the object code file. At this point, the object code file is being *executed*. However, in a sense, nothing is happening to the object code file itself. The system merely reads instructions from the object code file; the contents of the file remain unchanged.

There is, nonetheless, a dynamic entity called a *process*, which is separate from the object code file, but which reflects the current state of the execution of the object code file. A process stores the current values of variables used by the program, as well as information about which procedures have been entered and which statement is currently being executed. (Procedures are discussed under "Internal and External Processes" later in this section.)

Each process exists in the system memory, and consists of several distinct structures that are discussed in Section 8, "Controlling Process Memory Usage."

The distinction between object code files and processes is a very important one. This is because, at any given time, there can be multiple processes that are executing the same object code file; these are referred to as *instances* of that object code file. A new instance is created each time a user or an existing process submits a statement that initiates the object code file.

Because many instances of the same object code file can be running at the same time, the object code file title is not sufficient to uniquely identify a process. Therefore, in system command displays, the various processes are identified both by an object code file title and by a unique number called the *mix number.* For further information on mix numbers, refer to Section 5, “Establishing Process Identity and Privileges.”

Even if processes are executions of the same object code file, the processes are completely separate entities and do not interact with each other. For example, suppose the object code file called OBJECT/PROG includes a declaration of an integer variable named N, as well as various statements that assign values to N. In this case, each instance of OBJECT/PROG has its own copy of variable N in memory. When one process changes the value of N, there is no change to the value N has for the other processes.

The fact that processes are separate and maintain their own copies of variables generally prevents confusion and simplifies program design. However, there can also be cases where you want processes to have shared access to a particular variable. For these cases, the system provides a variety of interprocess communication techniques, which are described in Part II of this guide.

Tasking consists of using various features to initiate, monitor, and control processes. You can perform tasking functions by entering commands through various system operation interfaces, or by writing programs that initiate, monitor, and control the execution of other programs.

Task Attributes

Task attributes are entities that record various properties of a process, such as its usercode, mix number, priority, printing defaults, and so on.

There are a limited number of task attributes, which are defined by the operating system and have fixed meanings. Each process possesses all of these task attributes, but the values of the task attributes can vary. For example, each process has a USERCODE task attribute, but where one process might have a USERCODE value of JASMITH, another process might have a USERCODE value of JANEDOE.

Task attributes record or modify many aspects of process execution, including security, processor usage, memory usage, and I/O activity. You can assign task attributes to a process either through commands entered at an interactive source, or through statements in a program.

This guide introduces many of the important uses of task attributes. The remaining sections in Part I of this guide introduce task attributes within discussions of general functional areas, such as processor usage, memory usage, and so on. For detailed information about any of these task attributes, you can refer to the *Task Attributes Programming Reference Manual*, which presents the task attributes in alphabetical order.

Interactive Tasking

You can perform tasking functions through any of the following interactive interfaces:

- Command and Edit (CANDE)

  A command-driven environment that provides file handling and tasking capabilities
- Menu-Assisted Resource Control (MARC)

  A menu-driven interface to system operations functions
- Operations Center

  A Windows-based application that provides a graphical user interface (GUI) to system operations
- Operator display terminals (ODTs)

  Terminals that support an interface called *system command mode*

Each of these products provides the following general types of tasking capabilities:

- A command or menu selection that allows you to initiate any object code file by name. Examples are the RUN command in CANDE and MARC.
- Syntax for specifying *task equations*, which are task attribute assignments applied to a process when it is first initiated.
- Task attribute *inheritance*, which causes a process to receive task attributes associated with the initiating source.
- Various commands or selections for monitoring process status and resource use, or for intervening in process execution.

The tasking capabilities of CANDE, MARC, Operations Center, and the ODT are described in Section 3, "Tasking from Interactive Sources."

Note that many commands entered by users can indirectly cause a process to be initiated. For example, the Transaction Server initiates instances of direct window programs in response to variations in the message traffic from users. Similarly, the system initiates processes to execute some specialized system commands, such as LOG.

This guide does not attempt to describe all such cases of indirect tasking. CANDE, MARC, and the ODT are all introduced in this guide because they provide direct, generalized tasking interfaces. With these products, you can initiate any object code file, as well as monitor and control any process (to the extent allowed by system security).

Programmatic Tasking

You can perform tasking functions using any of the following programming languages: ALGOL, C, COBOL74, COBOL85, and WFL. This guide provides details about the tasking capabilities of ALGOL, WFL, and both versions of COBOL.

Each of these languages provides you with the following types of tasking capabilities:

- Statements that allow you to initiate any object code file by name. Examples are the CALL, PROCESS, and RUN statements in ALGOL and COBOL.
- Constructs for reading and assigning the task attributes of a process before the process is initiated, while it is running, and after it completes execution.

The tasking capabilities of each of these languages are described in Section 4, "Tasking from Programming Languages."

At this point you might be aware of the potential for some ambiguity in the use of task attributes within programs. For example, every process has a USERCODE task attribute. If you write a program that makes an assignment to the USERCODE task attribute, how does the system know which process the USERCODE should be applied to?

The answer is that ALGOL, COBOL, and WFL all provide a special type of variable called a *task variable*. A task variable is also known as a *control point* in COBOL. You can declare one or more task variables in a program, each with a distinct name. When you use a process initiation statement, you include a reference to a task variable in that statement. The task variable thereafter becomes associated with the new process.

Statements that use task attributes always specify a task variable name as well as a task attribute name. In this way, it is always clear which process is being referred to.

When one process initiates another process, many of the task attributes of the initiating process are transferred to the new process. This transference is called *inheritance*. Details about the task attributes that are inherited, and under what circumstances they are inherited, are given in the *Task Attributes Programming Reference Manual*.

Process Termination

A process typically ends when the last instruction in the object code file is executed. This is referred to as a *normal termination*.

However, a process can also terminate prematurely for any of a number of reasons. For example, you can use the DS (Discontinue) system command to terminate a process. A process can also terminate because a flaw in program design causes it to attempt to do something impossible, such as dividing by zero. Additionally, all processes are terminated in the event of a system halt/load. All of these types of terminations are referred to as *abnormal terminations* because the inference is that something went wrong.

When you initiate a process, you usually want to be able to find out later whether it ran successfully or not. The system provides a number of facilities to help you determine whether the process ran successfully, and why it failed if it was not successful. These facilities include the HISTORYTYPE, HISTORYCAUSE, and HISTORYREASON task attributes, and the program dump facility. These facilities are described in Section 10, "Determining Process History."

Sometimes you might want to rerun a process that terminated abnormally. For example, if the process was terminated by a system halt/load, then the underlying program might be perfectly sound. Restarting the process could enable it to complete its work successfully. However, a number of design issues must be considered for processes that are intended to be restartable. These design issues, and the means of restarting processes, are explained in Section 11, "Restarting Jobs and Tasks."

Internal and External Processes

Up to this point, this section has discussed only cases where an object code file is executed from beginning to end as a single process. However, the system gives you the option of causing individual procedures to be initiated as separate processes. These processes fall into two general categories: internal and external processes.

The following subsections describe the various types of internal and external processes. For a discussion of the varying capabilities of these types of processes, refer to the discussion of inclusion in Section 2, "Understanding Interprocess Relationships."

Internal Processes

Many programming languages give you the ability to create groups of declarations and statements within a program, and to assign a name to each group. In ALGOL, these groupings are referred to as *procedures*. In WFL, these groupings are referred to as *subroutines*. However, the basic concept is similar in both cases, and the term "procedure" in this guide refers equally to ALGOL procedures and WFL subroutines.

Other programming languages offer similar types of structures, but ALGOL and WFL are the only languages that give you a choice between the following two methods of invoking a procedure:

- Procedure entrance

  The syntax for entering a procedure consists of using the procedure name as if it were a statement. Entering a procedure causes the procedure to be executed as part of the same process that invoked the procedure. When the process finishes executing the procedure, the process *exits* that procedure.

- Procedure initiation

  The syntax for initiating a procedure consists of using a CALL, PROCESS, or RUN statement in ALGOL, or a PROCESS <subroutine> statement in WFL. Initiating a procedure causes it to be executed as a new process, separate from the process that invoked the procedure. This new process is referred to as an *internal process* because it is executing part of the same object code file as the initiating process.

Of these methods, procedure entrance has the advantages of simplicity and low impact on system resources, as discussed under "Limitations of Tasking" later in this section.

On the other hand, procedure initiation allows you to use parallel processing or to assign the new process different task attribute values than those of the initiating process. These features are introduced under "Advantages of Tasking" later in this section.

Note that, if you use the Binder utility to bind a procedure from a subprogram into a host program, that procedure is thereafter considered an internal procedure of the host program. If the host program is an ALGOL program, the host program can either enter or initiate the bound procedure. If the procedure is initiated, the resulting process is considered to be an internal process. For information about the Binder utility, refer to the *Binder Programming Reference Manual*.

External Processes

An external process is one that results when a statement in a program initiates an external procedure. An external procedure is one that resides in a program other than the program containing the statement that invokes the procedure. External procedures are of three types:

- Separate programs

  Any program, taken as a whole, can be thought of as an external procedure when it is invoked by a statement in a different program. A separate program is always executed as a separate process; that is, a process can initiate, but cannot enter, a separate program. WFL, ALGOL, and COBOL all allow you to initiate separate programs. In ALGOL and COBOL, you must specify dummy procedures, called *declared external procedures*, in statements that initiate separate programs.

- Passed external procedures

  These are procedures passed into the program as parameters. You can write programs in ALGOL that accept procedures as parameters from the initiating program. Statements in the receiving ALGOL program can either enter or initiate a passed procedure.

- Library procedures

  These are procedures that are provided by a special type of program called a *library*. Libraries make procedures available for use by other programs. Statements in an ALGOL program can either enter or initiate a library procedure. Programs written in other languages can enter, but cannot initiate, a library procedure. The methods for writing libraries and programs that use libraries are discussed in Section 18, "Using Libraries."

Program Structure

Each program is viewed by the operating system as having a certain *block structure*. The block structure of the program can have implications for the critical block definition and for the ability of processes to communicate through global objects. For further information on these topics, refer to “Critical Blocks” in Section 2, “Understanding Interprocess Relationships” and to Section 15, “Using Global Objects.”

The term *flow of control* refers to the order in which the statements of a program are executed. Most statements perform an action and then pass control to the immediately following statement. However, some statements can pass control to structures residing elsewhere in the program.

A *block* is a program, or program subunit, that can contain a group of declarations and a group of statements. The declarations create objects that are for local use by the statements in the block. There are two kinds of blocks: procedures and simple blocks.

A *procedure* is a block that can be executed using a procedure invocation statement, which passes control to the start of the procedure. When the procedure finishes executing, control automatically returns to the procedure invocation statement, and passes to the next statement in the program.

This abstract definition of a procedure corresponds to the way procedures are viewed by the operating system. Procedures are called by different names in the syntax of the various programming languages. This definition of a procedure corresponds, for example, to a PROCEDURE in ALGOL, a PROCEDURE or FUNCTION in Pascal, or a SUBROUTINE or FUNCTION in FORTRAN77. It also corresponds to a complete program written in any of these languages.

Note that a complete program written in COBOL is also considered a procedure. However, a paragraph or a section in COBOL is not considered a procedure. It is true that a PERFORM statement resembles a procedure invocation statement in that it causes control to pass through the paragraph or section and then return to the PERFORM statement. However, paragraphs and sections cannot include declarations and thus are not treated as procedures by the operating system. Therefore, the various properties of procedures discussed in this guide do not apply to paragraphs or sections.

Similarly, COBOL85 nested programs are not currently treated as procedures by the operating system. However, this implementation is subject to change. In future versions of COBOL85, nested programs might be treated like blocks.

A *simple block* is a block that cannot be specified in a procedure invocation statement. Simple blocks exist only in ALGOL, where they appear among the statements in the program, rather than among the declarations. The beginning and end of a simple block are marked by the keywords BEGIN and END. A simple block is executed in sequence between the statements that immediately precede and follow the simple block.

Note that a BEGIN...END group is considered to be a simple block only if it contains at least one declaration. Otherwise, it is considered a compound statement. Compound statements do not affect tasking or interprocess communication issues, and will not be further discussed in this guide.

Some languages, including WFL and ALGOL, allow blocks to be declared within other blocks. This practice is referred to as *nesting*. A block that contains a nested block is said to be *global* to that nested block. The most global block is referred to as the *outer block* of the program.

The *lexical level* of a block is a measure of how deeply the block is nested. By default, the outer block of a program has a lexical level of 2; however, compiler control options can be used to cause the outer block to be compiled with a higher lexical level. Each procedure has a lexical level one higher than the outer block or procedure in which it is declared.

ALGOL and NEWP support special TYPE declarations called *structure blocks* and *connection blocks.* Either of these declarations creates a data type consisting of a group of objects of possibly varying types. Note that structure blocks and connection blocks are not considered blocks in the sense that the term *block* is used in this book.

Advantages of Tasking

The benefits of tasking fall into the general areas of simplifying system operations, increasing programmer productivity, and improving performance of an application.

Simplifying System Operations

Many applications involve running a sequence of programs, one after another in a certain set order. Often it is necessary to specify parameters and task attribute assignments for each of the programs. An operator can initiate the programs individually, providing the needed parameters and task attribute assignments in each case. However, this proves to be too time consuming in an environment where many applications are run during a given work shift.

An alternative, which reduces the labor required of the operator, is to write a small program whose only purpose is to initiate a series of other programs. Such a program can provide a standard set of parameters and task attribute assignments. You can write such a program in ALGOL, COBOL, or WFL. This enables the operator to initiate a single program and leave it to initiate all the others.

WFL is particularly suitable for implementing such programs because WFL programs typically pass through *job queues*. An operator can use the MQ (Make or Modify Queue) system command to create job queues and assign various job queue attributes to them. The use of job queues enables the operator to submit jobs when it is convenient, while relying on the system to initiate jobs at specified times or according to specified criteria. Job queues are further discussed under “Selecting the Queue for a Job” in Section 4, “Tasking from Programming Languages.”

Increasing Programmer Productivity

Tasking techniques can improve programmer productivity by modifying the behavior of existing programs, by allowing you to use programs as modules in a larger application, and by allowing multiple programming languages to be used in an application.

Modifying Program Behavior

Sometimes a program is designed to run in a particular environment, and later that environment changes. For example, a program might be designed to read a file on a family named DATAPK. Later, you might want to run a copy of that program on a different system that does not have a family with that name. Rewriting the source program and recompiling it can be a time-consuming process. Fortunately, many such behaviors can be modified through task attribute assignments.

For example, there is a task attribute called FAMILY that causes a process to use files on a different family than it otherwise would. Suppose a process expects to find all its input files on the family named DATAPK. You can assign the FAMILY task attribute a value of "DATAPK = CONTROL OTHERWISE DISK". This causes the process to look for all its input files on the family named CONTROL instead of the family named DATAPK.

You can assign a task attribute to a process in any of the following ways, none of which requires recompiling or rewriting the program that is being initiated:

- If you are running a program from CANDE or MARC, you can append task attribute assignments to the RUN command that initiates the program.
- You can use a WFL *MODIFY* statement to assign default task attribute values to an object code file. The system assigns these task attribute values each time the object code file is run.
- ALGOL, COBOL, and WFL all allow you to assign task attributes to a task variable. If you then specify this task variable in a statement that initiates a separate program, the task attribute assignments are applied to the new process.

The *Task Attributes Programming Reference Manual* gives examples of these methods of assigning task attributes.

Using Programs as Modules

A *module* is a body of code that can be reused in a variety of different contexts. The use of modules simplifies the programmer's job by making it unnecessary to repeat large amounts of code. One advantage of tasking is that it allows you to use an entire program as a module in one or more larger applications.

For example, you could have a report-formatting and printing program. You might also have a program that retrieves customer data from a database, and another program that does an inventory analysis. The customer data program and the inventory analysis program could both use process initiation statements to invoke the report-formatting and printing program and cause it to create reports using the data collected.

Tasking is only one of the methods that the system provides for allowing code to be reused by different programs. Some of the other methods are

- Compile-time options

  You can use a $INCLUDE option in a program source file. At compilation, the compiler inserts text from a separate source file specified by the $INCLUDE option. This option is discussed in the manuals for each programming language.
- Binding

  This technique enables you to insert a compiled procedure from one object code file into a separate object code file. This technique is documented in the *Binder Programming Reference Manual.*
- Libraries

  This technique enables a process to dynamically invoke a procedure in another running process. This technique is described in Section 18, “Using Libraries.”

All of these methods have their virtues. Compared to the $INCLUDE option or binding, tasking has the advantage of enabling you to maintain the shared module separately from the programs that call on it. You can make changes to the module without having to recompile another program or rerun the Binder.

On the other hand, both the $INCLUDE option and binding have the advantage of enabling you to insert an external procedure directly into the source or object program. Because the inserted procedure is treated by the system as an internal procedure, the main program can enter the procedure rather than initiating it. This results in savings of processor time and memory.

Compared to libraries, tasking has a slight performance advantage in some situations. Initiating a program carries a certain cost in terms of processor time, memory, and so on. The cost of entering a library procedure varies, and can be higher or lower than the cost of initiating a process. For the first call on a particular library, the system must initiate the library process and establish a linkage between the calling program and the library. Once the library is running, it is more economical to enter a library procedure than to initiate a process.

Another advantage of the tasking method arises in situations where there already exists a program that performs a function needed by your application. You can initiate that program as a process without having to rewrite or recompile the program that performs the function. Changing the program into a library would require rewriting, and binding the program into another program requires using the Binder utility.

Using Multiple Languages in an Application

Different programming languages have different unique capabilities. These might make it easier to implement some types of routines in one language, and other types of routines in another language. If the same application requires routines in two or more different languages, then those routines have to be stored in separate source files and compiled separately.

One way to enable an application to use modules written in different languages is through tasking. You can accomplish this by using statements that initiate separate object code files. For example, you can write a COBOL program that initiates another program written in ALGOL.

A nice thing about this technique is that the system also enables you to pass parameters between programs written in different languages. The operating system allows parameters to match as long as they are of compatible types. Section 17, “Using Parameters,” explains which parameter types are considered compatible by the operating system.

Alternatively, you could use binding or libraries to create an application that uses modules written in different languages. The advantages of using tasking instead of binding or libraries are introduced under “Using Programs as Modules” earlier in this section.

Improving Application Performance

The definition of *performance* for an application has two general aspects: measurements of the resource usage of an application and measurements of the elapsed time of the application. Resource usage includes total processor time, average memory usage, and so on. Elapsed time means the total clock time a batch program takes to run, or the average time an online program takes to respond to a transaction.

If you find that the elapsed time of an application is of crucial importance to your business, you can use tasking features to help decrease the elapsed time by allowing the application to use system resources more intensively. The two features that allow you to do this are process priorities and parallel processing.

The system is designed to be able to execute large numbers of processes simultaneously. However, each central processor can execute only one process at a time. The operating system frequently reevaluates the processes waiting for processor service, and assigns the processor to the process with the highest priority. You can use task attributes and system commands to control some aspects of process priority, as discussed in Section 7, “Controlling Processor Usage.”

Parallel processing consists of dividing your application into two or more processes that run concurrently. Parallel processing enables the application to use system resources more intensively than a single process can. This increased intensity of system resource usage results because each process typically alternates among using the central processor, I/O processor, and other resources. With parallel processes, one process can use the central processor while the other is waiting for an I/O to complete, and so on.

You can create parallel processes by designing one process to initiate another process of type PROCESS or type RUN. These process types are discussed in Section 2, “Understanding Interprocess Relationships.”

Limitations of Tasking

If you do not need any of the benefits of tasking described in the preceding subsection, you can simply implement your entire application as a single program, and use only procedure entrance statements rather than procedure initiation statements. Procedure entrance uses fewer system resources than procedure initiation, and allows your application to complete faster and interfere less with other running applications.

Some of the expenses involved in initiating a procedure are

- It takes slightly more processor time than entering a procedure.
- It causes several hundred words of save memory to be allocated for the new process stack.
- It causes the system to create additional system log entries, and thus adds to general system overhead.
- It adds to the number of entries visible to the operator in a mix display. It thus tends to complicate the system operator's efforts to monitor the system.

The performance differences between entering and initiating a procedure are not great if the procedure is to be executed only once. However, for a procedure that is invoked many times, the performance loss can slow an application noticeably.

Section 2
Understanding Interprocess Relationships

The relationship between a process and its initiator is defined in terms of three major properties, which are defined in the following subsections. These properties are inclusion, flow of control, and dependency. These properties affect the speed and efficiency with which a process is executed, and the ability of the initiator to interact with the process. You can control these properties in two ways:

- By choosing among the various process-initiation statements that are available
- By choosing a program structure appropriate to the type of process desired

This section examines these choices and their implications for a family of processes.

Several of the discussions that follow refer to the term *parent*. This term is defined fully under "Dependency" in this section. For now, it is enough to know that the initiator of a process is usually also the parent of that process.

Inclusion

Section 1, "Understanding Basic Tasking Concepts," introduced the distinction between internal and external procedures, and the concept that initiating procedures results in internal or external processes. The differing properties of internal and external processes are referred to in this guide as *inclusion* properties and are as follows:

- An internal process must be dependent. Similarly, external processes resulting from initiating library procedures or passed external procedures must be dependent. Only external processes resulting from initiating separate programs can be either dependent or independent. Any attempt to initiate a procedure that is not a separate program as an independent process causes the error "NON - EXTERNAL RUN." For an explanation of the difference between dependent and independent processes, refer to "Dependency" in this section.
- In ALGOL and WFL, internal procedures have access to variables declared globally in the program. These global variables can serve as a medium for interprocess communication if the internal procedure is initiated. For information about this interprocess communication technique, refer to Section 15, "Using Global Objects."
- Several task attributes inherited by internal processes are not inherited by external processes. These task attributes include LIBRARY, NAME, OPTION, STACKSIZE, and TADS. For a discussion of task attribute inheritance, refer to the *Task Attributes Programming Reference Manual*.

Flow of Control

In Section 1, “Understanding Basic Tasking Concepts,” control was defined as the path execution takes among the various statements of a program. In a broader sense, control is the path execution takes among the statements of a procedure and any procedures initiated by that procedure. The programmer specifies the type of control path used by choosing the corresponding process initiation statement.

The control path determines whether the initiating process and new process execute in parallel or by taking turns. If they are executing by turns, the control path specifies when and how often they take turns before the new process terminates. The following subsections discuss the types of control paths that are available.

Synchronous Processes

When a synchronous process is initiated, control is transferred from the initiating process to the new process. In other words, the initiating process stops executing and the new process begins executing. The initiating process is still considered active during this period and its process stack still exists. When the new process terminates, the initiating process begins executing again, starting with the first statement after the process initiation statement.

Examples of statements that initiate synchronous processes are the CALL statement in ALGOL or COBOL and the RUN statement in Work Flow Language (WFL). Synchronous processes are sometimes referred to as coroutines, but more properly the term *coroutine* has a different use. (Refer to “Coroutines” in this section for details.)

The initiating process can set the attributes of a synchronous process only at initiation time and can interrogate the attributes only after the synchronous process has terminated.

Synchronous processes can be simpler to design than coroutines or asynchronous processes because you do not have to deal with certain complexities of timing that arise for these other types of processes.

Asynchronous Processes

When an asynchronous process is initiated, the necessary memory structures are created for the new process. Thereafter, the new process and the initiator execute in parallel. Although they execute at the same time, they do not necessarily execute at the same speed. It is for this reason that the new process is called *asynchronous.*

Examples of statements that initiate asynchronous processes are the PROCESS statement in ALGOL or COBOL, and the PROCESS RUN or PROCESS <subroutine> statement in WFL.

Asynchronous processes are useful because, in many situations, two or more processes running in parallel can do needed work in less elapsed time than a single process. What is saved in elapsed time does not necessarily translate into savings in processor or I/O time, however.

The task attributes of an asynchronous process can be read or assigned by its initiator while the asynchronous process is executing. This makes it possible for the initiator to intervene in the execution of the asynchronous process.

A disadvantage to initiating processes asynchronously is that, except in WFL, the programmer must take special measures to prevent a critical block exit error from occurring. (See the discussion of "Critical Blocks" in this section.)

In addition, initiating processes asynchronously can create ambiguous timing situations because it is impossible to predict exactly how long a process will take to execute. If an asynchronous process and its initiator share a data item, such as a global variable, and both change the value of that data item, it will be difficult to predict the order in which the changes will occur.

Various methods are used to regulate the timing of asynchronous processes. These methods are discussed in Section 16, "Using Events and Interlocks."

Coroutines

The term *coroutines* refers to a group of processes that exist simultaneously but take turns executing, so that only one of the processes is executing at any given time. Coroutines offer some of the advantages of asynchronous processes, but generally are easier to design because coroutines execute in a sequential order that prevents any ambiguities of timing. The use of coroutines offers the following benefits:

- The ability to execute a procedure repeatedly without incurring the processor time required to enter or initiate the procedure each time
- The ability to execute a procedure repeatedly without losing the values of objects declared in the procedure between each execution

Note, however, that coroutines use the processor less efficiently than do asynchronous processes. Only one coroutine runs at a time, and there might be periods when the processor is unused because the coroutine is waiting for an I/O operation to complete. Furthermore, the statements coroutines use to transfer control to other coroutines use more processor time than the event-related functions that asynchronous processes can use to suspend or resume each other.

Creating Coroutines

An ALGOL or COBOL process can create a coroutine by executing a CALL statement. The new process and its initiator are referred to as coroutines. When the initiator executes a CALL statement, the initiator temporarily ceases execution and its stack state becomes "TO BE CONTINUED". The stack state can be displayed by using the Y (Status Interrogate) system command. A coroutine with this stack state is referred to as a *continuable coroutine.*

The new process has one of the stack states that indicate the process is being processed, or soon will be, such as ALIVE or READY. The new process is referred to as an *active coroutine.*

The total number of coroutines increases each time an active coroutine executes a CALL statement. The new process created is an active coroutine and all others are continuable coroutines.

The concept of a coroutine is closely related to that of a synchronous process, as defined in “Synchronous Processes” in this section. Every synchronous process is also a coroutine; however, not every coroutine is a synchronous process. An asynchronous process can execute a CALL statement and thus become a continuable coroutine.

Using Continue Statements

An active coroutine can transfer control to a continuable coroutine by executing an ALGOL *CONTINUE* statement or a COBOL *CONTINUE* or *EXIT PROGRAM* statement. For convenience, these are all referred to as *continue statements* in the following discussion.

The other programming languages (FORTRAN77, Pascal, RPG, and WFL) do not provide continue statements. Therefore, processes other than ALGOL or COBOL processes can be considered coroutines only in a restricted sense. For example, a WFL job can create a synchronous offspring by executing a RUN statement. The stack state of the WFL job then becomes “TO BE CONTINUED.” However, the system does not allow the offspring to use a continue statement to transfer control to the WFL job. Instead, the system automatically continues the WFL job when the offspring terminates. This act is referred to as an *implicit continue* and is discussed further in “Continuing the Partner Process” in this section.

To understand the effects of a continue statement, suppose an active coroutine called A executes a continue statement that specifies a continuable coroutine called B. When the continue statement is executed, the coroutine A ceases execution and coroutine B resumes execution. In other words, coroutine A becomes a continuable coroutine and coroutine B becomes an active one. Control passes from coroutine A to coroutine B.

Coroutine B can later reverse this situation by executing a continue statement that passes control back to coroutine A. However, control does not always have to pass back and forth between the same pair of processes. For example, coroutine B might continue another coroutine called C and that coroutine might then continue coroutine A.

Control can pass between coroutines any number of times. In the course of its lifetime, a coroutine can execute many continue statements applying to any number of other processes. However, for a continue statement to be successful, it must be executed by an active coroutine and it must specify a continuable coroutine. The continue statement results in an “ILLEGAL VISIT” error if it transfers control to a process that is not a continuable coroutine.

Coroutines usually belong to the same process family because continue statements must explicitly or implicitly specify the task variable of the process to be continued. A process usually has access only to the task variables of processes in its own process family. Process families are defined under “Process Families” in this section. The means of accessing the task variables of related processes are discussed under “Accessing Task Variables” in this section.

Determining Where Execution Resumes

When any coroutine continues an ALGOL coroutine, the ALGOL coroutine resumes at the point where it left off. Thus, if an ALGOL coroutine executes a CALL statement, it later resumes with the first statement after the CALL statement. If an ALGOL coroutine executes a CONTINUE statement, it later resumes with the first statement after the CONTINUE statement.

By contrast, a COBOL coroutine can resume execution at either of two points. If a COBOL coroutine executes a CONTINUE statement or an EXIT PROGRAM RETURN HERE statement, then the coroutine later resumes at the point where it left off. However, if a COBOL coroutine executes a simple EXIT PROGRAM statement, then the coroutine later resumes with the first statement in the program. (Certain limitations on the EXIT PROGRAM statement are discussed under "Continuing the Partner Process" later in this section.)

Block Structure and Coroutines

Continue statements can occur in any of the procedures executed by a process. For example, a process can execute a continue statement and, after being continued later on, can enter another procedure and execute another continue statement. Both of those continue statements can transfer control to the same coroutine, or they can transfer control to different coroutines.

If a coroutine uses a continue statement to resume its parent, and the parent exits the critical block for that coroutine, then the parent is terminated with a "CRITICAL BLOCK EXIT" error. The methods of preventing a critical block exit are discussed under "Critical Blocks" in this section.

Continuing the Partner Process

There are two types of continue statements: specific continue statements and general continue statements.

A specific continue statement is one that specifies a task variable. An ALGOL example of a specific continue statement is CONTINUE (T1). A COBOL example of a specific continue statement is CONTINUE T1. Either of these statements continues the coroutine specified by the task variable T1.

A general continue statement does not specify a task variable. In ALGOL, the general continue statement is CONTINUE. In COBOL, the general continue statement is EXIT PROGRAM or EXIT PROGRAM RETURN HERE.

The effect of the general continue statement is to continue the *partner process.* The partner process is the process specified by the PARTNER task attribute. This task attribute is said to be *task-valued* because it accesses the task variable of a particular process. For a synchronous process, the system assigns the initiating process as the partner process by default. You can design a program to assign a different task variable to the PARTNER task attribute. Thereafter, any general continue statements affect the process with that task variable.

When a synchronous process terminates, the system implicitly continues the partner process. This is the reason the initiating process usually resumes after a synchronous process terminates. However, if a synchronous process has another task variable assigned to the PARTNER task attribute, then the system continues that partner process rather than the initiating process.

Setting the PARTNER task attribute to a process other than the initiator is not recommended. Such a practice causes general continue statements or implicit continues to consume more processor time than they otherwise would. This practice also leads to source code that is difficult to understand and maintain.

A process can interrogate the PARTNEREXISTS task attribute to determine whether the current partner process is in a continuable state. This can be a useful method for avoiding "ILLEGAL VISIT" errors.

For further information regarding the PARTNER and PARTNEREXISTS task attributes, see the discussions of these attributes in the *Task Attributes Programming Reference Manual*.

Communication between Coroutines

When an active coroutine becomes a continuable coroutine, or vice versa, objects declared by the coroutine retain their values and are not reinitialized.

Nevertheless, the values of objects declared by a continuable coroutine can be changed by any active coroutine having access to those objects. For example, if a process executes a CALL statement, passing call-by-reference parameters, the process becomes a continuable coroutine. The offspring process is an active coroutine and can change the values of the call-by-reference parameters. The offspring process can use this method to communicate information to the parent process. When the parent process is continued, it can check to see if the parameter values were changed.

Similar considerations apply to the task attributes of a coroutine. An active coroutine can read or assign the task attributes of other coroutines, including continuable coroutines. When a continuable coroutine is continued, it can check its task attribute values to see if any were changed.

Complex Coroutine Structures

The continue statements enable you to develop complex coroutine structures that do not exactly correspond to the classical model of coroutines. A complex coroutine structure is one in which two or more active coroutines exist at the same time. In a simple coroutine structure, only one of the coroutines is active at a time.

A complex coroutine structure can result, for example, if a process called INITP initiates an asynchronous offspring called PROCP, and then initiates a synchronous offspring called CALLP. While INITP is waiting for CALLP to complete, INITP is in a "TO BE CONTINUED" state. PROCP can, therefore, execute a continue statement that causes INITP to resume. In this case, PROCP becomes a continuable coroutine and INITP and CALLP are active coroutines at the same time.

In general, the use of complex coroutine structures is not recommended because they lack the simplicity that is the primary benefit of using coroutines.

Dependency

The last of the three main properties the programmer can specify for a process is *dependency.* To understand the concept of dependency, the programmer must first be familiar with the following related concepts.

- Critical objects

  Every process makes use of certain objects originally declared by another process. These include the task variable, the procedure the process is executing, and any objects passed as actual parameters to the process. In this guide, these objects are referred to as the critical objects of the process.

- Parents

  When a process is initiated, it receives these critical objects from a process called the parent. In most cases, the initiator of a process is also the parent of that process. The exact method for determining which process is the parent of a particular process is given under “Critical Blocks” later in this section.

Dependency is the relationship between a process and its parent that determines how these critical objects are stored. For an independent process, the system creates copies of these critical objects when the process is initiated. For a dependent process, the system creates references to the objects stored by the parent.

The programmer can specify the dependency of a process by choosing an appropriate process initiation statement. The dependency of a process remains the same throughout execution; if it is initiated as dependent, it cannot later become independent, or vice versa.

To initiate an independent process, you can use an ALGOL or COBOL *RUN* statement or a ??RUN (Run Code File) system command. Also, a WFL job submitted through a START statement is executed as an independent process.

To initiate a dependent process, you can use a CALL or PROCESS statement in ALGOL or COBOL, or a RUN statement in Command and Edit (CANDE), Menu-Assisted Resource Control (MARC), or WFL.

Many implications result from the choice to initiate a process as dependent or independent. However, the most crucial difference is that an independent process can continue to exist after its parent has terminated. A dependent process must terminate before its parent does.

The second most crucial difference between dependent and independent processes is that a dependent process and its parent can communicate through shared objects, whereas an independent process and its parent cannot.

Communications Effects

Some objects declared by the parent process can be shared with a dependent process, but not with an independent process.

For example, a parent can declare a task variable and include it in a process initiation statement executed by the parent. For a dependent process, the task variable remains associated with the process for as long as the process exists. After the dependent process terminates, the task variable continues to store the final task attribute values of the dependent process (though later assignments can change these values). The parent can use the task variable to access the task attributes of the process before initiation, while the process is in use, or after the process terminates. However, for an independent process, the task variable ceases to be associated with the process once initiation is complete. Only task attributes assigned to the task variable before initiation have any effect on the independent process.

Similarly, a procedure declared in the parent can be initiated only as a dependent process. A separate program, on the other hand, can be initiated as a dependent or independent process. Thus, an independent process is always an external process.

Like any external process, an independent process is unable to access objects declared globally in the parent. On the other hand, a dependent process, if it is also internal, can access objects declared globally in the parent.

Finally, any parameters passed to an independent process must be passed by value. A dependent process can be passed parameters by name, by reference, or by value.

Flow of Control Effects – Synchronization

The dependency of a process affects the ability of the process to be synchronous or asynchronous, and the ability of the parent to exit certain blocks without incurring an error.

An independent process is always asynchronous. The initiator of an independent process continues execution without waiting for the independent process to terminate. By contrast, a dependent process can be synchronous or asynchronous, depending on the type of initiation statement that is used. Another difference is that an independent process can continue executing after its parent has terminated, whereas a dependent process must terminate before its parent does.

Flow of Control Effects – Critical Blocks

Another flow of control issue related to dependency is the prevention of critical block exits. To understand exactly what a critical block exit is and why it is important, you must first understand the following basic concepts:

- Critical objects

  This concept is introduced under "Dependency" in this section. You should be aware that the critical objects of a process can be stored in more than one process stack, and they can be stored in more than one activation record in a process stack. If any block that declares one of these critical objects is exited, the corresponding activation record is removed and that critical object ceases to exist. This block exit causes the process using that critical object to terminate abnormally.

- Critical block

  This is a block that includes a definition of at least one critical object and is so positioned that it is normally exited before any other blocks that declare critical objects are exited. If you ensure the parent does not exit the critical block prematurely, then the other blocks declaring critical objects also are not exited prematurely.

At this point, the definition of a parent can be further refined as follows: the parent is the process that owns the critical block of a specified process. In other words, the parent has entered the critical block and not yet exited that block. A dependent process is said to be an *offspring* of its parent.

You need to be concerned with the critical block for a process only if that process is an asynchronous dependent process or a coroutine. If the process is either of these, you must take steps to ensure the critical block is not exited before the process terminates.

By contrast, if a process is independent, it is not affected by critical block exits. If the process is synchronous, then the parent ceases execution until the process terminates and therefore has no opportunity to exit the critical block prematurely.

Effects of a Critical Block Exit

When a parent exits an offspring's critical block, the parent is discontinued and the error message "CRITICAL BLOCK EXIT" is displayed. When the parent terminates, all its offspring processes currently in use are discontinued and a "PARENT PROCESS TERMINATED" error message is displayed.

Defining the Critical Block

The critical block of a process usually occurs somewhere in the program containing the statement that initiated the process. Within that program, the critical block is the procedure of the highest lexical level that contains any of the following items:

- The declaration of the task variable specified in the process initiation statement.
- The declaration of the procedure that was initiated, if it is an internal procedure, a passed external procedure, or an imported library procedure. The position of a declared external procedure has no effect on the critical block definition.
- The declarations of any actual parameters passed to the process by name or by reference. (For call-by-value parameters, the declaration of the actual parameter does not affect the critical block definition.)
- Any *thunk* generated for the process by the compiler. A thunk, also referred to as an accidental entry, is generated if the procedure initiation statement passes a constant or an expression to a call-by-name parameter. The thunk is located in the procedure containing the procedure-initiation statement. For an illustration of the effect of a thunk on the critical block definition, refer to Example 3 in “Critical Block Examples” later in this section.

Note that the definition of the critical block can be affected if any of the critical objects are passed as parameters from one procedure to another. If a critical object is passed as a parameter to a procedure, then for purposes of defining the critical block, the formal parameter receiving the critical object must be considered the declaration of that critical object. For an illustration, refer to Example 4 in “Critical Block Examples” later in this section.

There is one exception to the rule about the effects of passing critical objects as parameters. If a task variable is passed as a parameter to an external procedure, the critical block is affected by the declaration of the actual parameter rather than the formal parameter. This exception holds true for all types of external procedures: separate programs, passed external procedures, and imported library procedures. This exception also makes it possible for the procedure-initiation statement to reside in a different program than the critical block does. For an illustration, refer to Example 5 in “Critical Block Examples” later in this section.

The initiator of a process might or might not also be the parent of that process. This issue is illustrated by Examples 1 and 2 in “Critical Block Examples” later in this section.

TYPE Declarations and Critical Blocks

ALGOL and NEWP support special declarations called TYPE declarations. TYPE declarations define a customized class of variables. Later declarations can declare particular variables that are instances of that TYPE. If a variable based on a TYPE is passed to a dependent process as a by-name or by-reference parameter, then the critical block is affected by the location where the particular variable is declared and not the location where the TYPE is declared.

Among the TYPE declarations supported by ALGOL and NEWP is a TYPE called a *structure block.* Each structure block can include multiple objects of varying kinds, among which could be one or more of the critical objects for a task. The effect of such objects on the critical block definition is determined by the location of the structure block instance, not the structure block type declaration.

For example, suppose there is a structure block type declaration called TASKSTUFF that includes a task variable TVAR. Further, there is a structure block instance called THIS_SB, and an asynchronous task is initiated by a statement such as PROCESS PROC1 [THIS_SB.TVAR]. For purposes of critical block definition, the task variable declaration is considered to reside in the block where THIS_SB is declared.

ALGOL and NEWP also support a TYPE declaration called a *connection block.* Like a structure block, a connection block can include multiple objects of varying types. However, a connection block is intended for use by connection library declarations. For purposes of critical block definition, any critical objects declared in a connection block are considered to reside where the connection library is declared, not where the connection block is declared.

Preventing ALGOL Critical Block Exits

In ALGOL, the programmer can prevent a critical block exit by including a statement such as the following at the end of the critical block:

```
WHILE T.STATUS GTR VALUE(TERMINATED) DO
     WAITANDRESET(MYSELF.EXCEPTIONEVENT);
```

In this example, T is the task variable of the dependent process. This statement causes the parent to wait on its own EXCEPTIONEVENT task attribute, which is automatically caused by the system whenever the offspring changes status. The program then checks the status of the offspring and returns to a waiting state if the offspring has not yet terminated.

Preventing COBOL Critical Block Exits

The paragraph and section structures supported by COBOL74 and COBOL85 are not blocks and therefore do not affect the critical block definition. Similarly, nested programs in COBOL85 are also not blocks and do not affect the critical block definition. A COBOL process cannot receive a critical block error for exiting any of these types of structures. However, a COBOL process can incur a critical block exit error if the process

- Terminates while one of its offspring is in-use
- Exits a bound-in procedure that is the critical block for an offspring
- Exits an imported library procedure that is the critical block for an offspring

Statements such as the following can be included at the end of a COBOL program to prevent it from terminating before an offspring terminates:

```
PROCWAIT SECTION.
P2.
    WAIT AND RESET UNTIL ATTRIBUTE EXCEPTIONEVENT OF MYSELF.
    IF ATTRIBUTE STATUS OF TASK-VAR-1 IS GREATER THAN
       VALUE TERMINATED THEN GO PROCWAIT.
STOP RUN.
```

The preceding example assumes that an asynchronous offspring was initiated using task variable TASK-VAR-1. The COBOL program waits on its own EXCEPTIONEVENT task attribute, which is automatically caused whenever the offspring changes status. The program then checks the status of the offspring and returns to a waiting state if the offspring has not yet terminated.

Note: *The implementation of nested program structures by the COBOL85 compiler is subject to change. Nested programs may affect the critical block definition in future releases.*

Automatic Protection from WFL Critical Block Exits

The programmer does not need to include any special statements in WFL jobs to prevent critical block exits. WFL implicitly waits for the termination of asynchronous processes initiated by the job. The implicit wait occurs at the end of the subroutine that executed the process initiation statement.

Critical Block Examples

The following examples illustrate various factors that affect the definition of the critical block for a process. The more typical cases are presented first.

Example 1

In most cases, the initiator of a process is also the parent of that process. However, this is not always the case. The following ALGOL program is an illustration of the difference between the parent and the initiator:

```
PROCEDURE TRUEPARENT;
BEGIN
   TASK T1, T2;
   REAL I;

   PROCEDURE WAITFOR(T);
   TASK T;
   BEGIN
      WHILE T.STATUS GTR VALUE(TERMINATED) DO
         WAITANDRESET(MYSELF.EXCEPTIONEVENT);
   END;

   PROCEDURE OFFSPRING(X);
   REAL X;
   BEGIN
      X:= 1;
   END;

   PROCEDURE INITIATOR;
   BEGIN
      PROCESS OFFSPRING(I) [T2];
   END;

   PROCESS INITIATOR [T1];
   WAITFOR(T1);
   WAITFOR(T2);
END.
```

In this example, the procedure TRUEPARENT initiates the procedure INITIATOR as an asynchronous process. INITIATOR then initiates the procedure named OFFSPRING. In this situation, the initiator of OFFSPRING is INITIATOR, but the parent is TRUEPARENT.

TRUEPARENT is considered the parent because the declarations of the procedure OFFSPRING, the task variable T2, and the actual parameter I all occur in the outer block of TRUEPARENT.

Example 2

In the following ALGOL example, the process called INITIATOR is both the initiator and the parent of the process named OFFSPRING. INITIATOR is considered the initiator because INITIATOR includes the task initiation statement that initiates the OFFSPRING procedure. INITIATOR is considered the critical block for OFFSPRING because the task initiation statement passes OFFSPRING a parameter declared within INITIATOR. An invocation of the WAITFOR procedure is added to INITIATOR to prevent a critical block exit.

```
PROCEDURE OUTERBLOCK;
BEGIN
   TASK T1, T2;

   PROCEDURE WAITFOR(T);
   TASK T;
   BEGIN
      WHILE T.STATUS GTR VALUE(TERMINATED) DO
         WAITANDRESET(MYSELF.EXCEPTIONEVENT);
   END;

   PROCEDURE OFFSPRING(X);
   REAL X;
   BEGIN
      X:= 1;
   END;

   PROCEDURE INITIATOR;
   BEGIN
```

```
290       REAL R;
300       PROCESS OFFSPRING(R) [T2];
310       WAITFOR(T2);
320    END;
330
340    PROCESS INITIATOR [T1];
350    WAITFOR(T1);
360 END.
```

Example 3

The following is an ALGOL example of a case where the presence of a thunk affects the critical block definition for a process:

```
100 PROCEDURE OUTERBLOCK;
110 BEGIN
120    TASK T1, T2;
130    REAL A, B, C, D;
140
150    PROCEDURE WAITFOR(T);
160    TASK T;
170    BEGIN
180       WHILE T.STATUS GTR VALUE(TERMINATED) DO
190          WAITANDRESET(MYSELF.EXCEPTIONEVENT);
200    END;
210
220    PROCEDURE OFFSPRING(X);
230    REAL X;
240    BEGIN
250       C:= X;
260    END;
270
280    PROCEDURE INITIATOR;
290    BEGIN
300       PROCESS OFFSPRING(A + B) [T2];
310       WAITFOR(T2);
320    END;
330
340    A:= 2;
350    B:= 5;
360    PROCESS INITIATOR [T1];
370    WAITFOR(T1);
380 END.
```

In the preceding example, X is a call-by-name formal parameter of the procedure OFFSPRING. The statement invoking OFFSPRING passes the expression (A + B) to the parameter. This creates a thunk at the point of the procedure initiation, which causes the INITIATOR procedure, rather than the OUTERBLOCK procedure, to be considered the critical block of OFFSPRING. Because the statement at line 360 initiates INITIATOR rather than entering it, INITIATOR becomes a separate process that is the parent of OFFSPRING.

You can avoid some thunks by making the formal parameter call-by-value rather than call-by-name. For example, you can avoid the thunk in the preceding example by adding a line to the procedure heading of the procedure OFFSPRING at line 220. The revised procedure heading appears as follows:

```
PROCEDURE OFFSPRING(X);
VALUE X;
REAL X;
```

This change has the side effect of making OUTERBLOCK the critical block, instead of INITIATOR.

Example 4

In the following ALGOL example, the location of the critical block is affected by a formal parameter specification:

```
100 PROCEDURE OUTERBLOCK;
110 BEGIN
120    TASK T1, TVAR;
130    REAL I;
140
150    PROCEDURE WAITFOR(T);
160    TASK T;
170    BEGIN
180       WHILE T.STATUS GTR VALUE(TERMINATED) DO
190          WAITANDRESET(MYSELF.EXCEPTIONEVENT);
200    END;
210
220    PROCEDURE OFFSPRING(X);
230    REAL X;
240    BEGIN
250       X:= X + 1;
260       WAIT ((10));
270    END;
280
290    PROCEDURE INITIATOR(T2);
300    TASK T2;
310    BEGIN
320       PROCESS OFFSPRING(I) [T2];
330       WAITFOR(T2);
340    END;
350
360    PROCESS INITIATOR(T1) [TVAR];
370    WAITFOR(TVAR);
380 END.
```

In this example, INITIATOR is the critical block for the procedure OFFSPRING, because the task variable T2 is declared in the procedure heading of INITIATOR. It makes no difference that the actual parameter T1 is declared in the outer block. It is the formal parameter T2 that is mentioned in the procedure invocation statement, and therefore the declaration of T2 takes precedence.

Example 5

In the following ALGOL examples, the critical block is located in a different program than the one that contains the process-initiation statement. The following is program OBJECT/CALL:

```
100 BEGIN
110   TASK T, T1;
120   PROCEDURE OB (T);
130     TASK T;
140     EXTERNAL;
150   REPLACE T.NAME BY "OBJECT/CALL/2.";
160   PROCESS OB (T1) [T];
170   WHILE T1.STATUS GTR VALUE(TERMINATED)
180     DO WAITANDRESET (MYSELF.EXCEPTIONEVENT);
190 END.
```

The previous program initiates a separate program called OBJECT/CALL/2, passing a task variable as a parameter. The following is the program OBJECT/CALL/2:

```
100 PROCEDURE OB (T);
110   TASK T;
120 BEGIN
130 PROCEDURE X;
140   EXTERNAL;
150 REPLACE T.NAME BY "OBJECT/TASK.";
160 PROCESS X [T];
170 END.
```

The preceding program uses its task variable parameter to initiate a third program. The procedure declaration at lines 130 and 140 does not affect the critical block definition, because it is an external procedure declaration. Note that since the process initiation statement is PROCESS, and no WAIT statement follows it, the preceding program finishes executing while the third program, OBJECT/TASK, is still running. However, no CRITICAL BLOCK EXIT error occurs.

The following is the third program, OBJECT/TASK:

```
100 BEGIN
110   EBCDIC ARRAY FORMALARRAY[0:119];
120   REPLACE FORMALARRAY BY MYSELF.EXCEPTIONTASK.NAME;
130   DISPLAY (FORMALARRAY);
140   WAIT(MYSELF.ACCEPTEVENT);
150 END.
```

This program displays the name of its EXCEPTIONTASK, which, by default, is the same as the parent. The name it displays is OBJECT/CALL, which is therefore the parent. Because OBJECT/CALL is the parent, no CRITICAL BLOCK EXIT occurs when OBJECT/CALL/2 terminates.

Recall the rule about passing task variables to external procedures that is discussed under "Defining the Critical Block" earlier in this section. It is the declaration of the actual task variable parameter, at line 110 in OBJECT/CALL, that affects the critical block definition. The critical block is therefore the outer block of OBJECT/CALL. However, the process initiation statement occurs in OBJECT/CALL/2. This is the only type of situation where it is possible for the process-initiation statement and the critical block to reside in separate programs.

Process Families

A *process family* is a group of processes that have relationships based on dependency. These relationships have many effects, including effects on interprocess communication, handling of printer output, and enforcement of resource usage limits.

Familial Relationships

Each process belonging to a process family is called a *member* of that process family. Every process family includes a single independent process as its founding member. The process family also includes any dependent offspring of that independent process, any dependent offspring of those offspring, and so on

Familial terms are used to describe the relationships between the members of a process family. Of these, *parent* and *offspring* are defined under "Critical Blocks" earlier in this section. A related term is *sibling.* Offspring processes that have the same parent are referred to as siblings.

Each offspring of a process is considered a *descendant* of the process. Any offspring of the descendants of a process are also considered descendants of the original process.

Conversely, the parent of a process is considered to be an *ancestor* of the process, and any ancestors of the parent are also considered to be ancestors of the same process. Processes having a common ancestor, but not a common parent, are referred to as *cousins.* The independent process in a process family is the common ancestor of all the processes in that family.

Finally, processes are considered *related* if they belong to the same process family, and *unrelated* if they do not.

A dependent process is dependent on the continued existence of all its ancestors, not only its parent. This is true because a type of domino effect occurs if any of the ancestors terminates. The immediate offspring of the terminated process are discontinued with a "PARENT PROCESS TERMINATED" error. The offspring of the discontinued processes are, in turn, discontinued with the same error, and so on.

In contrast, a member of a process family does not depend on the continued existence of any of its descendants. For example, the descendants of a process can terminate abnormally without affecting the process.

Jobs and Tasks

The independent process in a process family is called the *job* for that family. The dependent processes in a process family are referred to as *tasks*.

Note that, in some older publications, you might find the term *task* used with a different meaning than the one defined here. In addition to the meaning given here, *task* has sometimes been used to refer to any process, to the offspring of some particular process, or to any discrete unit of work. These usages are generally avoided in this guide, except in the terms *task attribute* and *task variable*, which have been retained because they are well known. (More properly, these terms would be *process attribute* and *process variable* because they can apply to either jobs or tasks.)

Certain services that the system provides for a process family are linked to the job for the family. The job provides the following services:

- Job logging

  The job has a job file associated with it that stores the job log. The job log includes information about the activities of all the processes in the process family. When the job terminates, the system can issue a printout of the job log, called the job summary. (The job file for a WFL job includes additional information, which is described under "Special Types of Jobs" later in this section.)

- Printer output

  By default, any printer backup files created by process family members are saved until the job terminates. The system groups these files into a single entry in the print queue, unless the files have incompatible print attributes, such as different DESTINATION values. For further information about print requests, refer to the *Print System and Remote Print System Administration, Operations, and Programming Guide*.

Operators or programmers can use the following means to determine whether a process is a job or a task:

- Process messages

  The system displays a "BOJ" message when a job is initiated and an "EOJ" message when the job is terminated. For a task, the corresponding messages are "BOT" and "EOT."

- Job displays

  The J (Job and Task Display) system command displays all the process families that currently exist. The members of each process family appear in hierarchical order, beginning with the job.

- Job number

  A task has a job number that differs from the mix number and indicates the job or session associated with the task. For a job, the job number and mix number are equal. The operator can see the job number and mix number in the output of many system commands. A process can also read these values from the JOBNUMBER and MIXNUMBER task attributes.

- Process type

  A process can determine whether a particular process is a job by reading the TYPE task attribute. For WFL jobs, the value is JOBSTACK; for other jobs, the value is RUN. For tasks, the value is CALL or PROCESS.

Special Types of Jobs

The following subsections describe WFL jobs, BDBASE tasks, and MCS sessions, all of which are special types of jobs and entities that resemble jobs.

WFL Jobs

A program written in WFL is usually executed as an independent process. Because of this, the execution of a WFL program is referred to as a *WFL job*. The TYPE task attribute of a WFL job usually has a value of JOBSTACK.

When a WFL job is submitted from one of the available sources, the system initiates the WFL compiler. (The sources for submitting WFL jobs include START commands in CANDE and MARC sessions, and various statements in programming languages.) The WFL compiler creates the job file for the WFL job.

The job file for a WFL job contains several kinds of information not included in the job file for any other kind of job. In addition to the logging information, a WFL job file includes the following:

- A copy of the WFL source program.
- Object code for the job. The job file also serves as the code file for a WFL job.
- Data specifications used by the job. A data specification is a portion of the WFL source program that can be used as an input file by one or more of the offspring of the job.
- Job restart information.

WFL jobs have several other properties not shared by any other type of process. For details, refer to Section 4, "Tasking from Programming Languages."

BDBASE Tasks

Setting the BDBASE option of the OPTION task attribute causes a task to assume some characteristics of a job. The exact effects of the BDBASE option depend on whether it is assigned before or after initiation of the task. If BDBASE is assigned before task initiation, then the task receives the following joblike characteristics:

- Its own job file.
- Ability to produce a job summary.
- A mix number equal to its job number.
- "BOJ" and "EOJ" messages.
- Automatic printing, when the BDBASE task terminates, of any backup files created by the BDBASE task or its descendants. Note that this behavior applies only to backup files whose PRINTDISPOSITION file attribute has the default value of EOJ.

If BDBASE is assigned after task initiation, then its only effect is to cause default printing of backup files when the task terminates. Even if BDBASE is assigned before initiation, it does not make the task into a true job. A BDBASE task differs from a job in the following ways:

- The BDBASE task usually is not an independent process. (There is no point in setting BDBASE for an independent process, because such a process already has all job capabilities.)
- The JOBNUMBER value for a descendant of a BDBASE task does not equal the MIXNUMBER of the BDBASE task. Rather, the JOBNUMBER equals the MIXNUMBER of the job at the head of the process family.
- The MYJOB task variable never refers to a BDBASE task. For details, refer to "MYJOB Task Variable" in this section.

In the past, the main use of the BDBASE option was to cause printer backup files produced by a task to print when the task terminated, rather than being saved until the job terminated. However, other Print System features now enable you to provide the same control over printing, without assigning any other joblike characteristics to the task. For further information, refer to the discussion of printing in Section 9, "Controlling Process I/O Usage."

MCS Sessions

CANDE and MARC sessions have the following job characteristics:

- Job summaries that are produced at the end of the session and that summarize the activities of all tasks initiated from the session
- Default printing, when the session ends, of backup files produced by tasks initiated from that session
- A mix number, also called the *session number*, that is inherited by the JOBNUMBER task attribute of tasks initiated from the session

However, CANDE and MARC sessions are not really jobs because they are not processes. Each session is merely a dialogue between the user and the CANDE or MARC software. The MYJOB task attribute has a special meaning for tasks initiated from CANDE and MARC sessions. For further information, refer to "Access to Ancestral Processes in CANDE" and "Access to Ancestral Processes in MARC" in Section 3, "Tasking from Interactive Sources."

Accessing Task Variables

The system automatically provides several task variables, called *predeclared task variables,* for use by a process. The process can use these task variables to access task attributes of certain related members of the process family.

MYSELF Task Variable

A process can access its own task attributes by way of the predeclared task variable MYSELF.

MYSELF has a special meaning for processes that are descendants of CANDE or MARC sessions. For more information, refer to Section 3, "Tasking from Interactive Sources."

MYJOB Task Variable

A process can use the predeclared task variable MYJOB to access the task attributes of its job. When a job uses MYJOB, it has the same meaning as the MYSELF task variable.

If a BDBASE task, or a descendant of a BDBASE task, uses the MYJOB task variable, MYJOB does not refer to the BDBASE task. Instead, MYJOB refers to the independent process that is the eldest ancestor of the BDBASE task and, therefore, the real head of the process family. In other words, MYJOB refers to the job.

MYJOB has a special meaning for processes that originate from CANDE or MARC sessions or from an ODT. For more information, refer to Section 3, "Tasking from Interactive Sources."

Exception Task

Every process has an associated *exception task* with which it has a special relationship. There are two aspects to this relationship:

- Whenever the value of the STATUS task attribute of the process changes, the system notifies the exception task by causing the EXCEPTIONEVENT task attribute of the exception task.
- A process can access the task attributes of its exception task by way of its own EXCEPTIONTASK task attribute. For example, the following ALGOL statement assigns a value to the TASKVALUE task attribute of the exception task:

```
MYSELF.EXCEPTIONTASK.TASKVALUE:= 5;
```

The parent of a dependent process is the default exception task of the process. An independent process, by default, is its own exception task; however, in this case, the exception task relationship embodies only the second of the aspects in the previous list. The EXCEPTIONEVENT of the independent process is not caused when the status of the independent process changes.

A dependent process can use the EXCEPTIONTASK task attribute to access the task variable of any of its ancestors. The process can specify EXCEPTIONTASK repeatedly to access ancestors two or more generations back (for example, the grandparent, great-grandparent, and so on). The following statement assigns an attribute to the grandparent of the process:

```
MYSELF.EXCEPTIONTASK.EXCEPTIONTASK.SW1:= TRUE;
```

A process can override the default exception task and assign a different process as the exception task. The following ALGOL statement specifies that the process identified by the task variable TVAR be treated as the exception task:

```
MYSELF.EXCEPTIONTASK:= TVAR;
```

The process assigned as the exception task must be the process itself or an ancestor, sibling, or cousin of the process. The exception task cannot be a descendant of the process. An attempt to assign a descendant as the exception task results in the error "UP LEVEL TASK ASSIGNMENT."

It is recommended that only the process itself or one of its ancestors be assigned as the exception task. If a sibling or cousin is assigned as the exception task, then any attempt to access the exception event of the exception task causes a "NON ANCESTRAL TASK REFERENCE" error. For example, in such a situation, the following statement would cause an error:

```
CAUSE (MYSELF.EXCEPTIONTASK.EXCEPTIONEVENT);
```

Assigning a process that is not the parent as the exception task can also have more subtle side effects. Suppose the task is called T and the parent contains a statement such as the following:

```
WHILE T.STATUS GTR VALUE(TERMINATED) DO
   WAITANDRESET(MYSELF.EXCEPTIONEVENT);
```

This statement causes the parent to wait until its exception event is caused, at which point it checks the status of T. If T has terminated, the next statement in the parent is executed. If T has not terminated, the parent goes back into a waiting state.

The problem is that, if a parent is not also the exception task for its offspring, then any changes in the offspring's status do not cause the parent's exception event. Instead, changes in the task's status cause the exception event of the process assigned as the exception task. Therefore, the parent continues waiting indefinitely, regardless of any changes in the task's status.

Sometimes, however, it is not desirable for the exception event of a process to be caused whenever the status of any of its offspring changes. For example, the process might be waiting for a HI (Cause EXCEPTIONEVENT) system command. In this case, each of the offspring could be assigned itself as its exception task. This assignment prevents any of the offspring from accidentally causing its parent's exception event.

The MCS that controls a session is the parent of any tasks initiated from that session. By default, therefore, the MCS is also the exception task for any tasks initiated from that session.

Partner Processes

The *partner process* is the process specified by the task-valued task attribute PARTNER. For a synchronous process, the default value of this attribute is the initiator. However, a process can assign any task variable to this attribute. A process can use the PARTNER task attribute as a convenient means of accessing the task attributes of the partner process. For example, the following ALGOL statement assigns a value to the TASKVALUE task attribute of the partner process:

```
MYSELF.PARTNER.TASKVALUE:= 3;
```

The partner process has a special significance for coroutines. For details, refer to “Continuing the Partner Process” in this section.

Other Task Variables

A programmer can make it possible for two sibling or cousin processes to access each other's task variables by declaring the task variables in a common ancestor of the two processes. Internal processes can access task variables that are declared globally in the same object code file as the internal procedure declaration. Task variables can also be passed as parameters to offspring processes.

Private Processes

A *private process* is a process whose task attributes cannot be altered by any of its descendant processes. Assigning the private process option to the OPTION task attribute causes the process to become a private process. Any descendant process that attempts to access the task attributes of a private process is terminated with the error “NON OWNER WRITE ACCESS OF A PRIVATE TASK.”

Both CANDE and MARC are private processes.

Setting Resource Limits

Any resource limits attached to a job are propagated downward through all the job's descendants. Resource limits are stored in the values of the task attributes ELAPSEDLIMIT, MAXIOTIME, MAXLINES, MAXPROCTIME, PRIORITY, RESOURCE, SAVEMEMORYLIMIT, TEMPFILELIMIT, and WAITLIMIT. Information about the amount of resources a particular process has used is stored in the task attributes ACCUMIOTIME, ACCUMPROCTIME, ELAPSEDTIME, and TEMPFILEMBYTES. If the accumulated usage of a resource rises above the maximum allowed, the process terminates abnormally. Most of these resource limits are propagated in two ways:

- When a task is initiated, by default each resource limit for the task is assigned the difference between the parent's own limit for the resource and the parent's accumulated usage of the resource. For example, if the parent's MAXPROCTIME is 100 and its ACCUMPROCTIME is 75, then the task is assigned a MAXPROCTIME of 25. The parent's own MAXPROCTIME and ACCUMPROCTIME values are not affected. The parent can assign resource limits to the task through task equation, but the values are ignored unless they specify lower limits than the task would receive by default.
- When a task terminates, the values of its accumulated usage attributes are added to the accumulated usage attributes of the task's job. If this addition causes any accumulated usage attribute of the job to be assigned a value greater than the corresponding maximum usage attribute, the job is abnormally terminated. The termination of the job in turn causes the termination of all the other members of the process family.

The resource-limiting attributes of a task cannot be set above the values of the corresponding attributes of the job. The MAXPROCTIME and MAXIOTIME task attributes can be set above the job values for an inactive task, but when the task is initiated, the values of these task attributes are automatically reduced to a value within the allowed limits.

If a job is a WFL job, then its resource-limiting attributes can inherit values specified by the queue attributes of the job queue from which the WFL job was initiated. For further information on queue limits, refer to Section 4, "Tasking from Programming Languages."

Section 3
Tasking from Interactive Sources

An interactive tasking source is one that enables you to enter at a terminal commands that initiate, monitor, and control processes. This section reviews the tasking capabilities of the most important sources for interactive tasking: Command and Edit (CANDE), Menu-Assisted Resource Control (MARC), Operations Center, and the operator display terminal (ODT).

The information in this section can help you decide which of these interfaces best serves your needs. This section also explains considerations to keep in mind when writing programs that are to be initiated from these sources.

Note that many users access applications primarily through the Transaction Server direct windows. This interface is not reviewed here because the direct window interface does not provide any direct means to control processes. Rather, the Transaction Server initiates and controls direct window programs automatically, within various parameters set by the system administrator. For information about direct window programs, refer to the *Transaction Server Programming Guide*.

CANDE

CANDE is a message control system (MCS) that enables you to interactively perform functions such as file editing, program compilation, and program execution. You initiate communications with CANDE by logging on at a terminal controlled by CANDE, or by opening a CANDE window dialogue on a terminal controlled by the Transaction Server. Your interactions with CANDE between the times you log on and log off are referred to as a *session*.

CANDE Tasking Capabilities

CANDE offers a number of process-initiation commands, as well as other commands for monitoring or controlling processes. For details about any of the commands discussed in the following subsections, refer to the *CANDE Operations Reference Manual*.

Initiating Dependent Processes from CANDE

You can initiate a task from a CANDE session by using the RUN command. (EXECUTE is a synonym for the RUN command.) The RUN command can pass only a single string parameter to a program.

The CANDE *RUN* command is unique in that it usually specifies a program by its source file title rather than by its object code file title. CANDE takes the file title specified in the RUN command and looks for an object code file with the same title, except that the object code file title is prefixed by *OBJECT/*. For example, the object code file OBJECT/TEST can be initiated by the command *RUN TEST*.

However, if you prefix the file title with a dollar sign ($), then CANDE interprets the file title as an object code file title. You can use this form of the RUN statement to initiate programs whose object code file title does not begin with *OBJECT/*. An example of such a command is *RUN $ACCOUNTS/INPUT*, which initiates the object code file named ACCOUNTS/INPUT.

CANDE also assumes that the file title is an object code file title if the file title is nonusercoded. You can indicate that a file title is nonusercoded by including an asterisk (*) at the start of the title. For example, you can initiate an object code file titled *SYSTEM/FILEDATA with the command *RUN *SYSTEM/FILEDATA*.

If you omit the file title from the RUN statement, CANDE assumes the current work file is the source program. If no object code file with the related file title exists, or if the object code file does not reflect recent changes to the work file, then CANDE automatically compiles the work file and executes the resulting object code file.

The task is asynchronous (that is, it runs in parallel with the CANDE software that initiated it). However, the process appears to the user to be a synchronous task because most CANDE commands are not available while the task is running. Only control commands (commands, such as ?Y, that start with a question mark) can be used. It is not possible to issue file maintenance or editing commands or to initiate another task until the first task terminates.

An alternative to the RUN command is the UTILITY command. The UTILITY command behaves like the RUN command in most respects. However, the UTILITY command enables you to append to it unquoted text that is passed as a string parameter to the program. If you do not append any text, the UTILITY command passes an empty string parameter. The following are examples of UTILITY commands and the equivalent RUN commands:

```
U DAILY UPDATE OUTPUT = PRINTER
RUN DAILY/UPDATE("OUTPUT=PRINTER")

U DAILY UPDATE
RUN DAILY/UPDATE("")
```

The UTILITY command also automatically passes certain task equations and file equations to the program initiated. These equations make it possible for the program to use the unsaved work file, work source, or work object associated with the session. Certain utilities, such as the Editor, are designed to accept these task and file equations. Such programs must be initiated with the UTILITY command instead of the RUN command. For details about the task and file equations that are passed, refer to the UTILITY command discussion in the *CANDE Operations Reference Manual*.

Initiating Compilations from CANDE

You can use the COMPILE command to compile a program. This command allows you to specify the compiler to use, the input file titles, the object code file title, and task equations for the compiler and the resulting object code file. For example:

```
COMPILE DAILY/UPDATE/PATCH AS DAILY/UPDATE/NEW WITH COBOL74;
  COMPILER FILE SOURCE = DAILY/UPDATE/SOURCE;
  PRIORITY = 40;
```

This example initiates the COBOL74 compiler, specifying a primary input file called DAILY/UPDATE/PATCH and a secondary input file called DAILY/UPDATE/SOURCE. The object code file that results is called OBJECT/DAILY/UPDATE/NEW. The compiler stores the PRIORITY assignment in the resulting object code file, so that OBJECT/DAILY/UPDATE/NEW receives a default PRIORITY value of 40 whenever it is run.

The COMPILE command can be used more simply than it is in the preceding example. Suppose that DAILY/UPDATE is your work file. Simply entering *COMPILE* in your CANDE session is sufficient to compile your work file. CANDE chooses the compiler that matches the file type of the source file. The resulting object code file consists of the source file title with "OBJECT/" prefixed (for example, OBJECT/DAILY/UPDATE.)

The COMPILE command cannot cause the execution of the resulting object code file. However, a simple RUN command compiles and runs the work file if no object code file exists.

Initiating Utilities from CANDE

The RUN and UTILITY commands can be used to initiate a variety of system utility programs such as FILECOPY, LOGGER, and so on. However, CANDE also includes a number of specialized commands that you can use to initiate particular utilities. The following are the commands and the names of the corresponding utilities:

Command	**Utility**
BACKUPPROCESS	Backup Processor
DCSTATUS	DCSTATUS
LFILES	FILEDATA
LOG	LOGANALYZER

Submitting WFL Jobs from CANDE

You can submit WFL programs from CANDE sessions using the START or WFL command. The START command submits a WFL program that is stored in a disk file. The WFL command enables you to enter WFL statements directly at the terminal.

The START command can pass any number or type of parameters that are expected by the WFL program. In addition, you can use the *FOR SYNTAX* clause for syntax checking. This clause causes the program to be compiled, but not executed, and displays information about any syntax errors in the WFL program. You can also assign the STARTTIME task attribute to delay initiation of the program. However, you cannot assign any other task attributes to the program.

While the WFL program is compiling, only CANDE control commands are available. If you enter any other CANDE commands during this period, CANDE queues the commands and executes them when the compilation is finished. However, after the WFL program is compiled and entered in a job queue, all CANDE commands are available again. The WFL program executes as a job and can have a job summary or printer backup files associated with it. By default, these files are queued for printing when the WFL program terminates.

You can use the WFL command to submit one or more WFL statements. Simply enter *WFL*, followed by the WFL statements. You can omit the *?BEGIN JOB* and *?END JOB* statements. The WFL statements can include all the constructs defined in WFL with the exception of data specifications and STARTTIME specifications.

When you submit WFL input by way of the WFL command, only CANDE control commands are available while the WFL input compiles and executes. Any other CANDE commands that you enter during this period are queued for later execution. By default, any backup files created by the WFL process are saved with the CANDE session. The files are queued for printing when you end the CANDE session.

The CANDE *ADD*, *COPY*, and *PRINT* commands correspond to the WFL commands of the same names. When you enter any of these commands, CANDE passes it to WFL for execution.

Monitoring and Controlling Processes in CANDE

Any messages generated by a task initiated from a CANDE session are automatically displayed at that session, including any “BOT”, “EOT”, DISPLAY, and RSVP messages and error or warning messages. However, for processes indirectly associated with a session, the display of messages is optional. Processes indirectly associated with a session include WFL processes initiated by a *START* or *WFL* command, the descendants of such processes, and the descendants of any task initiated from a session.

The CANDE session option *MSG* controls the display of messages by processes indirectly associated with a session. While MSG is set, all messages generated by such indirect processes are displayed at the session. While MSG is reset, all such messages are suppressed. CANDE sets the MSG option to TRUE if the usercode attribute CANDEGETMSG is set for the usercode of the session. You can also set the MSG option to TRUE for a session by entering a CANDE *SO MSG* command. You can use the equivalent CANDE control command, *?SO MSG*, even when the station is busy.

A number of CANDE control commands are available for monitoring and controlling particular processes. You can use these commands to monitor or control any process that has the same usercode as the session usercode. This includes processes initiated from the current session as well as processes initiated from other sources, such as MARC or the ODT.

Most CANDE commands related to process control correspond to system commands with similar names. Some restrictions and differences in spelling apply to the CANDE versions of these commands. For further information, refer to "Tasking Command Equivalents" later in this section.

The system assigns a unique number, also known as the *session number*, to each CANDE session. The session number is assigned from a range of numbers defined by the MAX (Maximums) system command. Depending on the range defined, the session number can be as low as 100 or as high as 65535.

The CANDE session number does not appear as a process in mix display commands. However, the session number does appear in the output from two system commands: Y (Status Interrogate) and C (Completed Mix Entries). The output from these commands shows both the job number and the mix number of a process. If the process is a task, and it was initiated from a CANDE session, then the job number shown is actually the CANDE session number.

Access to Task Attributes in CANDE

For each session, CANDE maintains a task variable and sets selected attributes. CANDE requests some of this information from the user at log-on time and obtains most of the rest from usercode attributes defined in the USERDATAFILE. Tasks initiated within the session inherit the attributes following typical inheritance rules. Attributes that can be set by CANDE include the following:

ACCESSCODE	JOBNUMBER	PRINTDEFAULTS
CHARGE	JOBSUMMARY	PRIORITY
CONVENTION	JOBSUMMARYTITLE	SOURCESTATION
DESTNAME	LANGUAGE	STATION
DISPLAYONLYTOMCS	NOJOBSUMMARYIO	USERCODE
FAMILY		

Accessing the CLASS attribute of the session or of a task within the session returns the queue number from which CANDE was started.

A task initiated from a CANDE session receives a JOBNUMBER value equal to the session number. The JOBNUMBER value for such a task can range from 100 to 65535, depending on the session number range established by the system command MAX SESSION = <number>.

For a WFL job started from a CANDE session, the MCP assigns a MIXNUMBER value from the mix number pool and assigns a JOBNUMBER value equal to the MIXNUMBER value. The JOBNUMBER value for such a task can range anywhere from 100 to 65535, depending on the mix number range established by the system command MAX MIX = <number>.

You can use CANDE commands to change the values of some of the session attributes. By using these commands, you create new defaults that are applied to all tasks initiated later in that session. The ACCESS, CHARGE, DESTNAME, FAMILY, and LANGUAGE commands each display or assign the session attribute of the same name. Additionally, the PDEF command displays or assigns the PRINTDEFAULTS session attribute.

CANDE supports the following methods of providing default values for the DISPLAYONLYTOMCS task attribute:

- The session option DISPLAYONLYTOMCS establishes a default DISPLAYONLYTOMCS value for all processes initiated from the current CANDE session. To establish a default of TRUE, use the command *?SO DISPLAYONLYTOMCS*. To establish a default of FALSE, use the command *?RO DISPLAYONLYTOMCS*.
- The ?DISPLAYONLYTOMCS control command establishes the initial value of the DISPLAYONLYTOMCS session option for all CANDE sessions. To cause DISPLAYONLYTOMCS to default to TRUE, use the command *?OP + DISPLAYONLYTOMCS.* To cause DISPLAYONLYTOMCS to default to FALSE, use the command *?OP- DISPLAYONLYTOMCS*.

You can also assign task attributes to specific processes by using task equations. Task equations can be appended to most CANDE process initiation statements, including RUN, UTILITY, COMPILE, and the various special-purpose commands for initiating utilities. Task equations can assign values to all but task-valued or event-valued task attributes, such as EXCEPTIONTASK or EXCEPTIONEVENT. If a task equation conflicts with task attribute inheritance, the task equation takes precedence. For example, the following CANDE command assigns to a process a LANGUAGE value different from the LANGUAGE value of the session:

```
RUN DRIVER;LANGUAGE = FRANCAIS
```

For information about the task attributes available in CANDE, refer to the *CANDE Operations Reference Manual*.

Saving CANDE Commands for Later Use

You can achieve some of the convenience of programmatic task initiation and control by saving CANDE commands in a file for later use. You can use the DO or SCHEDULE command to execute the commands in the file. You can reuse the file as many times as desired.

The DO command takes effect immediately and prevents you from using most other commands in the session until the DO file is completed. However, you can use the SCHEDULE command to cause the file to be executed separately from your current session or at a later time.

Files that store CANDE commands are different from programs in that they are not compiled and are not executed as separate processes. Their process control abilities are more limited than those of WFL, ALGOL, or COBOL programs, because no conditional statements or variables are available.

CANDE Programming Considerations

When you design a program to be run from CANDE, you need to be aware of CANDE features affecting parameter passing, task attribute access, and terminal communications.

Receiving Parameters from CANDE

If you are designing a program to be initiated from CANDE, be aware that the program can receive only one parameter from the RUN or UTILITY command that initiates it. This parameter appears as a string to the user, but in the program it must be declared as type Real Array (or compatible parameter type) with an unspecified lower bound. For information about Real Array parameters and compatible parameter types, refer to Section 17, "Using Parameters."

Access to Ancestral Processes in CANDE

If you initiate a task from a CANDE session, and that task accesses its own EXCEPTIONTASK task attribute, the system interprets EXCEPTIONTASK as a reference to the CANDE MCS. The task can use the EXCEPTIONTASK task attribute to query the values of the task attributes of the CANDE MCS. However, if the task attempts to modify the task attributes of the CANDE MCS, the task is terminated with a task attribute error. This error occurs because CANDE runs with the private process option of the OPTION task attribute set to TRUE.

Session attributes can be interrogated and set using the MYJOB task variable.

The following ALGOL example assigns a value of SUPPRESSED to the JOBSUMMARY task attribute of the session. If this program is initiated by a CANDE *RUN* command, the program prevents a job summary from being printed when the session ends.

```
BEGIN
  MYJOB.JOBSUMMARY:= VALUE(SUPPRESSED);
END.
```

The MCSNAME task attribute of tasks initiated from CANDE sessions typically returns a value of *SYSTEM/CANDE*, which might or might not be preceded by an asterisk (*). Note that the MCSNAME value can be different if CANDE was installed at your site under a different name.

Communicating with CANDE Terminals

The STATIONNAME task attribute is the preferred method of specifying the station where remote files should be opened. However, CANDE does not assign the STATIONNAME attribute of programs or WFL jobs initiated from a CANDE session. It is therefore advisable for you to explicitly assign a STATIONNAME value to any interactive processes originating from a CANDE session.

For example, you can include the following statement in WFL jobs initiated from a CANDE session:

```
MYSELF(STATIONNAME = #MYSELF(SOURCENAME));
```

This STATIONNAME value is inherited by any tasks of the WFL job, and enables them to open remote files successfully.

The STATION task attribute can be used for a similar purpose. However, the STATIONNAME task attribute is more reliable than STATION, because STATION stores a logical station number (LSN), and the LSN of any given station is subject to change.

MARC

MARC is a Transaction Server transaction processor that enables you to perform system operations and tasking functions. You initiate communications with MARC by opening the MARC window. Depending on the way your terminal is defined to the Transaction Server, the MARC window might appear automatically after you log on to the Transaction Server. If it does not, you might still be able to open the MARC window by entering the command *?ON MARC*. Your interactions with MARC between the time you open the MARC window and the time you log off or close the window are referred to as a *session*. MARC assigns each session an identifying number called the *session number*.

MARC Tasking Capabilities

MARC provides the only menu-assisted interface to tasking. You can use MARC menu selections or commands to submit WFL jobs or to initiate programs written in any language.

MARC offers commands and menu selections for initiating dependent processes, submitting WFL jobs, and initiating utilities. Once the process is initiated, MARC displays the TASK command in the Action field of the current screen. By transmitting this command, you can display a special screen called TASKSTATUS. You can use the TASKSTATUS screen to monitor and control the process.

Because the system administrator can modify MARC to add or delete functions, some features mentioned here might not be available at your site. The descriptions apply to the original version of MARC.

The following paragraphs provide an overview of MARC tasking capabilities. For further details about these features, refer to the *Menu-Assisted Resource Control (MARC) Operations Guide*.

Initiating Dependent Processes from MARC

You can enter RUN in the choice field of the MARC home menu to initiate a program as a dependent process. This selection can initiate a program written in any language except WFL. Entering this selection displays the RUN screen. You can use the RUN screen to specify the object code file title, any parameter that is to be passed, and any assignment to the TASKVALUE task attribute. You enter TASKVALUE assignments in the Value field of the screen.

An alternate method of initiating dependent processes is by using the RUN command. You can enter this command in the Action field of a screen or on the COMND screen. The syntax of this command is similar to the WFL *RUN* statement, except that the command can pass only a single parameter. Depending on the requirements of the program being initiated, the parameter can be a string of characters enclosed in quotation marks (") or a number with no quotation marks. The following are both valid examples:

```
RUN OBJECT/RECOMM("REPORT=DAILY")

RUN OBJECT/TELEMAX(346)
```

Initiating Compilations from MARC

You can initiate compilations from MARC in either of the following ways:

- By using the MARC *WFL* command to submit a WFL *COMPILE* statement. For details, refer to "Submitting WFL Jobs from MARC" later in this section.
- By using the EDIT screen to initiate an Editor session. While in the Editor, you can use the Editor COMPILE command to initiate a compilation.

Initiating Utilities from MARC

You can initiate utilities by using either the RUN screen or the RUN command. However, you can also use either of two special screens, UTIL or TOOLS, which list many utilities as selections. By choosing one of the selections on the UTIL or TOOLS screen, you cause the corresponding utility to be initiated. If parameters are needed, MARC prompts you to supply them.

Submitting WFL Jobs from MARC

You can use the START selection on the MARC screen to submit a WFL program that is stored in a disk file. Entering this selection displays the START screen. Use this screen to enter the file title of the WFL program and any parameter values to be passed to the program. You can also use this screen to enter a value for the STARTTIME task attribute of the WFL program.

WFL programs stored in disk files can also be initiated by way of the START command. The START command can pass parameters to the WFL program, but cannot include a STARTTIME specification.

You can use the MARC *WFL* command to submit WFL statements directly at the terminal. Simply type the word *WFL*, followed by the statements that constitute the WFL program. You can omit the *?BEGIN JOB* and *?END JOB* statements. The program cannot include any WFL constructs except data specifications or a STARTTIME specification. For example, the following WFL input initiates another program and assigns it a task attribute:

```
WFL RUN OBJECT/INVENTORY;FAMILY DISK = DPMAST OTHERWISE DISK
```

Monitoring Processes Initiated from MARC

When you initiate any dependent process, WFL job, or utility from a MARC session, the TASK command appears as a prompt on the current screen. Entering *TASK* in the Action field displays the TASKSTATUS screen. This screen displays information about the process and includes a field in which you can enter process control commands. You can leave the TASKSTATUS screen at any time by entering one of the screen traversal commands, such as HOME or GO, that are displayed. As long as the process is running, you can return to the TASKSTATUS screen by using the TASK command.

The TASKSTATUS screen includes fields that display various types of information for the process. The following are the fields and their meanings:

- The Task field displays the mix number and the name of the process.
- The Parameter field, if it appears, displays the value of the parameter passed to the process.
- The Task Status field displays the current stack state of the process. For a discussion of what the stack states mean, refer to Section 6, "Monitoring and Controlling Process Status."
- The Elapsed field displays the time elapsed since the process was initiated.
- The Processor field displays the processor time used by the process.
- The I/O field displays the accumulated I/O initiation time for the process.
- The area below the Elapsed, Processor, and I/O fields displays messages generated by the process, including "BOT", "EOT", DISPLAY, and RSVP messages.

You can enter process control commands in the Action field. The list of available actions below the Action field includes the most common system commands used for process monitoring and control. You can enter any of the listed commands without having to prefix them with the mix number of the process; MARC automatically prefixes the command with the mix number listed in the Task field. You can also enter system process control commands that are not listed as actions, but you must prefix them with the mix number of the process. For a list of system commands related to process monitoring and control, refer to "Tasking Command Equivalents" in this section.

If you submit a WFL job by way of the START screen or the START command, then the process control commands are displayed only during the compilation of the job. However, you can enter these commands even after they no longer appear as prompts, provided that you prefix them with a mix number. You can prefix them with the mix number of the job or of any task initiated by the job. The TASKSTATUS screen continues to display any messages generated by the job as it executes. You can initiate another process as soon as the job has finished compiling and has been inserted in a job queue.

However, if you submit a WFL job by way of the MARC *WFL* command, the process control commands continue to be displayed as the job executes. Also, it is not possible to initiate new processes until the job terminates.

If you initiate a process that initiates offspring, then any messages created for the offspring are included with the other process messages on the TASKSTATUS screen. You can enter process control commands for the offspring in the Action field, but you must always prefix the command with the mix number of the offspring process.

You can usually learn the mix number of the offspring by looking at its "BOT" message in the process messages display. However, if MARC has scrolled this message off the screen, you can learn the mix number by entering the VIEW command in the Action field. This command causes MARC to display the TASKVIEW screen, which lists the mix numbers and the names of the original process and all its descendants in a hierarchical order.

You cannot enter process control commands on the TASKVIEW screen. You can display the TASKSTATUS screen for a particular offspring by entering the mix number of the offspring in the Action field of the TASKVIEW screen. You can then enter process control commands on that TASKSTATUS screen. Alternatively, you can return from the TASKVIEW screen to the original TASKSTATUS screen by entering the RETURN command in the Action field.

Monitoring Other Processes in MARC

All system commands related to process monitoring and control can be entered through MARC, except for the primitive commands (commands preceded by two question marks). You can use these commands to monitor or control processes initiated from the current MARC session or processes initiated from other sources, such as CANDE or an ODT.

You can enter system commands on the COMND screen or in the Action field of any screen that displays "COmnd" as a prompt. However, system commands that you enter through MARC are screened for security. Many system commands are available only if the usercode of the session has privileged, SYSTEMUSER, or security administrator status. For details, refer to "Tasking Command Equivalents" in this section.

Each MARC session receives a unique number, also called the *session number*, which appears in the output from some system commands, including mix display commands. The session number is assigned from a range of numbers defined by the MAX (Maximums) system command. Depending on the range defined, the session number can be as low as 100 or as high as 65535.

The MARC session number does not appear as a process in mix display commands. However, the session number does appear in the output from two system commands: Y (Status Interrogate) and C (Completed Mix Entries). The output from these commands shows both the job number and the mix number of a process. If the process is a task, and it was initiated from a MARC session, then the job number shown is the MARC session number.

Communicating with Interactive Processes in MARC

A special window called a *task window* is created if a remote file is opened by a process run from a MARC session. In most cases, when the process opens the remote file, MARC automatically displays the task window. The current screen disappears and MARC displays the following message:

```
Enter ?MARC for task status
```

If the process writes to the remote file, the messages appear in the task window. If you type and transmit any text in the task window, MARC interprets this as input to the remote file. The only exceptions are the ?MARC command and other Transaction Server commands that are prefixed with question marks.

You can return to the TASKSTATUS screen by entering the ?MARC command. You can return to the task window by entering the TASK command in the Action field of any screen.

If you are on the task window when the process terminates, then MARC returns you to the originating screen. In some cases, MARC prompts you to press the SPCFY key before making this transfer. For information about why this happens, refer to "Communicating with MARC Terminals" later in this section.

Note that if you submit a WFL job through the START command and the job initiates a task that opens a remote file, you are not automatically transferred to the task window when the remote file is opened. When the task opens the remote file, a message of the following form appears on the TASKSTATUS screen:

```
<time> <mix number> Remote window <remote window name> OPEN.
  INTNAME = <internal name>. PROGRAM = <object code file title>.
```

Note the <remote window name> value in this message. You can transfer to the remote window by entering a command of the form:

```
?ON <remote window name>
```

You can return to the TASKSTATUS screen by entering the following command:

```
?ON MARC
```

The shorter form, ?MARC, is not accepted in this situation.

Access to Task Attributes in MARC

For each session, MARC stores information about a few selected task attributes. MARC requests some of this information from the user at log-on time and obtains the rest from usercode attributes defined in the USERDATAFILE. MARC assigns these task attribute values to any process initiated by that session (for example, by a MARC *RUN* command). The task attributes stored by MARC include the following:

BACKUPFAMILY	JOBNUMBER	PRINTDEFAULTS
CHARGE	JOBSUMMARY	SOURCESTATION
CONVENTION	JOBSUMMARYTITLE	STATION
DESTNAME	LANGUAGE	USERCODE
EXCEPTIONTASK	NOJOBSUMMARYIO	
FAMILY	PRIORITY	

A task initiated from a MARC session receives a JOBNUMBER value equal to the session number. The JOBNUMBER value for such a task can range anywhere from 100 to 65535, depending on the session number range established by the system command MAX SESSION = <number>.

For a WFL job started from a MARC session, MARC assigns a MIXNUMBER value from the mix number pool and assigns a JOBNUMBER value equal to the MIXNUMBER value. The JOBNUMBER value for such a task can range anywhere from 100 to 65535, depending on the mix number range established by the system command MAX MIX = <number>.

Certain of the session attributes established for MARC dialogue 1 are inherited by any sessions started in other MARC dialogues; these session attributes are USERCODE, ACCESSCODE, CHARGE, FAMILY, and LANGUAGE.

MARC provides commands and menu selections that you can use to set the values of the following attributes: DESTNAME, FAMILY, JOBSUMMARY, JOBSUMMARYTITLE, LANGUAGE, NOJOBSUMMARYIO, and PRINTDEFAULTS. The other attributes in the previous list cannot be accessed by the user.

You can also assign task attributes to specific processes by using task equations. You can enter task equations in MARC in either of the following ways:

- RUN, FILEEQUATE, and TASKATTR screens

  The RUN screen includes boxes you can fill to indicate that file equations or task attribute assignments are needed. If file equations are needed, the FILEEQUATE screen is displayed. You can enter any number of file equations. Implicitly, these are assignments to the FILECARDS task attribute. If task attribute assignments are needed, the TASKATTR screen is displayed. This screen includes fields for assigning selected task attributes, as well as an empty field you can use to assign one or more additional task attributes of your choice.

- RUN command

  When you initiate a task by using a RUN command, you can include task equations that assign task attribute values for the task. The following RUN command includes several task equations:

  ```
  RUN OBJECT/PROGA;TASKVALUE=1;DISPLAYONLYTOMCS=TRUE;FILE OUT=OUT/FILE;
  ```

MARC Programming Considerations

When you design a program to be run from MARC, you need to be aware of MARC features affecting parameter passing, task attribute access, and terminal communications.

Receiving Parameters from MARC

If you are designing a program to be initiated from MARC, be aware that the program can receive only one parameter from the RUN screen or RUN command that initiates it. If the user encloses the parameter in quotation marks ("), MARC passes the parameter as type Real Array with an unspecified lower bound. If the user does not enclose the parameter in quotation marks, MARC passes the parameter as type Real. For information about the parameter types in each language that are compatible with the Real and Real Array types, refer to Section 17, “Using Parameters.”

Access to Ancestral Processes in MARC

If you initiate a task through the MARC *RUN* command and that task accesses its own EXCEPTIONTASK task attribute, the system interprets EXCEPTIONTASK as a reference to the MARC library, *SYSTEM/MARC/COMMANDER. The task can use the EXCEPTIONTASK task attribute to query the values of the task attributes of the MARC MCS. However, if the task attempts to modify the task attributes of the MARC MCS, the task is terminated with a task attribute error. This error occurs because MARC runs with the private process option of the OPTION task attribute set to TRUE.

For tasks initiated through a MARC *RUN* command, the MYJOB task variable and the PARTNER task attribute act as synonyms for the MYSELF task variable. When such a task uses MYJOB or PARTNER to access any task attributes, the task attributes accessed are those of the task itself. However, if the task changes the values of the job summary-related task attributes, the changes affect the job summary of the MARC session. The job summary-related task attributes are JOBSUMMARY, JOBSUMMARYTITLE, and NOJOBSUMMARYIO.

For WFL statements submitted through a MARC *WFL* command, the MYJOB task variable refers to the WFL compiler process. The NAME of the WFL compiler process in this case is *MARC WFL*, prefixed by the usercode of the session. The MYSELF task variable refers to the task that is executing the compiled WFL statements. The NAME of this task is *WFLCODE*, prefixed by the usercode of the session. MYSELF(JOBNUMBER) returns the MARC session number, but MYJOB(MIXNUMBER) returns the mix number of the WFL compiler process.

When statements submitted through the WFL command use the MYJOB construct to alter job summary-related task attributes, these changes affect the job summary of the MARC session. However, if the MYSELF variable is used to access these task attributes, there is no effect on the job summary of the MARC session.

Note: *When you use the JOBSUMMARY command to display the current JOBSUMMARY value for the session, the output does not reflect any JOBSUMMARY assignments made by tasks of the session. Nevertheless, such assignments made by tasks do affect the job summary of the session unless overridden by a later JOBSUMMARY command.*

The MCSNAME task attribute of tasks initiated from MARC sessions typically returns a value of *SYSTEM/COMS*, which might or might not be preceded by an asterisk (*).

Communicating with MARC Terminals

The STATIONNAME task attribute is the preferred method of specifying the station where remote files should be opened. However, MARC does not assign the STATIONNAME attribute of programs or WFL jobs initiated from a MARC session. It is therefore advisable for you to explicitly assign a STATIONNAME value to any interactive processes originating from a MARC session.

For example, you can include the following statement in WFL jobs initiated from a MARC session:

```
MYSELF(STATIONNAME = #MYSELF(SOURCENAME));
```

This STATIONNAME value is inherited by any tasks of the WFL job, and enables them to open remote files successfully.

The STATION task attribute can be used for a similar purpose. However, the STATIONNAME task attribute is more reliable than STATION, because STATION stores a logical station number (LSN) that is subject to change.

The “Communicating with Interactive Processes” subsection pointed out that MARC opens a task window to enable a process to communicate with a user through a remote file. You can use the AUTOSWITCHTOMARC attribute to affect the handling of the task window for users. If you set the AUTOSWITCHTOMARC task attribute to TRUE, then users of the program are automatically transferred from the task window to the originating screen when the process terminates. If AUTOSWITCHTOMARC is FALSE, then the user must press the SPCFY key to return to the originating screen.

Operations Center

Operations Center is an application that runs under the Microsoft Windows operating system and is available on ClearPath MCP servers. This utility provides a graphical user interface (GUI) to system operations. If Operations Center is available on your workstation, initiate Operations Center by double-clicking the icon that appears in the Unisys Administration program group or folder.

Operations Center Tasking Capabilities

Operations Center enables you to submit jobs and to execute the supported task-related commands in a real-time environment. Through Operations Center, you can monitor and respond to all processes in the system mix. If you want to initiate and monitor a single program, you can do so more conveniently from CANDE or MARC.

Submitting WFL Jobs from Operations Center

From the Commands menu, you can submit jobs from Operations Center by using any of the following methods:

- Typing in an entire WFL program, including a BEGIN JOB; statement at the start, and then transmitting it. (There is no need to include an END JOB statement.)
- Entering one or more WFL statements preceded by the letters *CC.* (The BEGIN JOB is not necessary in this case.)
- Entering one of a certain group of WFL statements that do not require a BEGIN JOB or any other prefix when used in Operations Center. These statements include COMPILE, COPY, PROCESS, RERUN, RUN, and START.

When you submit WFL statements through the methods listed, the system usually executes the input as a WFL job. However, the system can execute some statements directly, without creating a WFL job. Such statements do not pass through the job queue mechanism and, therefore, are not affected by job queue attributes. For a list of these statements, refer to Section 4, "Tasking from Programming Languages."

For further details about submitting WFL programs from Operations Center, refer to the *Work Flow Language (WFL) Programming Reference Manual.*

Initiating Processes from Operations Center

You can use the PRIMITIVE RUN system command to initiate a program as an independent process. The program can be written in any language except WFL. The resulting process receives its own job file and job summary.

Note that if you enter *RUN* without specifying PRIMITIVE, the system treats this as the WFL *RUN* statement. The system creates a WFL job to execute the RUN statement and enters the job in a job queue. The job can be delayed by the queue mix limit or affected by other job queue attributes. Further, the job affects the job queue active count.

Therefore, you might prefer to use PRIMITIVE RUN to initiate processes, such as MCSs, that you do not wish to go through the job queue mechanism.

Initiating Compilations from Operations Center

You can initiate compilations from Operations Center by using the WFL *COMPILE* statement. The system responds to this command by creating a WFL job that includes the COMPILE statement and sending it through the job queue mechanism for initiation.

Monitoring and Controlling Processes from Operations Center

Operations Center provides the same commands as the ODT for monitoring and controlling processes. These system commands are used to monitor or control all the processes on the system, including processes initiated from any of the sources discussed in this section. These system commands are listed under "Tasking Command Equivalents" in this section. You can enter system commands by choosing the Commands Menu.

When you initiate Operations Center, there are nine standard views that can be opened from either the File Menu or the standard toolbar. These views correspond to the types of information you can view at an ODT in automatic display mode (ADM). You can modify the standard views or create new views. Standard views include:

View	Description
Active Entries	Currently running jobs and tasks in the mix
Waiting Entries	Tasks that need operator or user action because they are suspended on an RSVP condition
Scheduled Entries	Tasks that have not begun processing
Completed Entries	Jobs and tasks that have finished executing
Message Entries	System messages and programmatic display messages
Database Entries	Active database stacks
Library Entries	Frozen libraries
NT Task Management Entries	All active processes and their threads

From these views, you can perform tasks and access information. Select a view or an entry within a view. The commands on the Command toolbar and the Commands Menu that are valid for that view or entry are bold. Those commands that are not valid are dimmed. You can also display a Commands Menu by using the right mouse button to select an entry within a view.

To obtain more information than what is normally visible in a view, double-click an entry within a view. A Detailed Information dialog box appears that contains additional information about the entry and allows you to see fields such as Display and Reply Text that may be too short for a single line.

Access to Task Attributes from Operations Center

You can include task equations after a WFL task initiation statement submitted from Operations Center. In addition, if you type in a complete WFL job from Operations Center, you can include task attribute assignments in the job attribute list. However, you cannot use task equations after the PRIMITIVE RUN command.

When you initiate a process from Operations Center, the process typically does not inherit any of the task attributes that it would if you initiated the process from a MARC or CANDE session. For example, the USERCODE, ACCESSCODE, CHARGE, and FAMILY values of the process are usually null, unless explicitly assigned.

However, task attributes are inherited in the following cases:

- If you submit a WFL job that includes a USERCODE assignment in the job attribute list, then the following task attributes of the WFL job inherit values from the corresponding usercode attributes: CHARGE, CLASS, CONVENTION, FAMILY, LANGUAGE, PRINTDEFAULTS, and PRIORITY. This inheritance can be overridden by assignments to these attributes in the job attribute list.
- If you submit a job, the job can inherit attributes from a job queue. Refer to "Deciding on the Queue for a Job" in Section 4.

Note that programs initiated by a PRIMITIVE RUN command do not inherit the terminal usercode or any other usercode attributes.

Operations Center Programming Considerations

When you design a program to be run from Operations Center, you need to be aware of Operations Center features affecting parameter passing, task attribute access, and terminal communications.

Receiving Parameters from Operations Center

If you are designing a program to be initiated by the PRIMITIVE RUN command, be aware that the program cannot receive any parameters.

If the program is to be initiated by a WFL *RUN* statement entered from Operations Center, the program can receive the four parameter types passed by WFL: Boolean, integer, real, and string. The string parameter should be declared in the program as a real array (or compatible parameter type) with an unspecified lower bound. For information about real array parameters and compatible parameter types, refer to Section 17, "Using Parameters."

Access to Ancestral Processes in the Operations Center

For a process initiated by the PRIMITIVE RUN command, the MYJOB task variable and the EXCEPTIONTASK and PARTNER task attributes are all references to the process itself.

For a process initiated by a WFL *RUN* statement from Operations Center, MYJOB, EXCEPTIONTASK, and PARTNER are all references to the job that was created by the system to execute the RUN statement. The name of this job consists of the first 17 characters of the job input you submitted or the name after BEGIN JOB.

Communicating with Operations Center

Operations Center is not a terminal like an ODT. A program cannot use an ODT file as a remote file to display or read text in an Operations Center session. A WFL display statement is the only way to send text to Operations Center for display.

ODT

An operator display terminal (ODT) is any data comm terminal or workstation that is connected to the system through the System Control Processor (SCP) or the ODT-DLP. The system provides ODTs with access to two operational modes: *system command mode* and *data comm mode*. When an ODT is in data comm mode, you can log on to the Transaction Server and use various programs that run under the Transaction Server, such as MARC. When an ODT is in system command mode, you can enter system commands or view automatic displays of system information.

The following subsections discuss tasking capabilities and programming considerations for an ODT running in system command mode. For details about any of the system commands, refer to the *System Commands Operations Reference Manual*.

ODT Tasking Capabilities

The ODT provides you with the capability to submit WFL jobs and initiate dependent or independent processes. The ODT also enables you to conveniently monitor all the processes in the system mix.

Submitting WFL Jobs from an ODT

You can submit WFL programs at an ODT by using any of the following methods:

- Typing in an entire WFL program, including a BEGIN JOB statement at the start, and then transmitting it. (There is no need to include an END JOB statement.)
- Entering one or more WFL statements preceded either by a question mark (?) or by the letters *CC.* (The BEGIN JOB is not necessary in this case.)
- Entering one of a certain group of WFL statements that do not require a BEGIN JOB or any other prefix when used at the ODT. These include COMPILE, COPY, PROCESS, RERUN, RUN, and START.

When you submit WFL statements through the methods listed, the system usually executes the input as a WFL job. However, the system can execute some statements directly, without creating a WFL job. Such statements do not pass through the job queue mechanism, and therefore are not affected by job queue attributes. For a list of these statements, refer to Section 4, “Tasking from Programming Languages.”

For further details about submitting WFL programs from an ODT, refer to the *Work Flow Language (WFL) Programming Reference Manual.*

Initiating Processes from an ODT

You can use the ??RUN (Run Code File) primitive system command or the PRIMITIVE RUN system command to initiate a program as an independent process. The program can be written in any language except WFL. The resulting process receives its own job file and job summary.

Note that if you enter *RUN* without the two question marks, the system treats this as the WFL *RUN* statement. The system creates a WFL job to execute the RUN statement and enters the job in a job queue. The job can be delayed by the queue mix limit or affected by other job queue attributes. Further, the job affects the job queue active count. Therefore, you might prefer to use ??RUN or PRIMITIVE RUN to initiate processes, such as MCSs, that you do not wish to go through the job queue mechanism.

Initiating Compilations from an ODT

You can initiate compilations at an ODT by using the WFL *COMPILE* statement. The system responds to this command by creating a WFL job that includes the COMPILE statement and sending it through the job queue mechanism for initiation.

Initiating Utilities from an ODT

Utilities can be initiated at the ODT by way of the ??RUN or the PRIMITIVE RUN command or the WFL *RUN* statement. There are other system commands that initiate specific utilities, such as the TDIR (Tape Directory) command, which initiates the FILEDATA utility to list the directory of a tape, and the DA (Dump Analyzer) system command, which initiates the DUMPANALYZER utility.

Two WFL statements that initiate specific utilities can be entered at the ODT. The LOG statement initiates the LOGANALYZER utility, and the PB statement initiates the BACKUP utility. To use the WFL *PB* statement at the ODT, you must prefix it with a question mark (?); otherwise, the system interprets it as the PB (Print Backup) system command, which does not initiate the BACKUP utility.

Monitoring and Controlling Processes at an ODT

Of all the interactive sources for process initiation, the ODT provides the most complete selection of commands for monitoring and controlling processes. The operator can use these system commands to monitor or control all the processes on the system, including processes initiated from any of the sources discussed in this section. These system commands are listed under “Tasking Command Equivalents” in this section.

A unique feature of the ODT is Automatic Display mode. You initiate and control this mode by using the ADM (Automatic Display Mode) system command. You can use this feature to cause various types of information to be displayed at intervals, such as active entries, waiting entries, completed entries, and process messages. This feature allows you to monitor processes from beginning to end without having to enter commands repeatedly.

By default, Automatic Display mode displays seven lines of A (Active Mix Entries) system command output, three lines of W (Waiting Mix Entries) system command output, two lines of S (Scheduled Mix Entries) system command output, five lines of C (Completed Mix Entries) system command output, and devotes the remainder of the display to MSG (Display Messages) system command output. By default, the system updates the contents of the display every four seconds. You can use the ADM command to cause different system commands to be displayed or to change the time interval for updates to the display.

Access to Task Attributes from an ODT

You can include task equations after a WFL task initiation statement submitted from the ODT. Also, if you type in a complete WFL job at the ODT, you can include task attribute assignments in the job attribute list. However, you cannot include task equations after the ??RUN or the PRIMITIVE RUN command.

When you initiate a process from the ODT, the process typically does not inherit any of the task attributes that it would if you initiated the process from a MARC or CANDE session. For example, the USERCODE, ACCESSCODE, CHARGE, and FAMILY values of the process are usually null, unless explicitly assigned.

However, task attributes are inherited in the following cases:

- If you submit a WFL job that includes a USERCODE assignment in the job attribute list, then the following task attributes of the WFL job inherit values from the corresponding usercode attributes: CHARGE, CLASS, CONVENTION, FAMILY, LANGUAGE, PRINTDEFAULTS, and PRIORITY. This inheritance can be overridden by assignments to these attributes in the job attribute list.
- If you use the TERM (Terminal) system command to assign a terminal usercode to an ODT. This usercode is inherited by WFL jobs submitted from the ODT, unless overridden by a USERCODE assignment in the job attribute list. The job also inherits values for the same set of task attributes listed in the previous item in this list.
- If you submit a WFL job, the job can inherit attributes from a job queue. Refer to "Deciding on the Queue for a Job" in Section 4.

Note that programs initiated by a ??RUN command or a PRIMITIVE RUN command do not inherit the terminal usercode or any other usercode attributes.

Special types of security status apply to nonusercoded processes and certain WFL statements when they are entered at the ODT. These privileges are discussed in Section 5, "Establishing Process Identity and Privileges."

ODT Programming Considerations

When you design a program to be run from the ODT, you need to be aware of ODT features affecting parameter passing, task attribute access, and terminal communications.

Receiving Parameters from an ODT

If you are designing a program to be initiated by the ??RUN primitive system command or the PRIMITIVE RUN system command, be aware that the program cannot receive any parameters.

If the program is to be initiated by a WFL *RUN* statement entered at an ODT, the program can receive the four parameter types passed by WFL: Boolean, integer, real, and string. The string parameter should be declared in the program as a real array (or compatible parameter type) with an unspecified lower bound. For information about real array parameters and compatible parameter types, refer to Section 17, "Using Parameters."

Access to Ancestral Processes in the ODT Environment

For a process initiated by the ??RUN primitive system command or the PRIMITIVE RUN system command, the MYJOB task variable and the EXCEPTIONTASK and PARTNER task attributes are all references to the process itself.

For a process initiated by a WFL *RUN* statement at an ODT, MYJOB, EXCEPTIONTASK, and PARTNER are all references to the WFL job that was created by the system to execute the RUN statement. The name of this WFL job consists of the first 17 characters of the WFL input you submitted.

Communicating with an ODT

Interactive programs that are designed for use at remote terminals might not run successfully if initiated from the ODT. You must design the program somewhat differently if it is to be initiated at an ODT. If the process opens a file with KIND = REMOTE, it is discontinued with an "UNKNOWN FILE/STATION" error. The process should open a file with KIND = ODT instead. A process can determine whether it was initiated from an ODT or a remote terminal by interrogating the SOURCEKIND task attribute.

A process can open a file either at a labeled ODT or at a scratch ODT. A labeled ODT is one that has been assigned a label by the LABEL (Label ODT) system command. A scratch ODT is one that has not been assigned such a label.

To open a file at a labeled ODT, a process should first set the TITLE file attribute to match the label assigned to the ODT. In addition, the NEWFILE file attribute value should be FALSE or else unspecified. If NEWFILE is unspecified, the MYUSE file attribute value should be IN or IO. When the process runs, the system opens the remote file at any ODT with a matching label. If none of the ODTs has a matching label, the process is suspended with a "NO FILE <file title> (SC)" RSVP message. The process resumes execution when an operator uses the LABEL command to label an ODT with the requested file title.

To open a file at a scratch ODT, a process should set the NEWFILE file attribute to TRUE, or leave NEWFILE unspecified and set MYUSE to OUT. The value of the TITLE file attribute makes no difference in this case. If the process was initiated from an ODT, and that ODT is a scratch ODT, the system opens the file at that ODT. Otherwise, the system selects another scratch ODT and opens the file there.

To open a file at a particular ODT, regardless of whether that ODT is labeled or scratch, the process can assign the UNITNO file attribute a value equal to the physical unit number of the ODT. The system opens the file at the requested ODT even if the ODT is labeled and the label does not match the TITLE file attribute. However, note that use of the UNITNO file attribute is restricted on systems running the Security Services for ClearPath MCP at the S2 level; refer to the *Security Administration Guide* for details.

To open a file at the ODT where the process was initiated, regardless of whether that ODT is labeled or scratch, the process should first read the physical unit number from its own SOURCESTATION or ORGUNIT task attribute value. The process can then assign the physical unit number to the UNITNO file attribute, as described previously.

When a process opens an ODT file, automatic display mode at the ODT is temporarily suspended. However, system commands continue to be available. You can enter text into the ODT file by preceding the text with a GS character. The GS character is also known as the *delta* character and looks like an upward-pointing triangle. (Do not confuse the GS character with the circumflex character, which resembles an inverted letter V.) Refer to the documentation for your terminal to find out whether your terminal supports the GS character, and which key it is mapped to.

You can indicate that there is no more input, and cause an end-of-file condition, by entering the GS character, followed by ?END.

When the process closes the ODT file, the system removes the label from the ODT and resumes Automatic Display mode. You can also resume Automatic Display mode while the ODT file is still open by entering an ADM OK command at the ODT.

An example of a program that uses an ODT file is given in the ORGUNIT description in the *Task Attributes Programming Reference Manual*.

Tasking Command Equivalents

MARC and the ODT allow you to enter almost all of the same system commands for process initiation, monitoring, and control. In addition, CANDE allows you to enter process control commands that correspond closely to system commands.

The system commands available in MARC for process control are spelled the same as those available at an ODT, and have the same functionality, with the following exceptions:

- Security

  If the Transaction Server security category COMMANDCAPABLE is defined, then system commands can be submitted in MARC only by users defined as COMMANDCAPABLE. Further, some commands are available only to users with SYSTEMUSER or privileged status. Some other commands are filtered: in other words, they are limited to monitoring and controlling processes with the same usercode as the MARC session. For further information about COMMANDCAPABLE, SYSTEMUSER, and privileged status, refer to the *Security Administration Guide*.

- Spelling

  The MSG (Display Messages) system command is spelled SMSG in MARC.

CANDE process control commands differ from the corresponding system commands in the following ways:

- Spelling

  The CANDE process control commands each begin with a single question mark (?). In addition, the following spelling differences exist:

  – ?JA corresponds to the J (Job and Task Structure) system command.

  – ?CS corresponds to the mix number system command, which is formally known as the COMPILE STATUS (Information for Compiler Task) command. Note that the ?CS command in CANDE is not related to the CS (Change Supervisor) system command.

  – ?MXA corresponds to the MX (Mix Entries) system command. ?MXA can be abbreviated as ?MX or ?M.

- Implicit mix numbers

  For commands that apply to a dependent process initiated directly from the CANDE session, you can omit the mix number from the command. For example, instead of entering *?1234 Y*, you can enter simply *?Y*.

- Security

  In general, the CANDE process control commands can monitor or control only processes running with the same usercode as the CANDE session. If you attempt to apply a CANDE process control command to a process running with a different usercode, CANDE displays the message “INVALID NUMBER.” However, CANDE makes one exception to this restriction. If you initiate a process in a CANDE session, and that process later changes its own usercode, CANDE still enables you to apply process control commands to that process.

- Mix display options

  The CANDE mix display commands (?C, ?JA, ?LIBS, and ?MXA) do not provide the following options of the equivalent system commands: ALL, MCSNAME, NAME, QUEUE, and USER. However, the ALL option is implicitly set for all CANDE mix display commands. Furthermore, CANDE mix display commands do offer one feature that the corresponding system commands do not: the ability to specify a logical station number (LSN), which limits the display to processes originating from the specified station.

Table 3–1 shows the equivalent commands in these three interfaces and briefly states the function of each command. In Table 3–1, the abbreviations (f), (pu), and (su) are used in the MARC column to indicate commands that are filtered or that require SYSTEMUSER status or privileged status. For complete descriptions of these commands, refer to the *System Commands Operations Reference Manual*, the *Menu-Assisted Resource Control (MARC) Operations Guide*, and the *CANDE Operations Reference Manual*. For a general introduction to process monitoring and control from an ODT, refer to the *System Operations Guide*.

Table 3–1. Interactive Tasking Functions

Functional Area	ODT or Operations Center	MARC	CANDE	Specific Function
Initiating Processes	PRIMITIVE RUN	None	None	Initiate an object code file as an independent process.
	??RUN (ODT only)	None	None	Initiate an object code file as an independent process.
	RUN	RUN	RUN, UTILITY	Initiate an object code file as a dependent process.
	<WFL statements>	WFL	WFL	Submit WFL statements.
	START	START	START	Submit a WFL program stored in a file.
Managing Queued WFL Jobs	<mixno>DS	?<mixno> DS (f)	?<mixno> DS	Discontinue a queued WFL job.
	FS	FS (su)	None	Force initiation of a queued WFL job.
	MOVE	MOVE (su)	None	Change order of queued WFL jobs.

Table 3–1. Interactive Tasking Functions

Functional Area	ODT or Operations Center	MARC	CANDE	Specific Function
Managing Queued WFL Jobs (cont.)	PF	PF (su)	None	Display FETCH message associated with a WFL job.
	PQ	PQ (su)	None	Discontinue all the WFL jobs in a queue.
	PR	PR (su)	None	Change the priority of a queued WFL job.
	SQ	SQ (f)	?SQ	Display the WFL jobs in a queue.
	STARTTIME	STARTTIME (f)	?<mixno> STARTTIME	Assign a start time to a queued WFL job.
	Y	Y (f)	?<mixno> Y	Display information about a queued WFL job.
Monitoring the Mix	ADM (ODT only)	None	None	Periodically display system mix and other items.
	C	COMND C (f)	?C	Display completed entries.
	DBS	DBS (su)	None	Display database stacks.
	J	J (f)	?JA	Display active mix entries, grouped into process families.
	LIBS	LIBS (f)	?LIBS	Display library processes.
	MSG	SMSG (f)	?MSG	Display process messages.
	MX	MX (f)	?MXA	Display active, scheduled, and waiting mix entries.
	S	S (su)	?S	Display scheduled mix entries.
	W	W (f)	?W	Display waiting mix entries.

Table 3–1. Interactive Tasking Functions

Functional Area	ODT or Operations Center	MARC	CANDE	Specific Function
Displaying Process Status	Y	Y (f)	?Y	Display current status of a process.
	<mixno>	<mixno>	?<mixno> or ?CS	Display the status of a compilation.
	OT	OT (f)	?OT	Display contents of a selected word in the process stack.
Displaying Process Resource Usage	CU	CU (f)	?CU	Display current memory usage of a process.
	TI	TI (f)	?TI	Display accumulated processor, I/O, presence bit, ready queue, and elapsed times for a process.
Communicating with a Process	HI	HI (f)	?HI	Cause process EXCEPTIONEVENT and optionally assign a TASKVALUE.
	AX	AX (f)	?AX	Pass a string of text to a process.
	IB	IB (su)	None	Display instruction block associated with a WFL job.
	PF	PF (su)	None	Print a FETCH message associated with a WFL job.
	SW	SW (f)	?SW	Modify the SW1 through SW8 task attributes of a process.
Modifying an Active Process	DS	DS (f)	?DS	Abnormally terminate execution of a process.
	DUMP (DP on Operations Center toolbar)	DUMP	?DUMP	Initiate a program dump.
	FS	FS (su)	None	Force initiation of a scheduled process.

Table 3–1. Interactive Tasking Functions

Functional Area	ODT or Operations Center	MARC	CANDE	Specific Function
Modifying an Active Process (cont.)	LP	LP (f)	None	Protect a process from DS or QT commands.
	PR	PR (su)	None	Change the priority of a process.
	ST	ST (f)	?ST	Suspend execution of a process.
Responding to Suspended Processes	AX	AX (f)	?AX	Pass a string of text to a process.
	BADFILE (BF on Operations Center toolbar)	BADFILE (f)	None	Continue to copy a file from tape to disk when tape errors occur.
	DS	DS (f)	?DS	Discontinue a process.
	FA	FA (f)	?FA	Modify file attributes used by a process.
	FM	FM (su)	None	Change the printer form used by a process.
	FR	FR (f)	?FR	Specify that a tape reel is the last of a multireel set.
	IL	IL (su)	None	Change the physical unit used for an input file.
	NF	NF (f)	?NF	Return an open error to a process opening a file that is not an optional file.
	NOTOK	NOTOK (f)	?NOTOK	Prevent the process from attempting a given action, but do not discontinue the process.
	OF	OF (f)	?OF	Indicate an optional file is not present.
	OK	OK (f)	?OK	Cause a suspended process to attempt to resume processing.

Table 3–1. Interactive Tasking Functions

Functional Area	ODT or Operations Center	MARC	CANDE	Specific Function
Responding to Suspended Processes (cont.)	OU	OU (su)	None	Change the physical unit used for an output file.
	RM	RM (f)	?RM	Remove a file specified in a DUP LIBRARY message.
	UL	UL (su)	None	Assign an unlabeled tape file to a particular process.
Saving and Restarting Processes	BR	BR (su)	None	Display checkpoint eligibility or initiate a checkpoint.
	OK	OK (f)	?OK	Allow automatic restart of a process.
	DS	DS (f)	?DS	Deny automatic restart of a process.
	RERUN	WFL RERUN	WFL RERUN	Initiate manual restart of a process.
	RESTART	RESTART	None	Terminate and restart a WFL job.

Note: *The <mixno> syntax is formally known as the COMPILE STATUS (Information for Compiler Task) system command.*

The following abbreviations are used in Table 3–1:

- *f* *(Filtered if not SYSTEMUSER or privileged.)*
- *su* *(SYSTEMUSER or privileged status required.)*
- *mixno* *(Mix number)*

Communicating with an Operator

You can design a process to display information to an operator or accept information from an operator. You can accomplish this communication through any of the following methods:

- By accepting parameters from the operator in the statement that initiates the process. This topic is discussed earlier in this section under “Receiving Parameters from CANDE,” “Receiving Parameters from MARC,” and “Receiving Parameters from an ODT.”
- By performing read and write operations on a remote file or ODT file. This topic is discussed in the following subsections of this section: “Communicating with CANDE Terminals,” “Communicating with MARC Terminals,” and “Communicating with an ODT.”
- By using certain statements and task attributes that the system provides for operator communications. These methods are discussed in the following subsections.

Displaying Information to Operators

A process can display information to operators using any of the following features: DISPLAY statements, instruction blocks, and fetch specifications.

DISPLAY statements are the most commonly used of these methods. The DISPLAY statement is implemented in ALGOL, COBOL, and WFL. This feature is also available as the Display procedure in Pascal. The following is a WFL example of this statement:

```
DISPLAY "INCORPORATING NEW DATA - MAY TAKE AWHILE";
```

The output from a DISPLAY statement is referred to as a *DISPLAY message*. The DISPLAY message appears as one of the entries in the response to the MSG (Display Messages) system command. If the process is initiated from a CANDE or MARC session, the DISPLAY message is automatically displayed at the session. The programmer can use the DISPLAYONLYTOMCS task attribute to limit the display of the message to the originating session. If this task attribute is TRUE, then the DISPLAY message does not appear at the ODT.

You can use instruction blocks to store information that an operator can display at any time. By contrast, DISPLAY messages are only temporarily visible to the operator, because the MSG command displays only the most recent system messages. Instruction blocks are created using the INSTRUCTION statement, which is available only in WFL. The following is an example of this statement.

```
INSTRUCTION 3 TESTTAPE IS IN TAPE RACK 3.;
```

An operator can use the IB (Instruction Block) system command to display instruction blocks for a WFL job. For example, a command of the form *7645 IB* displays the most recent instruction block for the WFL job with mix number 7645. A command of the form *7645 IB 3* displays instruction block 3 for that WFL job.

The disadvantage of instruction blocks is that nothing prompts the operator to use the IB command. The operator has to know in advance that instruction blocks exist for a particular WFL job. If you want to be sure that an operator sees a message, you can use the FETCH task attribute. This task attribute can be used only in WFL jobs, and only in the job attribute list at the start of the job. You can assign any arbitrary string of text to this attribute. The following is an example of a FETCH assignment:

```
FETCH = "THIS JOB NEEDS THREE TAPE DRIVES";
```

If the operating system option NOFETCH is not set, then when a WFL job containing a FETCH assignment reaches the head of a job queue, the system suspends the job rather than initiating it. The job appears in the W (Waiting Mix Entries) system command display with an RSVP message of REQUIRES FETCH. The operator can use the PF (Print Fetch) system command to display the FETCH specification, and the OK (Reactivate) system command to cause the job to be initiated.

If NOFETCH is set, then the system does not suspend jobs with FETCH specifications. However, the PF system command can still be used to display FETCH specifications.

If you enter a PF command for a process that has no FETCH specification, the system displays the message "NO FETCH STATEMENT."

Accepting Information from Operators

A process can be passed information by an operator using the HI (Cause EXCEPTIONEVENT), AX (Accept), or SW (Switches) system command.

Accepting Input from HI Commands

EXCEPTIONEVENT is an event-valued task attribute, meaning that it has either of two states: HAPPENED or NOT HAPPENED. The HI command *causes* the EXCEPTIONEVENT, meaning that the value is changed to HAPPENED. This action has no effect on process execution unless the program is specifically designed to monitor the status of the EXCEPTIONEVENT. Only programs written in WFL, ALGOL, or COBOL have access to this attribute.

A program can monitor the EXCEPTIONEVENT in any of the following ways:

- To suspend execution until the EXCEPTIONEVENT is caused, the process can use a simple wait statement such as *WAIT(MYSELF.EXCEPTIONEVENT)* in ALGOL or *WAIT;* in WFL.
- To suspend execution until either the EXCEPTIONEVENT or some other event occurs, the process can use a complex wait statement that lists the EXCEPTIONEVENT as one of several events.
- To continue doing other work until the EXCEPTIONEVENT is caused, the process can attach an interrupt to the EXCEPTIONEVENT.

In addition to causing the EXCEPTIONEVENT, the HI command can also pass an assignment to the TASKVALUE task attribute of the process. For example, the command *3874 HI 14* causes the EXCEPTIONEVENT of the process with mix number 3874 and assigns a TASKVALUE of 14. To design a process to use this type of input, you must first use a wait statement or interrupt to monitor the EXCEPTIONEVENT. Whenever the EXCEPTIONEVENT occurs, the process can read its own TASKVALUE and take appropriate action.

Accepting Input from AX Commands

Because the programmer controls the way an application responds to a HI command, the operator has no direct way of discovering whether a HI command is needed or what effect it has. Another feature is available that allows the process itself to prompt the operator for certain types of input. This feature is the ACCEPT statement.

The ACCEPT statement displays a string of text to the operator and suspends execution of the process. The process appears in the W (Waiting Mix Entries) system command display, where it can attract the attention of an operator. Execution resumes when the operator uses an AX (Accept) system command to pass another string of text to the process.

In some situations, you might find it more convenient for a process to continue executing until an AX string is available from the operator. This goal can be achieved in any of the following ways:

- If the operator is familiar with the program, and knows that the program requires an AX string, he or she can enter the AX string without waiting for the process to become suspended. The operator can enter the AX string either by appending an AX task equation to the RUN statement or by entering an AX system command while the process is running. The system saves the AX string that was input by the operator. When the process executes an ACCEPT statement, the process retrieves the AX string and immediately continues executing.

  If an operator submits more than one AX string for a process before the process performs its next ACCEPT statement, then the system must either queue the extra AX strings or discard them. You can use the QUEUEDAX option of the SYSOPS (System Option) system command to enable or disable queuing of AX strings. If QUEUEDAX is set, then the system queues up to 250 AX strings for a process. If QUEUEDAX is reset, then each AX string overwrites any pending AX string for a process.

  QUEUEDAX is set TRUE by default on ClearPath systems.

- By using a conditional ACCEPT statement. This form of ACCEPT checks for AX text previously submitted by the operator. The conditional ACCEPT returns a Boolean value indicating whether such text was found. The process continues executing normally, regardless of whether an AX text was available.

- By using the ACCEPTEVENT task attribute. The system causes the ACCEPTEVENT of a process whenever the operator enters an AX command for that process. A process can monitor the ACCEPTEVENT using wait statements or interrupts, similar to those used for monitoring the EXCEPTIONEVENT. Whenever the ACCEPTEVENT is caused, the process can execute an ACCEPT statement to capture the AX input.

Note: *Programs should not use interrupts to detect AX strings passed to the program through the AX task attribute. AX task attribute assignments do not cause the ACCEPTEVENT task attribute. For further details, refer to the description of the AX task attribute in the* Task Attributes Programming Reference Manual*.*

Accepting Input from SW Commands

Each process has associated with it eight Boolean task attributes named SW1 through SW8. These task attributes have no intrinsic meaning, but serve to convey application-defined information to a process. Operators can modify the values of these attributes for a running process by using the SW (Switches) system command. For example, the following commands set and reset the SW1 attribute of the process with mix number 4875:

```
4875 SW1 TRUE

4875 SW1 FALSE
```

There is no event, equivalent to EXCEPTIONEVENT or ACCEPTEVENT, to inform the process that one of the switch values has changed. Instead, to detect that one of the switch task attributes has changed value, the process must periodically interrogate the value of that switch.

Alternately, you can establish a convention whereby the operator is expected to follow each SW command with a HI or AX command to notify the process to interrogate the switch values. For example, the following ALGOL statements cause a program to interrogate SW1 and SW2 whenever an AX command is received:

```
INTERRUPT SWITCHER;
BEGIN
  IF MYSELF.SW1 THEN
     <statements>
  ELSE IF MYSELF.SW2 THEN
     <statements>
END;
ATTACH SWITCHER TO MYSELF.ACCEPTEVENT;
```

The operator can also use the SW command to display the current values of the SW1 through SW8 task attributes of a process. The following is an example of a command that interrogates these attributes for mix number 4873, and the resulting display:

```
4873 SW

    SWITCH VALUES FOR 4873:
         SW 1 = FALSE
         SW 2 = FALSE
         SW 3 = FALSE
         SW 4 = TRUE
         SW 5 = FALSE
         SW 6 = FALSE
         SW 7 = FALSE
         SW 8 = TRUE
```

Section 4
Tasking from Programming Languages

The implementations of several programming languages include extensions for process initiation and control. You can use these features to

- Initiate related suites of programs, so there is no need for an operator to initiate them individually
- Divide an application into two or more cooperating, parallel processes for faster execution

Among the languages with advanced process initiation and control capabilities are WFL, ALGOL, and COBOL. Of these, WFL is the simplest to use, and has the advantage of passing through the job queue mechanism and offering automatic job restart after a halt/load. On the other hand, ALGOL and COBOL offer sophisticated features such as user-declared events, interrupts, port files, and a large variety of parameter types. Each of these languages provides access to task attributes.

This section describes the tasking capabilities of WFL, ALGOL, and COBOL in some detail and provides brief examples of tasking programs written in each of these languages. Additionally, this section includes a brief overview of the tasking capabilities of other languages.

Work Flow Language (WFL)

Work Flow Language (WFL) is a programming language that is designed specifically for use in task initiation and control. WFL is a block-structured language with syntax similar to ALGOL, although WFL is simpler and easier to learn.

The following subsections explain how WFL jobs are submitted and how they can be used to initiate other processes.

For further information about WFL, refer to the *Work Flow Language (WFL) Programming Reference Manual*.

Submitting WFL Input

WFL statements can be stored in disk files or in arrays in programs written in other languages. You can also enter and transmit WFL statements at a terminal. Regardless of how WFL statements are stored or submitted, a group of one or more WFL statements is referred to as *WFL input*.

WFL input must be submitted with special-purpose statements such as START and ZIP. You cannot use general-purpose initiation statements such as CALL, PROCESS, and RUN to initiate a WFL job.

The system can compile WFL input and execute it as a job or a task, or it can skip the compilation and simply interpret the WFL input. The statement you use to submit the WFL input and the statements contained in the WFL input together determine how the system executes that input.

Table 4–1 summarizes the factors that determine how the system executes WFL input. The various sources that can submit WFL input are listed at the left. The headings of the two right hand columns give information about the contents of the WFL input. The following are the meanings of these headings:

- The Single Interpretive Statement column indicates WFL input consisting of a single statement that is one of the following statements: ALTER, CHANGE, PRINT, REMOVE, RERUN, SECURITY, or START. The WFL input can also include a FAMILY job attribute assignment, but cannot include any other job attributes. For example, the following input is treated as a single interpretive statement:

  ```
  FAMILY DISK = SYSPK ONLY;CHANGE (JASMITH)ORDS TO (JASMITH)OLDORDS;
  ```

- The Other Statements column indicates WFL input that consists of either more than one statement or a single statement that is not one of the interpretive statements. A WFL input also falls into this category if it includes assignments to job attributes other than the FAMILY attribute. For example, the following input would fall into this category:

  ```
  JOBSUMMARY = SUPPRESSED;CHANGE (JASMITH)ORDS TO (JASMITH)OLDORDS;
  ```

Table 4–1. WFL Execution Modes

Sources for Submitting WFL Input	Single Interpretive Statement	Other Statements
CALL SYSTEM WFL (COBOL74, COBOL85)	Interpreted	Job
CONTROLCARD function (DCALGOL)		
With [38:01] =1 and [07:08] = 4 (Array Input):	Interpreted	Task
Otherwise:	Interpreted	Job
START Statement (CANDE, MARC, or WFL)	Job	Job
WFL Command (CANDE or MARC)	Interpreted	Task
WFL Statements Entered at the ODT	Interpreted (except PRINT, which is executed as a job)	Job
ZIP statement (ALGOL or RPG)		
With Array:	Interpreted	Job
With File:	Job	Job

If the system executes the WFL input as a job, it first calls an independent runner called CONTROLCARD to compile the job and create a job file. CONTROLCARD invokes the WFL compiler, which is a procedure exported by the system library WFLSUPPORT. CONTROLCARD runs in a special high-priority category that prevents it from being scheduled or suspended by the system if there is a shortage of available memory. The job file that CONTROLCARD creates contains more information than a typical job file, as discussed in Section 2, "Understanding Interprocess Relationships."

The system then inserts the job file in a job queue. (For a description of the job queue mechanism, refer to "Selecting the Queue for a Job" later in this section.) Later, the system selects the job file from the job queue and initiates it as a job (an independent process). When the job terminates, the system usually prints the job summary and any backup files associated with the job and its tasks. The system then deletes the job file.

If the system executes the WFL input as a task, the system initiates CONTROLCARD to compile the input and create an object code file. The system then initiates the WFL input as a task (a dependent process); the task does not pass through the job queue mechanism. When the task terminates, the system removes the object code file.

By default, no job summary or backup files are associated with the WFL task. For example, if the WFL task was initiated from a CANDE session, then backup files produced by the WFL task or its descendants are associated with the CANDE session and queued for printing only when the session is ended.

If the system handles the WFL input interpretively, then CONTROLCARD executes the WFL statement without creating a WFL job or a WFL task. In this case, CONTROLCARD creates neither a job file nor an object code file, nor does it use the job queue mechanism.

Selecting the Queue for a Job

A *job queue* is a list of WFL jobs that are awaiting initiation. Job queues are defined by the system administrator and managed by the operating system.

The purpose of job queue definitions is to allow the system administrator to set up some general parameters affecting the flow of WFL jobs on the system. Because a WFL job is typically an agent for initiating batch programs, the job queue system by implication can be used to regulate the initiation of batch programs in general.

Before defining the job queues, the system administrator usually analyzes the batch programs run on the system in terms of their patterns of resource usage and their relative urgency. The administrator then defines a separate job queue for each set of batch programs that show similar characteristics. For example, if there is a payroll application that has to finish processing before a precise deadline, the administrator might assign the application to a high-priority job queue. The administrator uses an MQ (Make or Modify Queue) system command to define the job queue.

For a complete explanation of job queues and using job queues in system administration, refer to the *System Administration Guide*. For information about using system commands to monitor and interact with jobs in queues, refer to the *System Operations Guide*. The following subsections describe the features of job queues that are of most direct interest to a programmer.

Deciding on the Queue for a Job

Depending on the policies that are in effect at your site, you might be required to ask your system administrator to which job queue to submit a particular WFL job. However, if the system administrator allows you to decide on the job queue, then you need to examine the job queue definitions to determine which queue is most suitable to your job.

The system command for displaying job queue definitions is QF (Queue Factors). The following is an example of a QF command and the response:

```
QF 4

     QUEUE 4:
       MIXLIMIT = 2
       DEFAULTS:
         PRIORITY = 50
         PROCESSTIME = 100
       LIMITS:
         PRIORITY = 60
         PROCESSTIME = 200
```

In this example, 4 is the job queue number. This number uniquely identifies a job queue. If the QF command does not specify a number, the output displays the definitions of all job queues on the system. The MIXLIMIT value specifies, roughly, the maximum number of jobs and descendant tasks initiated through this job queue that can be running concurrently. If the actual number of jobs and tasks originating from this job queue equals or exceeds the MIXLIMIT value, the system temporarily ceases initiating jobs from this job queue. After one or more of the jobs and tasks in this job queue terminates, the system resumes initiating jobs from this job queue.

The DEFAULTS and LIMITS portions of the job queue definition specify default values and maximum values for various task attributes that restrict the resource usage of a process.

The job queue defaults are inherited by the corresponding task attributes of a WFL job. However, the job can override this inheritance with assignments in the job header; that is, assignments that follow the BEGIN JOB construct but precede any of the declarations and statements in the job. Consider the following example:

```
?BEGIN JOB;
  CLASS = 4;
  PRIORITY = 55;
  TASK T;
  MYSELF(MAXPROCTIME = 150);
  RUN OBJECT/PROG;
?END JOB
```

Assume that this job is submitted through the job queue that was previously shown in the QF command example. Queue 4 has default values for both PRIORITY and PROCESSTIME (which corresponds to the MAXPROCTIME task attribute). The PRIORITY assignment in the job is part of the job header, and therefore overrides the PRIORITY queue default. However, the MAXPROCTIME assignment in the job is not part of the job header. Therefore, the job does inherit the default MAXPROCTIME of 100 at initiation. The statement that assigns MAXPROCTIME a value of 150 has no effect, because the system does not allow a process to increase its MAXPROCTIME value after initiation.

Now consider the following job:

```
?BEGIN JOB;
  CLASS = 4;
  PRIORITY = 75;
  MAXPROCTIME = 300;
  TASK T;
  RUN OBJECT/PROG;
?END JOB
```

The system would never accept this job into queue 4, because the job header assigns values to PRIORITY and MAXPROCTIME that are both higher than the queue limits for these attributes. Since the CLASS attribute explicitly requests queue 4, the system rejects the job and displays a “Q-DS” message. (The CLASS attribute is explained under “Requesting the Queue for a Job,” later in this section.)

The following are the job queue attributes that establish resource usage limits, and the task attributes that correspond to the job queue attributes....

Job Queue Attribute	**Task Attribute**	**Effect**
ELAPSEDLIMIT	ELAPSEDLIMIT	Limits the amount of time a job can be in use
IOTIME	MAXIOTIME	Limits the amount of processor time that can be devoted to initiating I/O operations for the job and its tasks
LINES	MAXLINES	Limits the number of lines the job and its tasks can print
PROCESSTIME	MAXPROCTIME	Limits the amount of processor time that a process can use for computations
PRIORITY	PRIORITY	Specifies the relative urgency of jobs and tasks as compared to other processes in the mix
SAVEMEMORYLIMIT	SAVEMEMORYLIMIT	Limits the amount of save memory the job and its tasks can use
TEMPFILELIMIT	TEMPFILELIMIT	Limits the space the job and its tasks can allocate for temporary disk files
WAITLIMIT	WAITLIMIT	Limits the amount of time the job and its tasks can remain waiting after executing a WAIT statement

If the actual resource usage of the job or its tasks exceeds one or more of the resource usage limits, the system discontinues the process that exceeded the limit. The point of this behavior is to encourage you to reexamine the job queue definitions and submit the job through the appropriate job queue.

In summary, you can determine an appropriate job queue for a job by estimating the resource usage requirements of the job and choosing a job queue whose resource usage limits are adequately high. There are, however, some additional restrictions:

- The system administrator can assign two attributes to your usercode that specify which job queues you are allowed to use. These attributes are CLASSLIST and ANYOTHERCLASSOK. If ANYOTHERCLASSOK is set, then CLASSLIST is interpreted as a list of the job queues you are forbidden to use. If ANYOTHERCLASSOK is not set, then CLASSLIST is interpreted as a list of all the job queues you are allowed to use. You should ask the system administrator whether these attributes are defined for your usercode.
- The system administrator can use the UQ (Unit Queue) system command to specify that all WFL jobs submitted from a particular ODT be routed into a particular job queue. The inquiry form of the UQ command can be used to display the unit queue assignments in effect on the system.

- The job queue definition can include a FAMILY attribute that corresponds to the FAMILY task attribute. However, the FAMILY queue attribute is not exactly a default or a limit. Rather, it excludes any job from the job queue if the job header includes a FAMILY assignment different from the FAMILY queue attribute. You can use the QF command to determine whether a job queue has a FAMILY queue attribute.

Requesting the Queue for a Job

If you have decided that a specific job queue is most appropriate for your job, then you can request the job queue through a CLASS assignment in the job header. For example, the following job requests queue 10:

```
?BEGIN JOB;
  CLASS = 10;
RUN OBJECT/PROG;
?END JOB
```

If the job does not include a CLASS assignment, it can inherit a value from the CLASS usercode attribute. An inherited CLASS value has the same effect as an assigned CLASS value.

The system evaluates the eligibility of a job for a requested job queue based on the factors discussed previously: queue resource usage limits, usercode class limits, unit queue assignments, and the FAMILY value. If the job qualifies for the requested queue, the system places the job in the queue. If the job does not qualify for the requested queue, the system rejects the job and displays the message “Q-DS.”

If the job has no assigned or inherited CLASS value, the system attempts to find an appropriate job queue in which to place the job. The method the system uses for making this selection depends on whether the operating system compile-time option QFACTMATCHING is set.

If the job has no CLASS assignment and QFACTMATCHING is set, then the system examines the various job queues to determine their eligibility for receiving the job. The system selects the first job queue that meets the following criteria:

- Any resource limits specified for the queue are greater than or equal to the corresponding resource limits in the WFL job header. For example, if the queue has a PRIORITY limit of 50, the job must have either no PRIORITY assignment in the job header or a PRIORITY assignment less than 51.
- The job queue must be one that is legal for a job with this usercode.

If the job has no CLASS assignment and QFACTMATCHING is reset, then the system selects the default job queue. The system administrator defines the default job queue using the DQ (Default Queue) system command. If no default queue has been defined, the system checks all the job queues, just as it would if QFACTMATCHING were set.

Whether QFACTMATCHING is set or not, the system performs an additional check. If the job queue selected by the system has a FAMILY attribute and the job has a FAMILY assignment in the job header, the system checks to see whether they match. If they do not specify identical family values, the system rejects the job and displays a “Q-DS” message.

Specifying a Start Time

You can use the STARTTIME task attribute to specify the earliest time and date that a particular job can be selected from a job queue. This task attribute can be assigned only to WFL jobs. It can be assigned in the task attribute list of the WFL job or in the statement that initiates the WFL job. You can also use the STARTTIME (Start Time) system command or the CANDE *?STARTTIME* command to assign this attribute to a job in a job queue. Changes made using these commands are maintained permanently, even if a halt/load occurs.

When you initiate a job with a STARTTIME specification, the job is compiled immediately and placed in an appropriate job queue. The job remains in the job queue at least until the date and time specified by the STARTTIME. You can use the SQ (Show Queue) system command to display the STARTTIME of jobs in a queue. The following is an example of the output for the command SQ 2:

```
QUEUE 2
 6643 01 TEST/WFL (#0001)
         QUEUED: 07/13/2001 AT 15:41:31   STARTTIME = 07/21/2001 AT 18:00:00
```

The STARTTIME specification provides a convenient means of scheduling a job for a time when the system load is lighter, such as in the evening or during a weekend. STARTTIME is also a convenient means of scheduling jobs that must run at regular intervals, such as every morning. The following example job, which is stored in the file (JASMITH)WFL/RUN, restarts itself on a daily basis:

```
?BEGIN JOB WFL/RUN;
  RUN OBJECT/PROG;
  START (JASMITH)WFL/RUN;STARTTIME = 10:00 ON +1
?END JOB
```

Structuring the WFL Job

A complete WFL job is considered a block, and each subroutine declared in the job is a block as well. The WFL job can enter or initiate subroutines. WFL automatically protects against critical block exits by performing an implicit wait at the end of the block that contains a task initiation statement. Control does not exit this block until all tasks initiated in that block have terminated.

WFL includes CASE, DO UNTIL, GO, IF, and WHILE DO statements that you can use to direct the flow of control in a job. By using these statements together with task attribute interrogations, a WFL job can provide conditional control over tasks. For example, the job can initiate the SYSTEM/PATCH utility as a task. When SYSTEM/PATCH terminates, the job can interrogate the task attributes of the SYSTEM/PATCH task. If the attribute values indicate that SYSTEM/PATCH ran without errors, the job can compile the merged source program. If the compilation is free of errors, the job can run SYSTEM/XREFANALYZER to produce an analysis of cross-references in the program.

Initiating Dependent Processes from WFL

In WFL, the RUN statement can be used to initiate an object code file as a synchronous dependent process. The TYPE task attribute of the resulting process shows a value of CALL. The initiated program can be written in any language except WFL.

The *PROCESS* keyword is used as a modifier in front of other initiation statements to cause the process to run asynchronously. Thus, a PROCESS RUN statement initiates an asynchronous task. The TYPE task attribute of the task has a value of PROCESS.

WFL cannot initiate a program as an independent process. In addition, a WFL job is never considered to be a coroutine; that is, a WFL job and its offspring cannot use CONTINUE statements to pass control back and forth.

There are some noteworthy differences between task initiation in WFL and task initiation in ALGOL or COBOL. In the latter two languages, RUN initiates an independent process and PROCESS initiates an asynchronous dependent process. Another difference is that WFL does not use external procedure declarations. In addition, there is no need to include a NAME task attribute assignment in WFL; the name of the object code file to be executed is specified in the RUN statement.

WFL jobs can also initiate internal procedures. An internal procedure in WFL is referred to as a *subroutine.* If the *PROCESS* keyword precedes a subroutine invocation statement, the system initiates the subroutine as an internal, asynchronous, fully dependent process. (If you do not use the *PROCESS* keyword, the subroutine invocation statement enters, rather than initiates, the subroutine.)

Initiating Compilations from WFL

A WFL job can initiate compilations by using the COMPILE statement. The COMPILE statement initiates a compiler and specifies the object code file to be compiled. The COMPILE statement can also include an object code file disposition, which specifies whether the object code file is to be executed once it is compiled, and whether the object code file is to be saved. The COMPILE statement can also be used to invoke the Binder. BIND is a synonym for the COMPILE statement.

Initiating Utilities from WFL

In addition to the RUN statement, WFL provides various special-purpose initiation statements. These statements include ADD, COPY, LOG, and PB. The COPY and ADD statements each initiate the visible independent runner LIBRARY/MAINTENANCE to copy a file. The LOG statement initiates the LOGANALYZER utility, and the PB statement initiates the BACKUP utility.

Initiating Interactive Processes from WFL

A WFL job can initiate an interactive process, but you might need to include a task attribute assignment for the interactive process to run properly. The STATIONNAME and STATION task attributes specify the station where any remote files used by the process are to be opened. WFL jobs initiated through a CANDE or MARC *START* command do not inherit the STATIONNAME or STATION of the remote terminal where they are initiated. You can remedy this problem by including the following statement at the start of the job:

```
MYJOB (STATIONNAME = #MYSELF(SOURCENAME));
```

This statement assigns the name of the station that initiated the job to the STATIONNAME attribute. This STATIONNAME value is inherited by all tasks initiated by the job. (Note that this assignment is lost across a halt/load. For details, refer to Section 11, "Restarting Jobs and Tasks.")

It is preferable to assign the STATIONNAME attribute rather than the STATION attribute, because the STATIONNAME attribute stores the station name, which is less volatile than the logical station number (LSN) stored by the STATION attribute.

Submitting Other WFL Jobs

A WFL job can include a START statement to initiate another WFL job. The START statement can initiate only WFL programs that are stored on disk files. This statement can include any of the parameter types that WFL recognizes. The START statement can also include an assignment to the STARTTIME task attribute, which specifies when the WFL job should be initiated.

Access to Task Attributes in WFL

A WFL job can include a job attribute list, which specifies task attributes to be applied to the job before initiation. Certain task attributes, if included in this list, can help determine the job queue in which the job is placed. The CLASS task attribute has the most direct effect on job queue selection; for more information about the CLASS attribute, refer to "Selecting the Queue for a Job" later in this section.

A WFL job can specify initial values for the attributes of a task if you include a task equation list in a task initiation statement. All task initiation statements in WFL (including RUN, COPY, and COMPILE) allow the use of task equations.

A WFL job can also use task variables to interrogate or modify the task attributes of a process. The task variable becomes associated with a task by being included in the task initiation statement. Assignments to the task variable before task initiation have the same effect as task equations. A job can monitor and control an asynchronous task while it is executing by accessing its task variable. After a task terminates, the job can interrogate the task variable to return task history information.

A WFL job can use the predeclared task variable MYSELF to access the job's own task attributes. A job can also use the predeclared task variable MYJOB, which has the same meaning as MYSELF unless it is referred to in an asynchronous subroutine. For an asynchronous subroutine, MYJOB refers to the parent WFL job and MYSELF refers to the subroutine's task attributes.

The COMPILE statement can specify task attributes that are stored in the object code file created by the compilation. These task attribute values are used each time the object code file is executed, unless the values are overridden by task equations at run time. In addition, a WFL job can use the MODIFY statement to assign task attributes to an object code file that already exists.

WFL jobs can directly access all task attributes except for task-valued or event-valued task attributes and the HISTORYREASON task attribute.

In general, the syntax for accessing task attributes in WFL is simpler than that used in ALGOL. Mnemonic-valued task attributes return string values rather than integers. Pointer-valued task attributes also return string values. Attributes that record resource usage, such as ACCUMPROCTIME, return values in units of seconds instead of 2.4 microseconds.

Using File Equations in WFL

Assignments to the FILECARDS task attribute are referred to as file equations. In WFL jobs, FILECARDS can be abbreviated to FILE. Using this task attribute, the job can modify the attributes of the logical files used by the task. The TITLE attribute can be used to cause the task to use a different physical file than it otherwise would.

You can include a construct called a *global file assignment* in a WFL job to cause an offspring to use a file declared in the WFL job. A global file assignment assigns a particular file declared by the WFL job to a particular internal name used by the offspring. Whenever the offspring attempts to use the file with that internal name, the system causes it to use the global file instead. This mechanism amounts to a hidden call-by-reference parameter because the job and its offspring use the same logical file.

A unique feature of WFL is the ability to include data specifications in the WFL source program. Whenever an offspring attempts to read from a card reader file, it reads instead from a data specification, if one is available. You can also use data specifications to replace other kinds of input files used by an offspring. To do this, you must include a file equation in the statement that initiates the offspring. The file equation must assign the input file a KIND file attribute value of READER. The offspring then reads lines from the data specification as if they were lines of the input file; for this reason, data specifications are also known as *pseudo-reader files*.

Responding to Error Conditions in WFL

Use the ON TASKFAULT statement to specify actions to be taken if a task terminates abnormally or if a compilation is terminated for syntax errors. WFL can also interrogate the values of the STATUS and HISTORYTYPE task attributes after a task terminates to determine the type of termination and take appropriate action.

Communicating with Other Processes in WFL

WFL jobs can communicate with their tasks by using any of several methods. The following list reviews each method of interprocess communication:

- Globally declared objects

  A subroutine initiated with a *PROCESS <subroutine>* statement can access objects declared globally to the subroutine in the WFL job.

- Parameters

  The RUN statement can include Boolean, integer, real, or string parameters. By default, these parameters are call-by-value parameters. However, you can specify that a parameter is call-by-reference by including the word *REFERENCE* after the parameter. A WFL job and an asynchronous task can communicate by interrogating and modifying the value of a call-by-reference parameter.

- Events

  A WFL job cannot declare events or interrupts and cannot access event-valued task attributes directly. However, a WFL job can use the WAIT statement, which can wait on many different types of implicitly declared events. For example, the simple form of the WAIT statement waits on the job's exception event. A job can also use WAIT statements to wait for a task to terminate or for one of the task attributes to attain a specified value. A WFL job can also access the LOCKED task attribute. LOCKED is a Boolean task attribute that acts like an event.

- Libraries

  Libraries cannot be written in WFL, nor can WFL jobs use libraries written in other languages.

- Port files and disk files

  WFL jobs cannot read from or write to files. A WFL job can create a single disk file and specify the contents of that file by using the DECK statement. However, the DECK statement, if used, must be the only statement in the job. Another useful feature is the ability of WFL to create a dummy file by simply declaring a file, opening it, and closing it. Such files can be used as flags to other processes. For example, a WFL job can perform a file-residence inquiry to determine whether a file with a certain title exists.

For details about any of these interprocess communication methods, refer to Part II of this guide, “Interprocess Communication.”

Controlling a Task from a Job

You can use the LOCKED attribute to control a task from its parent job, perhaps by causing it to wait while the job performs some action. If you set the LOCKED attribute to TRUE, it acts as an interlock, which blocks any subsequent attempt to set the attribute to TRUE until the attribute has been set to FALSE.

The following code excerpts show a possible use:

```
JOB
   BEGIN JOB J;
   TASK T;
   T (LOCKED=TRUE);
  PROCESS RUN program[T];
   .
   .
   .
   T (LOCKED=FALSE);
   END JOB

TASK
   BEGIN;
   % Perform initialization before synchronizing with JOB
   .
   .
   .
   T (LOCKED=TRUE); % Wait until job is ready
   .
   .
   .
   END
```

Restarting WFL Jobs

A WFL job automatically restarts if interrupted by a halt/load. WFL is the only language with this automatic restart capability. WFL also plays an important role in the restarting of checkpointed processes. These processes must be offspring of a WFL job in order to be checkpointed. In addition, the WFL *RERUN* statement is the means used to restart a checkpointed process. For further information, refer to Section 11, “Restarting Jobs and Tasks.”

WFL Example

The following example illustrates some WFL capabilities for task initiation and control:

```
?BEGIN JOB AUTOPB/HELP(STRING SOURCE, STRING PATCH);
  JOBSUMMARY = SUPPRESSED;
  DISPLAYONLYTOMCS = TRUE;
  CLASS = 15;
TASK T;
STRING RUN1, HELPTITLE;
HELPTITLE:= (PATCH & "/LEVEL1/HELPBOOK");
RUN1:= ("SOURCE=" & SOURCE
       & ",PATCH=" & PATCH
       & ",OUT=" & PATCH & "/LEVEL1/ED"
       & ",HELP=" & HELPTITLE
       & ",MESSAGEFILE=" & PATCH & "/LEVEL1/MESSAGES");
DISPLAY "RUNNING AUTOPB WITH " & RUN1;
RUN OBJECT/AUTOPB ON DOCMAST(RUN1) [T];
  FILE TEACHUTILNAME=*SYSTEM/HELP/UTILITY ON DOCMAST;
IF T(TASKVALUE) NEQ 1
   THEN BEGIN
          DISPLAY "HELPBOOK NOT CREATED; PRINTING ERRORS FILE";
          RUN *OBJECT/AUTOLP ON DOCMAST;
             TASKVALUE = 1;
             FILE SOURCE = #PATCH/LEVEL1/MESSAGES;
        END;
?END JOB
```

The main point of this job is to run a program called AUTOPB. The AUTOPB program accepts two input files, SOURCE and PATCH, and produces three output files, OUT, HELP, and MESSAGEFILE.

The job accepts two string parameters that provide the titles of the SOURCE and PATCH files. Using these, the job constructs an elaborate string parameter to pass to AUTOPB. This string parameter defines the titles for all the input and output files.

AUTOPB sets its own TASKVALUE to 1 unless it finds errors in the input files. The job inspects the TASKVALUE after AUTOPB terminates and prints out the MESSAGEFILE if there are errors.

ALGOL

ALGOL is a structured, high-level programming language with advanced computational and I/O capabilities. ALGOL also provides the most complete process initiation and control capabilities of any language available.

Closely related to ALGOL are several extended versions of the ALGOL language. DCALGOL is an extended ALGOL that includes some system control and data communications interfaces. DMALGOL includes special constructs for data management software. BDMSALGOL contains extensions for accessing Enterprise Database Server databases. In the following discussion, the features described are available in each of these languages, except where otherwise noted.

For further information about the ALGOL tasking features discussed in the following subsections, refer to the *ALGOL Programming Reference Manual, Volume 1: Basic Implementation*.

Structuring an ALGOL Program

The following ALGOL structures are considered blocks:

- Any complete ALGOL program. A complete ALGOL program can be initiated but cannot be entered.
- Any typed procedure; that is, any procedure designed to be invoked as a function that returns a value. (For example, Boolean procedures or real procedures.) Typed procedures can be entered and initiated. However, if a typed procedure is initiated, the returned value is discarded.
- A simple block, which is any group of declarations and statements that appears between the words BEGIN and END and is not preceded by a procedure heading. (An exception is the outer block of the program, which is not considered a simple block.) Such a block cannot be entered or initiated. The block is executed when control passes either from the previous statement in the program or from a GO TO statement elsewhere in the program. (Note that a BEGIN...END statement is not treated as a block if it does not include any declarations. In this case, it is simply a compound statement.)

When you initiate one of these ALGOL structures, the system creates a process stack. When you enter one of these ALGOL structures, the system creates an activation record. When a BEGIN...END block that includes declarations is executed, the system also creates an activation record.

An ALGOL program that initiates an asynchronous process should usually include a wait statement to prevent the critical block from being exited while the offspring is in use. An example of this wait statement is given in Section 2, “Understanding Interprocess Relationships.”

ALGOL includes an abundance of flow-of-control statements, such as CASE, DO, FOR, IF and WHILE. By using these statements together with task attribute interrogations, an ALGOL program can provide conditional control over tasks.

Initiating Processes from ALGOL

An ALGOL program can initiate any procedure, including imported library procedures, passed external procedures, and separate programs. However, if a typed procedure is initiated, the returned value is discarded.

To initiate another object code file, an ALGOL program must declare an external procedure and a task variable. The program must also assign the title of the object code file to the NAME attribute of a task variable. The program can then initiate the object code file with a process initiation statement that specifies the declared external procedure and task variable that were previously prepared.

Three process initiation statements are available. The CALL statement initiates a dependent, synchronous process. The PROCESS statement initiates a dependent, asynchronous process. The RUN statement initiates an independent process.

You can implement coroutines in ALGOL using CALL and CONTINUE statements. The CALL statement creates an active coroutine and changes the initiating process into a continuable coroutine. Coroutines can pass control back and forth by using CONTINUE statements.

Initiating Compilations from ALGOL

ALGOL does not provide any statement specifically for initiating compilations. However, an ALGOL program can submit a WFL job that includes a COMPILE statement. Alternatively, an ALGOL program can initiate a compiler like any other program, with a CALL, PROCESS, or RUN statement. An example of this method is given under “ALGOL Examples” later in this section.

Initiating Utilities from ALGOL

ALGOL does not provide any statements specifically for initiating utilities. However, the CALL, PROCESS, and RUN statements can initiate any utility and pass any parameters that are required by that utility. An example of an ALGOL program that initiates the LOGANALYZER utility is given under “ALGOL Examples” later in this section.

Initiating Interactive Processes from ALGOL

An ALGOL program initiated from a MARC or CANDE session inherits the STATION task attribute of the session. The STATION attribute is in turn inherited by any processes initiated by the ALGOL program. As a result, the processes initiated by the ALGOL program can usually open a remote file at the originating terminal without having to make any special remote file assignments.

However, remote file opens can fail because the STATION attributes stores the logical station number (LSN), which is subject to change. To prevent this problem, you can include statements such as the following in the ALGOL program:

```
EBCDIC ARRAY SOURCEARRAY[0:255];
REPLACE SOURCEARRAY BY MYSELF.SOURCENAME;
REPLACE MYSELF.STATIONNAME BY SOURCEARRAY;
```

The preceding statements assign the originating station name to the STATIONNAME task attribute, which in turn is inherited by all tasks.

An ALGOL program initiated from a WFL job or from an ODT might not inherit a STATION or STATIONNAME value. For further information, refer to "Work Flow Language (WFL)" in this section and to the ODT discussion in Section 3, "Tasking from Interactive Sources."

Submitting WFL Jobs from ALGOL

You can use the ZIP statement to submit a WFL job for execution. You can store the WFL job source in a disk file or in an array in the ALGOL program itself. Note that messages produced by the WFL job or its descendants will not be forwarded to the CANDE or MARC session that originated the ALGOL program. However, you can use the CANDE *?MSG* command or the MARC *SMSG* command to display these messages.

Access to Task Attributes in ALGOL

An ALGOL program can declare task variables for use in accessing the task attributes of offspring processes. Every process-initiation statement must specify a task variable, which thereafter is associated with the new process. An ALGOL program can interrogate or assign task attribute values of the task variable before or after the task variable is used in a process initiation statement. Assignments made to a task variable before initiation are saved and applied to the process at initiation time.

An ALGOL program can use the predeclared task variables MYSELF and MYJOB to access its own task attributes and those of its job.

An ALGOL program can interrogate and modify task attributes that store any of the possible data types, such as Boolean, integer, and so on. The task attribute types available in ALGOL include two types that are not available in WFL: event and task.

Communicating with Other Processes from ALGOL

ALGOL programs have full access to all of the interprocess communication methods discussed in this guide, including globally declared objects, call-by-reference or call-by-name parameters, events and interrupts, port files, and libraries. For details about any of these interprocess communication methods, refer to Sections 13 through 21 of this guide.

Restarting ALGOL Processes

An ALGOL program can include a CHECKPOINT statement that creates a checkpoint file. The checkpoint file stores information about the current state of a process. You can use the checkpoint file after a halt/load to restart the process. For further information, refer to Section 11, "Restarting Jobs and Tasks."

DCALGOL Features

In addition to the ALGOL features previously discussed, DCALGOL includes the CONTROLCARD function, which you can use instead of the ZIP statement to submit WFL jobs for execution. The CONTROLCARD function has several capabilities that are unavailable through ZIP. For example, the CONTROLCARD function can

- Specify whether the WFL job should be a dependent or independent process
- Compile the job for syntax checking only, without executing it
- Specify that any messages generated by the job be routed to an MCS for display in the originating session
- Define the invalid character to be something other than a question mark (?)
- Submit a job that is stored as a message in a DCALGOL queue

Additionally, the process that submits the CONTROLCARD function can determine whether the WFL job compiled without syntax errors. If a WFL job submitted through CONTROLCARD has syntax errors, the system assigns the value 1 to the TASKVALUE of the process that submitted the job.

A privileged DCALGOL process can also duplicate the process initiation and control capabilities that are available at an ODT. You can use the DCKEYIN statement to submit system commands to the operating system. The GETSTATUS and SETSTATUS functions directly invoke the operating system interfaces that are accessed by system commands. For information about ODT process initiation and control capabilities, refer to Section 3, "Tasking from Interactive Sources."

ALGOL Examples

The following sample program initiates a separate program called REPORTER. The REPORTER program is initiated twice, both times as an asynchronous task, and is passed a different parameter each time. The sample program then uses a WAITANDRESET statement to prevent a critical block exit.

```
BEGIN
EBCDIC ARRAY DAILYTYPE[0:5],
             WEEKLYTYPE[0:6];
TASK T, T2;
PROCEDURE REPORTS (ACTUALARRAY);
   EBCDIC ARRAY ACTUALARRAY[*];
EXTERNAL;

REPLACE T.NAME BY "(JASMITH)OBJECT/REPORTER ON DATAPK.";
REPLACE DAILYTYPE[0]  BY "DAILY";
PROCESS REPORTS (DAILYTYPE) [T];

REPLACE T2.NAME BY "(JASMITH)OBJECT/REPORTER ON DATAPK.";
REPLACE WEEKLYTYPE[0] BY "WEEKLY";
PROCESS REPORTS (WEEKLYTYPE) [T2];

WHILE (T.STATUS GTR 0 OR T2.STATUS GTR 0) DO
   WAITANDRESET (MYSELF.EXCEPTIONEVENT);
END.
```

The following is an example of initiating a compilation from an ALGOL program. The sample program passes an array parameter and makes FILECARDS assignments to tell the compiler what files to use:

```
BEGIN

TASK CTASK;
ARRAY SHEET[0:32];

PROCEDURE ALGOLCOMPILER(SHEET);
  ARRAY SHEET[*];
  EXTERNAL;


REPLACE CTASK.NAME BY "*SYSTEM/ALGOL ON DISK.";
REPLACE CTASK.FILECARDS BY
  "FILE CARD (KIND=DISK, TITLE=ALGOL/TASK);"
  "FILE CODE (KIND=DISK, TITLE=OBJECT/ALGOL/TASK);"
   48"00";

REPLACE SHEET BY 0 FOR 33 WORDS;
SHEET[8]:=VALUE(LIBRARY); % This statement specifies the
                          % object code file disposition.
SHEET[0]:= 0 & 1[47:1];
```

```
CALL ALGOLCOMPILER(SHEET) [CTASK];

END.
```

The following is an example of initiating a utility from ALGOL. This sample program includes a statement that initiates LOGANALYZER:

```
BEGIN

  TASK T;
  ARRAY ACTUAL_OPTIONS[0:19];

  PROCEDURE LOGRUN (FORMAL_OPTIONS);
    ARRAY FORMAL_OPTIONS[*];
    EXTERNAL;

  REPLACE T.NAME BY "*SYSTEM/LOGANALYZER ON DISK.";
  REPLACE ACTUAL_OPTIONS BY "PRINTER JOB 1260",48"00";

  CALL LOGRUN (ACTUAL_OPTIONS) [T];

  END.
```

The following ALGOL example submits WFL jobs for execution in two different ways. The first ZIP statement submits the WFL job stored in array WFLARRAY. The second ZIP statement submits the WFL job stored in the file WFL/TEST. Note the use of triple quotes (""") in the REPLACE statement wherever a single quote (") is to occur in the WFL program.

```
BEGIN
  EBCDIC ARRAY WFLARRAY[1:120];
  FILE WFLFILE(KIND=DISK,NEWFILE=FALSE,DEPENDENTSPECS=TRUE,
             TITLE="WFL/TEST.");

  REPLACE WFLARRAY BY
    "CLASS=2;JOBSUMMARY=SUPPRESSED;ELAPSEDLIMIT=120;"
    "MYSELF(STATIONNAME=#MYSELF(SOURCENAME));"
    "DISPLAY ("""HI""");";

  ZIP WITH WFLARRAY;
  ZIP WITH WFLFILE;
END.
```

COBOL

COBOL is available in two different implementations: COBOL74 and COBOL85. These correspond to the ANSI-74 and ANSI-85 levels of COBOL, respectively.

Both the COBOL implementations incorporate a full range of tasking capabilities. With few exceptions, the same statements for performing tasking functions are provided in each language. In the following descriptions, statements about COBOL refer to all three COBOL implementations unless otherwise specified.

For further information about COBOL85, refer to the *COBOL ANSI-85 Programming Reference Manual, Volume 1: Basic Implementation*. For further information about COBOL74, refer to the *COBOL ANSI-74 Programming Reference Manual, Volume 1: Basic Implementation*.

Structuring a COBOL Program

COBOL provides access to procedure-like subdivisions within the program, as well as to procedures outside the program.

Internal Procedure Structure

Paragraphs and sections within a COBOL program are not considered blocks, because executing a paragraph or a section does not result in the creation of an activation record. Paragraphs and sections therefore do not affect the definition of critical blocks.

If the Binder is used to bind a procedure from a separate object code file into the program, then the bound-in procedure is considered a separate block. The bound-in procedure could be another COBOL program or a procedure from a program written in some other language. A COBOL program can enter, but cannot initiate, a bound-in procedure.

COBOL85 provides a structure called *nested programs* that is not available in COBOL74. Nested programs are programs that reside inside another program or inside another nested program. Nested programs resemble ALGOL procedures in the respect that nested programs can include declarations of local variables. However, the rules determining the scope of variable declarations in COBOL85 differ from the scope rules in ALGOL. The COBOL85 scope rules are explained in the interprogram communication discussion in the *COBOL ANSI-85 Programming Reference Manual, Volume 1: Basic Implementation.*

COBOL85 can enter, but cannot initiate, nested programs.

Note: *COBOL85 is currently implemented in such a way that exiting a nested program cannot cause a CRITICAL BLOCK EXIT error. However, this implementation is subject to change. Nested programs may affect the critical block definition in future releases.*

Another structure unique to COBOL85 is that of *consecutive programs*. Consecutive programs are completely separate programs that are stored, one after the other, in the same source file. When you compile a source file that contains consecutive programs, the compiler creates a separate object code file for each consecutive program. The resulting object code files are not linked in any way and have no special abilities related to tasking or interprocess communication.

External Procedure Structure

The following rules govern COBOL access to external procedures:

- Separate programs

  A COBOL program can declare external procedures and use them to initiate separate programs.

- Passed external procedures

  COBOL does not provide any method for passing procedures as parameters. Therefore, a COBOL program generally has no access to passed external procedures.

  One exception to this rule occurs when a program passes a constant or an expression by name to a COBOL program. The system creates a procedure called a *thunk,* whose purpose is to evaluate the constant or expression. Whenever the COBOL program interrogates the parameter, the system executes the thunk on the COBOL program's process stack.

- Imported library procedures

  A COBOL program can enter, but cannot initiate, a procedure imported from a library.

A critical block exit error can occur if the COBOL program terminates before an asynchronous offspring or a coroutine. For information about how to prevent such critical block exits, refer to Section 2, “Understanding Interprocess Relationships.”

Initiating Processes from COBOL

A COBOL program can initiate separate programs as processes, but cannot initiate internal sections, paragraphs, or nested programs.

Separate object code files are initiated by statements that have the following general form:

```
<verb> <task variable> WITH <section name> [USING <parameter list>]
```

The verb in this statement can be CALL, which initiates a synchronous, dependent process; PROCESS, which initiates an asynchronous, dependent processor; or RUN, which initiates an independent process. EXECUTE is a synonym for RUN.

The task variable in this statement is a data item declared with a usage of TASK, CONTROL-POINT, or CP.

The section name in this statement must have been previously defined in the DECLARATIVES portion of the PROCEDURE DIVISION. The section name definition in the DECLARATIVES must be followed by a USE EXTERNAL statement.

The COBOL program must also associate an object code file title with a <section name> by one of the following methods:

- Using a mnemonic name in the SPECIAL-NAMES paragraph. This is the preferred method.
- Using a MOVE statement to assign the object code file title to the identifier specified in the USE EXTERNAL statement in the DECLARATIVES.
- Assigning the NAME task attribute to the task variable before task initiation. The title assigned must be a string enclosed in quotes and terminated with a period.

The *USING <parameter list>* clause passes parameters to the initiated program. If no parameters are to be passed, you can omit this clause.

Using Coroutines in COBOL

You can implement coroutines in COBOL using CALL, CONTINUE, and EXIT PROGRAM statements. The CALL statement creates a synchronous task that is an active coroutine and changes the parent process into a continuable coroutine. The task can return control to its parent by executing an EXIT PROGRAM statement. The parent can return control to its task by executing a *CONTINUE <task variable>* statement.

The EXIT PROGRAM statement, in addition to transferring control to the parent, also specifies where execution resumes when the parent later continues the task. The simple form EXIT PROGRAM specifies that the task resume from the beginning. The EXIT PROGRAM RETURN HERE form specifies that the task resume with the statement that follows the EXIT PROGRAM statement.

In COBOL85, the EXIT PROGRAM statement in a nested program merely causes the nested program to be exited. To return control from a synchronous task to a parent program, the EXIT PROGRAM statement must occur in the main program rather than in a nested program.

Entering Individual COBOL Procedures

COBOL allows the use of certain special formats for the CALL statement that enter, rather than initiate, a procedure.

A COBOL program can use a CALL statement with one of the following forms to enter a bound-in procedure:

```
CALL <section name>.
CALL <section name> USING <parameter list>.
```

A COBOL program can use any of several forms of the CALL statement to enter an imported library procedure. The following is an example:

```
CALL "PROCEDUREDIVISION OF OBJECT/COBOL/PROG" USING PARAM1.
```

COBOL85 allows you to optionally specify an access value of BYTITLE or BYFUNCTION in the CALL statement, as in the following example:

```
CALL "PROCEDUREDIVISION OF DELTASUPPORT BYFUNCTION" USING PARAM1.
```

COBOL85 also supports the use of explicit library declarations. For examples of libraries and user programs written in COBOL74 and COBOL85, refer to Section 18, “Using Libraries.”

By contrast, the GO and PERFORM statements do not enter procedures. They simply transfer control to a selected paragraph or section without creating an activation record.

Resolving Ambiguous CALL Statements in COBOL85

In COBOL85, the form of the CALL statement that invokes a nested program is identical to the form that implicitly invokes a library. The following is an example of such a statement:

```
CALL "QROUTINE".
```

By default, the system treats this as an invocation of a nested program. If the COBOL85 program does not contain a nested program with the requested name, or the nested program with that name is not declared with the appropriate scope, the system treats the CALL statement as an implicit library invocation. The system links the calling program to a library whose title matches the name specified in the CALL statement.

Initiating Compilations from COBOL

COBOL does not include any statement specifically for initiating compilations. However, a COBOL program can submit a WFL job that includes a COMPILE statement. Alternatively, a COBOL program can initiate a compiler like any other program, with a CALL, PROCESS, or RUN statement.

Initiating Utilities from COBOL

COBOL does not include any statements specifically for initiating utilities. However, the CALL, PROCESS, and RUN statements can initiate any utility and pass any parameters that are required by the utility.

Initiating Interactive Processes from COBOL

A COBOL program initiated from a MARC or CANDE session inherits the STATION task attribute of the session. The STATION attribute, in turn, is inherited by any processes initiated by the COBOL program. As a result, these processes can usually open a remote file at the originating terminal without having to make any special remote file assignments.

However, remote file opens can fail because the STATION attributes stores the logical station number (LSN), which is subject to change. To prevent this problem, you can include statements such as the following in the COBOL program:

```
MOVE ATTRIBUTE SOURCENAME OF MYSELF TO NAMEBUF.
CHANGE ATTRIBUTE STATIONNAME OF MYSELF TO NAMEBUF.
```

The preceding statement assigns the originating station name to the STATIONNAME task attribute, which in turn is inherited by all tasks. These statements assume that NAMEBUF was declared as *`01 NAMEBUF PIC X(80).`*

A COBOL program initiated from a WFL job or from an ODT might not inherit a STATION or STATIONNAME value. For further information, refer to “Work Flow Language (WFL)” earlier in this section and to “Communicating with an ODT” in Section 3, “Tasking from Interactive Sources.”

Submitting WFL Jobs from COBOL

A COBOL85 or COBOL74 program can submit WFL jobs with a statement of the following form:

```
CALL SYSTEM WFL USING <identifier or literal>
```

If a literal is used in this statement, it must be a string literal consisting of the complete text of a WFL source program. If an identifier is used in this statement, the identifier must be that of a data item that contains the complete WFL source program.

Note that when a COBOL program submits a WFL job, any messages produced by the WFL job or its descendants are not forwarded to the CANDE or MARC session that originated the COBOL program. However, you can use the CANDE *?MSG* command or the MARC *SMSG* command to display these messages.

Access to Task Attributes in COBOL

A COBOL program can access task attributes by using a task variable. A COBOL program can create a task variable by declaring a data item with a USAGE of TASK, CP, or CONTROLPOINT in the DATA DESCRIPTION entry.

The MYSELF and MYJOB task variables are available in COBOL and enable a COBOL program to access its own task attributes or those of its job.

A COBOL program can read task attribute values by using the MOVE statement, and can set task attributes using the CHANGE statement.

COBOL programs can use all types of task attributes, including event-valued and task-valued task attributes.

Invoking COBOL Programs

Most COBOL74 programs can be invoked in either of two ways: through process initiation statements or through the library linkage mechanism. If the program is invoked through the library linkage mechanism, the program automatically freezes and exports the PROCEDURE DIVISION. This automatic freeze occurs even though the program does not include a FREEZE statement or export declaration. For further information about COBOL library capabilities, refer to Section 18, “Using Libraries.”

By contrast, a COBOL85 program can be initiated as a library only if it includes the $SET LIBRARYPROG compiler control statement or the CALL SYSTEM FREEZE statement.

Communicating with Other Processes from COBOL

COBOL programs have access to almost all the interprocess communication methods discussed in this guide, including call-by-reference parameters, events and interrupts, port files, and libraries. The only interprocess communication method that does not apply to COBOL is the use of globally declared objects, because COBOL cannot initiate an internal procedure. For details about any of these interprocess communication methods, refer to Sections 13 through 21.

COBOL85 also supports ANSI-defined methods of *interprogram* communication. For the most part, ANSI interprogram communication simply defines the relationships between nested programs that are all executed as part of a single process. By contrast, the *interprocess* communication features of COBOL85 are enhancements that allow data to be exchanged between separate processes.

Terminating Processes from COBOL Programs

You can use the DETACH statement to terminate the process associated with a given task variable. The task variable must have been previously used in a CALL, PROCESS, or RUN statement.

The DETACH <task variable> statement has the same effect as using the CHANGE statement to assign the STATUS task attribute a value of TERMINATED. However, from the standpoint of COBOL programming style, the DETACH is the preferable statement to use for terminating processes.

The program that contained the DETACH statement continues execution asynchronously while the system terminates the specified process. If a program is to reuse the task variable in another process initiation statement, the program should first read the STATUS attribute to determine whether the system has finished terminating the process. If the process is terminated, then the STATUS attribute returns a value of TERMINATED.

The DETACH statement is also used to detach events from interrupt procedures. This use of DETACH is described under "Attaching or Detaching an Interrupt" in Section 16, "Using Events and Interlocks."

COBOL Examples

The following program can be compiled in COBOL74 or COBOL85. This program initiates a separate program called OBJECT/COBOL/TEST using the task variable TASK-VAR-1:

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 TASK-VAR-1  USAGE IS TASK.
01 EXT-NAME        PIC X(80).
PROCEDURE DIVISION.
DECLARATIVES.
PROC-EXTERNAL SECTION.
   USE EXTERNAL EXT-NAME AS PROCEDURE.
END DECLARATIVES.

START-HERE SECTION.
P1.
    MOVE "OBJECT/COBOL/TEST." TO EXT-NAME.
    PROCESS TASK-VAR-1 WITH PROC-EXTERNAL.

PROCWAIT SECTION.
P2.
    WAIT AND RESET UNTIL ATTRIBUTE EXCEPTIONEVENT OF MYSELF.
    IF ATTRIBUTE STATUS OF TASK-VAR-1 IS GREATER THAN
       VALUE TERMINATED THEN GO PROCWAIT.
STOP RUN.
```

The following program can be compiled in COBOL74 or COBOL85. This program invokes OBJECT/COBOL/TEST as an imported library procedure rather than as a task. OBJECT/COBOL/TEST is executed as part of the calling process.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
START-HERE SECTION.
P1.
    CALL "PROCEDUREDIVISION IN OBJECT/COBOL/TEST".
STOP RUN.
```

The following is the program OBJECT/COBOL/TEST. If this program is compiled in COBOL74, then it can be invoked by either of the two preceding programs, and execute either as a task or as a library. If this program is compiled in COBOL85, then it can be invoked only as a task. If you add the $SET LIBRARYPROG compiler control statement and compile this program in COBOL85, then the program can be invoked as a library.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 MIXNO  BINARY PIC 9(11).
PROCEDURE DIVISION.
START-HERE SECTION.
P1.
    MOVE ATTRIBUTE MIXNUMBER OF MYSELF TO MIXNO.
    DISPLAY MIXNO.
STOP RUN.
```

The following program can be compiled in COBOL74 or COBOL85. This program submits WFL input in array form for execution. The WFL statements are stored in an array of picture items. Note that if any of the WFL statements includes a quotation mark ("), the quotation mark must be represented by two quotation marks ("") in the MOVE statement that stores the statement in the array. The use of double quotation marks is necessary because the compiler interprets a single quotation mark as the end of the WFL input rather than as part of the WFL input.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 PARAM.
   05 PARAM-1      PIC X(80).
   05 PARAM-2      PIC X(80).
   05 PARAM-3      PIC X(80).

PROCEDURE DIVISION.
START-HERE SECTION.
P1.
MOVE "CLASS=2;JOBSUMMARY=SUPPRESSED;ELAPSEDLIMIT=120;" TO PARAM-1.
MOVE "MYSELF(STATIONNAME=#MYSELF(SOURCENAME));" TO PARAM-2.
MOVE "DISPLAY (""HI AGAIN"");" TO PARAM-3.
CALL SYSTEM WFL USING PARAM.

STOP RUN.
```

The following program can be compiled in COBOL74 or COBOL85. This program initiates a utility. This example also shows how to pass parameters to a task from a COBOL program.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 TASK-VAR-1  USAGE IS TASK.
01 EXT-NAME        PIC X(80).
01 ACTUALPARAM     PIC X(19).
LOCAL-STORAGE SECTION.
LD PARAMS.
 01 FORMALPARAM     PIC X(19).
PROCEDURE DIVISION.
DECLARATIVES.
PROC-EXTERNAL SECTION.
    USE EXTERNAL EXT-NAME AS PROCEDURE
    WITH PARAMS USING FORMALPARAM.
END DECLARATIVES.
START-HERE SECTION.
P1.
    MOVE "*SYSTEM/LOGANALYZER ON DISK." TO EXT-NAME.
    MOVE "PRINTER JOB 1260" TO ACTUALPARAM.
    CALL TASK-VAR-1 WITH PROC-EXTERNAL USING ACTUALPARAM.
    STOP RUN.
```

The following program can be compiled in COBOL74 or COBOL85. This program initiates a compilation:

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 TASK-VAR-1  USAGE IS TASK.
01 EXT-NAME        PIC X(80).
01 VALUE-ONE       PIC 9(11) BINARY VALUE 1.
01 ACTUALPARAM.
     03 PARAMWORD BINARY PIC 9(11) OCCURS 33.
LOCAL-STORAGE SECTION.
LD PARAMS.
01 FORMALPARAM.
    03 FORMALWORD BINARY PIC 9(11) OCCURS 33.
PROCEDURE DIVISION.
DECLARATIVES.
PROC-EXTERNAL SECTION.
    USE EXTERNAL EXT-NAME AS PROCEDURE
    WITH PARAMS USING FORMALPARAM.
END DECLARATIVES.
START-HERE SECTION.
P1.
    MOVE "*SYSTEM/ALGOL ON DISK." TO EXT-NAME.
    MOVE 2 TO PARAMWORD (9).
    MOVE VALUE-ONE TO PARAMWORD (1) [00:47:01].
    CHANGE ATTRIBUTE FILECARDS OF TASK-VAR-1 TO
       "FILE CARD (KIND=DISK,TITLE=ALGOL/TASK);".
    CHANGE ATTRIBUTE FILECARDS OF TASK-VAR-1 TO
       "FILE CODE (KIND=DISK,TITLE=OBJECT/ALGOL/TASK);".
    CALL TASK-VAR-1 WITH PROC-EXTERNAL USING ACTUALPARAM.
    STOP RUN.
```

In this example, the COBOL program initiates the compiler directly as a task. An alternative would be for the program to submit in array form a WFL program that contains a COMPILE statement.

Other Languages

The other user languages available are C, FORTRAN77, Pascal, and RPG. These languages are not primarily intended for process initiation and control. However, most of these languages have one or more of the following tasking capabilities:

- Submitting WFL jobs

  If a program can submit a WFL job, the job, in turn, can initiate and control programs written in any language.

- Invoking library procedures

  You can implement ALGOL or COBOL libraries that export procedures that initiate or control processes. Any language using libraries can invoke these procedures.

- Using bound-in procedures

  You can bind ALGOL procedures or complete COBOL programs into programs written in other languages. These bound-in procedures can be designed to initiate and control processes.

The following are the tasking capabilities of each language.

- C

  Includes tasking features defined by the POSIX standards. These features include the fork() procedure and the exec() procedures. For an overview of process handling in C, refer to the *POSIX User's Guide*.

  C programs can also invoke library procedures in libraries written in other languages. For further information, refer to the *C Programming Reference Manual, Volume 1: Basic Implementation.*

- FORTRAN77

  Can invoke library procedures in libraries written in other languages. Additionally, you can add tasking features to a FORTRAN77 program by binding in ALGOL procedures or COBOL programs. For further information, refer to the *FORTRAN77 Programming Reference Manual.*

- Pascal

  Can invoke library procedures in libraries written in other languages. Additionally, you can add tasking features to a Pascal program by binding in ALGOL procedures or COBOL programs. For further information, refer to the *Pascal Programming Reference Manual, Volume 1: Basic Implementation*.

- RPG

  Can include ZIP statements that submit WFL jobs for execution. An RPG program can use the external indicators U1 through U8 to interrogate the SW1 through SW8 task attributes. For further information, refer to the *Report Program Generator (RPG) Programming Reference Manual, Volume 1: Basic Implementation*.

Section 5
Establishing Process Identity and Privileges

Process identity is the term used in this guide for a number of task attributes and other features that uniquely identify a process and its capabilities. This section explains:

- The various aspects of process identity and their implications for security, billing, and operations
- The security classes a process can belong to, and the privileges associated with each of these classes
- Issues related to initiating processes with different USERCODE or FAMILY values than those of the initiator
- The way an interactive program can temporarily assume the identity of a user

Process Identity

Some of the aspects of process identity, such as mix numbers, are assigned by the operating system. You can control other aspects of process identity, such as the usercode, although the system provides default values for these aspects if you do not.

Mix Number, Session Number, Stack Number, and MPID Attribute

In Section 1, "Understanding Basic Tasking Concepts," it was pointed out that there may be multiple processes running that are instances of the same object code file. Thus, the object code file title cannot serve as a unique identification for a process. Instead, the system assigns two identifying numbers to a process: the *mix number* and the *stack number*. A related type of identifying number, which applies to CANDE and MARC sessions, is the *session number*. In addition, the *MPID task attribute* can be used to identify a process. This MPID task attribute is especially useful when more than one process using the same object code file is running.

The mix number is a 5-digit number that the system assigns to each process when the process is initiated. The name arises because all the processes running on the system are collectively referred to as the system *mix*, and the mix number distinguishes a process from the other processes in the mix. Mix numbers identify processes in the system log as well as in many system commands and CANDE commands that affect running processes or provide information about them.

The system assigns mix numbers to WFL jobs when they are first entered into a job queue. This mix number remains the same when the WFL job leaves the queue and begins executing.

Each process or queued WFL job is generally assigned a mix number one higher than the last assigned number in the mix number pool. However, a jump in the numbering can occur after a halt/load. This jump occurs because the system periodically reserves a range of numbers for use by new processes. After a halt/load, the system avoids reusing any number in the range reserved at the time of the halt/load.

The mix number pool is determined by the maximum mix number that is configured on the system. When the system is running in an active MoreTasks state (OP+ MORETASKS or OP+ 32 followed by a halt/load), the maximum mix number can be configured between 9999 and 65535 using the MAX MIX = <number> form of the MAX (Maximums) system command. When the system is running in a nonactive MoreTasks state (OP– MORETASKS or OP– 32 followed by a halt/load), the maximum mix number is fixed at 9999.

When the mix numbers reach the maximum configured mix number, which ranges from 9999 to 65535, the number starts over at 100. Certain low numbers are reserved for some operating system processes. Also, a number cannot be reused if the last process to which it was assigned is still in the mix.

Processes can determine their own mix numbers, or the mix number of a related process, by interrogating the MIXNUMBER task attribute. However, the mix number has little use in programmatic tasking. A process accesses the task attributes of another process by specifying a particular task variable, rather than by specifying a particular mix number.

A concept related to mix numbers is that of session numbers. The system assigns session numbers to identify individual CANDE and MARC sessions. This session number is inherited by the JOBNUMBER task attribute of tasks initiated from a session.

The system selects these session numbers from either the mix number pool (determined by the value of the system command, MAX MIX = <number>), or a separate session number pool (determined by the value of the MAX SESSION = <number>). The following are the effects of this command:

- The MAX SESSION value cannot be less than the MAX MIX value. An increase in the MAX MIX value results in a corresponding increase in the MAX SESSION value. An attempt to reduce the MAX SESSION value to be less than the MAX MIX value results in no change in the MAX SESSION value.
- If the MAX SESSION value is the same as the MAX MIX value, then the system allocates session numbers from the mix number pool, which ranges from 100 to the MAX MIX value. The system never assigns the same number to both a process and a session.

- If the MAX SESSION value is greater than the MAX MIX value, then the system allocates session numbers from a separate session number pool. The numbers in this pool range from the MAX MIX value +1 through the MAX SESSION value. If a session number is not available in the session number pool, a value is selected from the mix number pool.

  The use of a session number pool enables many more sessions to be active than would otherwise be possible, because you can create a session number pool that is larger than the mix number pool, and none of the numbers are used as mix numbers.

Like the mix number, the stack number of a process is a number assigned by the operating system. The stack number is unique to the process and remains constant for the lifetime of the process. However, while the mix number is intended primarily for use by system operators, the stack number of a process is intended primarily for internal use by the operating system. Yet the stack number is visible to operators and programmers in the following contexts:

- The stack number appears in the system log records for Major Type 0, Minor Type 1 (Establish Identity) and Major Type 1, Minor Types 1 (BOJ), 2 (EOJ), 3 (BOT), 4 (EOT), 5 (File Open), and 6 (File Close). The stack number is expressed in hexadecimal format.
- The output from the OT (Inspect Stack Cell) system command includes the stack number for the process. The stack number is expressed in hexadecimal format. Thus, in the following output, the stack number is 011B:

  ```
  011B STACK CELL 20= 7 09624650E003 (HEX)
  ```

- The PROCESSID function in ALGOL returns the stack number of the process. The stack number is expressed in integer format.
- The stack number can appear in memory dump analyses created by DUMPANALYZER. The stack number is reported in hexadecimal format.
- The STACKNUMBER task attribute returns the stack number of a process. The stack number is returned in integer format.

The MPID attribute is not assigned by the operating system, but can be assigned to a process before initiation. The attribute remains constant for the lifetime of the process. The MPID attribute enables system operators to distinguish between multiple instances of a process using a single object code file. The value of the attribute is visible to operators in the following contexts:

- The value is shown in response to mix-related system commands (A, C, J, LIBS, M, S, and W). It appears as a quoted string after the process name.
- The value appears in system log records for Major Type 0, Minor Type 1 (Establish Identity), and Major Type 1, Minor Types 1 (BOJ) and 3 (BOT).

Usercode, Access Code, Charge Code, and Group Code

USERCODE, ACCESSCODE, CHARGE, and GROUPCODE are closely related task attributes that help to specify the identity and privileges of a process.

The USERCODE task attribute stores a value that is intended to identify the user who initiated the process. In actual practice, more than one user of the system can use the same usercode, but only if all the users agree to do so. This is because you must know the password associated with a usercode to use the usercode, and only the owner of the usercode can tell you the password.

Usercodes are created by the security administrator for the system, usually by using the MAKEUSER utility. The security administrator can associate a variety of *usercode attributes* with each usercode. Some of these usercode attributes confer various types of special security privileges, as described under "Process Security Classes" later in this section.

Other usercode attributes interact with the values of various task attributes. Some of these usercode attributes provide default values for the corresponding task attributes. Other usercode attributes define a range of permitted values for a task attribute, or specify whether the task attribute must have a value. The following are these usercode attributes and the task attributes that are related to them:

Usercode Attribute	**Task Attribute**
ACCESSCODELIST, ACCESSCODENEEDED	ACCESSCODE
CHARGECODE, CHARGEREQ, USEDEFAULTCHARGE	CHARGE
CANDEDESTNAME	DESTNAME
CLASS, CLASSLIST, ANYOTHERCLASSOK	CLASS
CONVENTION	CONVENTION
DEPTASKACCOUNTING	DEPTASKACCOUNTING
FAMILY	FAMILY
FILEACCOUNTING	FILEACCOUNTING
GROUPCODE	GROUPCODE
LANGUAGE	LANGUAGE
PRINTDEFAULTS	PRINTDEFAULTS
PRIORITY	PRIORITY
SAVEMEMORYLIMIT	SAVEMEMORYLIMIT
SUPPLEMENTARYGRPS	SUPPLEMENTARYGRPS
TEMPFILELIMIT	TEMPFILELIMIT

The values supplied by usercode attributes are propagated to their corresponding task attributes in the following ways:

- MARC and CANDE read some usercode attributes when you log on, and store the corresponding task attribute values for your session. Thereafter, if you initiate a process from that session, the process inherits the task attributes of the session.
- If a WFL job includes a USERCODE assignment in the job header, the WFL job inherits the attribute values associated with the usercode.

For details about the effects of usercode attributes on task attributes, refer to the task attribute descriptions in the *Task Attributes Programming Reference Manual*.

The ACCESSCODE task attribute serves as a form of secondary identification, in addition to the usercode. This identification is relevant only when a process attempts to use a file that is guarded by a guard file; refer to "Nonprivileged Status" later in this section for further details.

The CHARGE task attribute serves as a form of group identification for billing purposes. Thus, all the people working in a particular department might have usercodes with the same CHARGECODE usercode attribute. The system records the CHARGE attribute of each process in the system log. This makes it possible for site personnel to write billing programs that analyze the system usage on a charge code by charge code basis. For further information about billing programs, refer to the *System Administration Guide*.

The GROUPCODE and SUPPLEMENTARYGRPS task attributes provide another form of group identification. The GROUPCODE attribute identifies the primary group to which a process belongs, and the SUPPLEMENTARYGRPS attribute identifies one or more secondary groups. The group identification can affect the ability of a process to access files that have the GROUP file attribute set. For further information about group usage, refer to the *POSIX User's Guide*.

You can override the propagation of most usercode attributes to task attributes by explicitly assigning task attributes to the process in question. However, the system enforces some consistency checks to ensure that the USERCODE, ACCESSCODE, and CHARGE attribute values are consistent with each other. For details about these consistency checks, refer to the descriptions of these attributes in the *Task Attributes Programming Reference Manual*.

A process can change its own usercode while it is running by making an assignment to the USERCODE attribute. Such an assignment must specify the password as well as the usercode. The system verifies the correctness of the usercode and password before making the usercode assignment.

Name

The name of a process is stored in the NAME task attribute of the process. The value of this attribute is, by default, the same as the title of the object code file that the process is executing. The process name appears in system log entries generated for the process. The process name also appears in the output from system mix display commands such as A (Active Mix Entries), W (Waiting Mix Entries), and C (Completed Mix Entries).

In addition to aiding the operator, the process name can affect the ability of the process to use some files. If a file has a guard file associated with it, the guard file can include a PROGRAM clause that specifies access rights for processes with a given name.

In some cases, the NAME value for a process can be different from its object code file title. This can occur if a WFL process or an ALGOL process initiates an internal procedure. The initiating process can make an arbitrary assignment to the NAME attribute of the new process before initiating it.

The initiating process can even assign the internal process the NAME of an entirely different program. This method enables the process to circumvent the PROGRAM clause in a guard file. To prevent such abuses, a CODEFILE clause is also available for guard files. This clause ignores the process name and instead specifies access rights for processes having a particular object code file title. For details, refer to the *Security Features Operations and Programming Guide*.

Object Code File

An operator can use the MP (Mark Program) system command to assign any of several options to an object code file. Some of these options confer special types of security status on a process, and these options are the following:

- COMPILER. This option marks an object code file with compiler status. The effects of compiler status are described under “Compiler Status” later in this section.
- PU. This option marks an object code file with privileged status. The effects of privileged status are discussed under “Privileged Status” later in this section.
- SECADMIN. This option marks an object code file with security administrator status. The effects of security administrator status are described under “Security Administrator Status” later in this section.
- TASKING. This option marks an object code file with tasking status. The effects of tasking status are described under “Tasking Status” later in this section.

The MP system command also can be used to mark an object code file with granulated privileges when the PU option is undesirable. Each granulated privilege is a subset of privileged status. Available granulated privileges are CHANGE, CHANGESEC, CREATEFILE, EXECUTE, GETSTATUS, GSDIRECTORY, IDC, LOCALCOPY, LOGINSTALL, LOGOTHERS, READ, REMOVE, SETSTATUS, SYSTEMUSER, USERDATA, and WRITE. Their effects are described under “Privileged Status” later in this section.

When an object code file is initiated, the resulting process receives the privileges that were assigned to the object code file. The process can make some of the procedures in the object code file available to other processes by initiating an internal procedure, by initiating a process and passing a procedure parameter, or by becoming a library and exporting procedures. Any of these processes temporarily assumes the privileges assigned to the object code file while it is executing procedures from the object code file.

The following subsections explain how these privileges are propagated to processes from object code files.

Transparent Object Code File Privileges

Most of the options available through the MP (Mark Program) system command have only two states: set or reset. However, the MP command enables you to specify a third state for the PU, SECADMIN, TASKING, and granulated privilege options. This third state is called *transparent*. The following are MP commands and the security categories they assign:

MP Command	**Security Category**
MP <file title> + <granulated privilege>	Granulated privileged
MP <file title> + <granulated privilege> TRANSPARENT	Granulated privileged transparent
MP <file title> + PU	Privileged
MP <file title> + PU TRANSPARENT	Privileged transparent
MP <file title> -PU	Nonprivileged
MP <file title> + SECADMIN	Security administrator
MP <file title> + SECADMIN TRANSPARENT	Security administrator transparent
MP <file title> -SECADMIN	Non-security administrator
MP <file title> + TASKING	Tasking
MP <file title> + TASKING TRANSPARENT	Tasking transparent
MP <file title>-TASKING	Nontasking

Each option can be in only one state at a time: enabled, disabled, or transparent. However, the three options (PU, SECADMIN, and TASKING) do not have to be in the same state. The following command assigns privileged status and security administrator transparent status, and removes tasking status:

```
MP <file title> + PU, + SECADMIN TRANSPARENT, - TASKING
```

Setting any granulated privilege option disables the PU option. Setting the PU option disables all granulated privilege options. The USERDATA option is mutually exclusive of the SECADMIN TRANSPARENT option and the SECADMIN option is mutually exclusive of the USERDATA TRANSPARENT option.

The concept of transparent status is intended primarily for libraries, to enable the actions of a library to be applied with the status of the user program that invokes the library. If a procedure resides in an object code file that has one of these options in the transparent state, then

- If the procedure is initiated, the resulting process is treated as if the option were disabled.
- If the procedure is entered, it inherits the enabled or disabled state of the option of the invoking procedure. Privileged, granulated privilege, security administrator, or tasking status can be inherited through a series of privileged transparent procedures.

For example, if a privileged program initiates a procedure in a privileged transparent library, the procedure is executed as nonprivileged. However, if the privileged program enters the same procedure instead of initiating it, the procedure is executed as privileged.

For information about how privileged transparent status applies to file access rights, refer to Section 19, "Using Shared Files."

Delayed Effects of Object Code File Privileges

When you mark an object code file with special privileges, these privileges do not affect any processes that are already running. The privileges take effect the next time you initiate the object code file.

Copying Privileged Object Code Files

If you copy an object code file marked with privileged, security administrator, or compiler status, the copy retains the same privileges as the original. However, the system administrator can limit the ability to copy or execute such object code files by using the RESTRICT (Set Restrictions) system command. For details, refer to the discussion of the RESTRICT command in the *Security Administration Guide*.

Originating Source

When you initiate a process through a peripheral device, the system records the type of peripheral device in the SOURCEKIND attribute. There is one situation in which the SOURCEKIND value can make an important difference in the capabilities of the process. If the SOURCEKIND value is ODT, the system accords the process ODT status, which is described under "Process Security Classes" in this section.

Additionally, the system records the physical unit number or logical station number (LSN) of the originating peripheral device in the SOURCESTATION task attribute. The value of this attribute allows messages generated by a process to be routed back to the station that originated the process, so that you can easily monitor the progress of your processes.

MARC and CANDE similarly assign the LSN of a session to the STATION task attribute of any tasks (but not jobs) initiated from that session. Refer to Section 9, “Controlling Process I/O Usage” for a discussion of the effects of this attribute.

The system also records the name of the originating station in the SOURCENAME task attribute. The station name can be more stable than the LSN, which often changes after a halt/load or Transaction Server quit.

Process Security Classes

The system software provides a number of security features that you can use to regulate the ability of processes to access other users' files or perform other restricted actions. Processes are classified according to security classes, and each security class allows the process to perform a somewhat different set of restricted actions.

The following subsections describe the capabilities of each of the process security classes and explain how a process can be assigned to a particular class. For further information about any of the security features discussed, refer to the *Security Features Operations and Programming Guide* and the *Security Administration Guide*.

The following are the security classes a user process can belong to:

- Nonprivileged
- Privileged
- Nonusercoded
- Operator display terminal (ODT)
- SYSTEMUSER
- Security administrator
- Compiler
- MCS
- Tasking

A process can belong to more than one of these classes, although certain classes are mutually exclusive. In addition, a process can belong to different security classes at different points in its execution.

Additional security classes exist for operating system processes. For information about system library security and library linkage classes, refer to Section 18, “Using Libraries.”

For a discussion of certain special security issues that arise from the sharing of logical files between processes, refer to Section 19, “Using Shared Files.”

Nonprivileged Status

The default security class for a process is nonprivileged. On a typical system, the vast majority of processes fall into this class. A nonprivileged process can perform any of the following actions:

- Inspect or modify any object within the extended addressing environment of the process. For information about the addressing environment, refer to Section 15, "Using Global Objects," and Section 17, "Using Parameters."
- Create, remove, open, close, read, write, copy, or access the file attributes of data files.
- Initiate, copy, remove, open, close, read, or access the file attributes of object code files.
- Use the nonprivileged form of the GETSTATUS directory call. The nonprivileged form of this call provides information only about directories having the same usercode as the process.
- Use the *VOLUME CHANGE* form of the WFL *VOLUME* statement to affect tape volumes whose FAMILYOWNER value is the same as the usercode of the process.
- Use the WFL ARCHIVE command to back up, roll out, or restore files that have the same usercode as the process.
- Use the MAKEUSER utility to change owner-modifiable attributes of the usercode of the process. A nonprivileged process can change only attributes of its own usercode. Of these attributes, the process can change only those marked with a status of OWNER by a PRIVILEGES segment in the USERDATAFILE.

The ability of a nonprivileged process to access a particular disk file is determined by the values of certain task attributes and file attributes. The following task attributes affect file access rights:

- USERCODE

  The USERCODE value generally grants the process access to files that are stored under the usercode. Certain USERCODE values can also grant special privileges, as discussed under "Privileged Status" and "Nonusercoded Status" later in this section.

- ACCESSCODE

  The ACCESSCODE value can grant the process access to some files that are protected by guard files, as discussed later in this subsection. The process can assign only accesscode/password combinations corresponding to values in the ACCESSCODELIST usercode attribute. Additionally, the process can delete the accesscode value by assigning a null string.

- NAME

  The value of this task attribute can grant the process access to some files that are protected by guard files, as discussed later in this subsection.

- FILEACCESSRULE

  The effects of this task attribute are discussed in Section 19, "Using Shared Files."

- GROUPCODE and SUPPLEMENTARYGRPS

  If a process is not the owner of a file, and one of the values of GROUPCODE or SUPPLEMENTARYGRPS matches the GROUP file attribute of the file, then the file access rights are determined by the group-related subattributes of the SECURITYMODE file attribute.

The process that creates a disk file can assign security-related file attributes to determine which nonprivileged processes can access the file. Thereafter, only privileged processes or processes running with the same usercode as the file can change the values of these security-related file attributes. Following are brief descriptions of the security-related file attributes:

- TITLE

  This file attribute includes the usercode under which the file is stored. For nonusercoded files, an asterisk (*) is included instead of a usercode. Only privileged or nonusercoded processes can create a nonusercoded file.

- SECURITYTYPE

  This file attribute specifies whether a process must have the same usercode as the file in order to access the file. A value of PUBLIC allows any process to access the file. A value of PRIVATE enables nonprivileged processes to access the file only if the processes are running under the same usercode as the file. For nonusercoded files, a value of PRIVATE enables only privileged processes and nonusercoded processes to access the file. A value of GUARDED or CONTROLLED specifies that a guard file is used to determine which nonprivileged processes can access the file.

- SECURITYUSE

  This file attribute specifies whether nonprivileged processes having a usercode different from the file can read from or write to the file. SECURITYUSE does not restrict the ability to initiate an object code file. SECURITYUSE has effect only if the SECURITYTYPE file attribute value is PUBLIC.

- SECURITYGUARD

  For files with a SECURITYTYPE value of GUARDED or CONTROLLED, the SECURITYGUARD file attribute specifies the title of the guard file to be used.

- SECURITYMODE

  This file attribute provides an alternative method of specifying the security restrictions for a file. SECURITYMODE provides functions similar to the SECURITYTYPE and SECURITYUSE attributes. However, SECURITYMODE provides more detailed control. SECURITYMODE specifies

  – Separate file access rights for three classes of users: the owner, group members, and other users. For each class of users, SECURITYMODE can specify any combination of read access, write access, and execution access.

  – Whether a guard file is used, and whether the guard file also applies to the owner of the file.

  - Whether code files should run under the usercode and group code of the initiator of the program, or under the usercode and group code of the code file itself. For more information about this topic, refer to “Real, Saved, and Effective Process Identities” later in this section.

- GROUP

  This file attribute specifies the group to which a file belongs. When determining the rights of a process to access a particular file, if the usercode of the process differs from the owner of the file, the system compares the GROUPCODE and SUPPLEMENTARYGRPS attributes of the process with the SECURITYMODE and GROUP attributes of the file.

- OWNER

  This attribute is read-only and reports the usercode portion of the TITLE file attribute. If TITLE begins with an asterisk (*) instead of a usercode, OWNER returns a null string.

These file attributes are described in detail in the *File Attributes Programming Reference Manual*.

Guard files can be created using the GUARDFILE utility, which is described in the *Security Features Operations and Programming Guide*. A guard file can include detailed information about the types of access allowed to various nonprivileged processes. The guard file can include USERCODE or ACCESSCODE clauses that discriminate between processes on the basis of the corresponding task attributes. The guard file can also include a PROGRAM clause that discriminates between processes on the basis of the NAME task attribute value, and a CODEFILE clause that discriminates between processes on the basis of the code file title.

If a guard file is used, it overrides the value of the SECURITYUSE attribute.

If the tape volume security feature of the Secure Accountability Facility is enabled on the system, then the rights of a nonprivileged process to access a particular tape file are regulated by the task attributes and file attributes listed in the previous discussion as well as by the tape volume attributes FAMILYOWNER, PERMANENTLYOWNED, and MATCHONLYSERIALNO. The tape volume attributes can be assigned only by a privileged user or a privileged process with the WFL *VOLUME* statement. The security administrator can enable tape volume security by using the SECOPT (Security Options) system command to set the security option TAPECHECK to AUTOMATIC. If tape volume security is not enabled, then a nonprivileged process can open a tape file on any tape unit that is not currently in use by another process.

An additional security restriction for disk files is *system file* status. The operating system marks disk files that are part of the acting system software as system files. Examples of system files are the object code file of the current MCP, the job description file, and the current system log. An application process cannot remove or change the title of any system file. Some files have a modified form of system file status. Thus, the USERDATAFILE has system file status and additionally is protected from being read by any application process (only system software can read this file).

Privileged Status

A privileged process has the capabilities of a nonprivileged process, as well as the ability to:

- Access physical files stored under other usercodes, regardless of the SECURITYTYPE, SECURITYUSE, SECURITYGUARD, and SECURITYMODE file attribute values. Note that logical access to a database guarded by a guardfile is not affected by privileged status.
- Use the following WFL statements on files regardless of their usercode:

ADD	COPY	RUN
ALTER	MODIFY	RESTORE
ARCHIVE	MOVE	RESTOREADD
CATALOG	PRINT	SECURITY
CHANGE	REMOVE	START

- Use the following WFL statements:

  VOLUME ADD

  VOLUME DELETE

  VOLUME DESTROYED

  VOLUME OFFSITE

  VOLUME ONSITE

- Create files stored under other usercodes and to create nonusercoded files.
- Set the value of the USERCODE task attribute to a null string.
- Set the GROUP file attribute to any group code, not just the group codes specified by the GROUPCODE task attribute or the SUPPLEMENTARYGRPS task attribute.
- Set the FILEACCESSRULE task attribute to a value of ACTOR.
- Survive most task attribute access errors.
- Use the MAKEUSER utility to change selected usercode attributes. A privileged process can change attributes of any usercode. However, the process can change only those attributes marked with a status of PU by a PRIVILEGES segment in the USERDATAFILE.

Privileged status also grants several other capabilities on systems where the Security Services for ClearPath MCP security administrator feature is not enabled. On systems where the security administrator feature is enabled, these capabilities are wholly or partially reserved for processes with security administrator status. (Refer to “Security Administrator Status” later in this section.) The following are the capabilities:

- The ability to access certain system interfaces, including the DCKEYIN, GETSTATUS, and SETSTATUS functions in DCALGOL.
- The ability to create, modify, and delete usercode definitions in the USERDATAFILE.

Note that the following types of file access are *not* granted by privileged status: the ability to remove or change the titles of most system files, and the ability to write to object code files. Further security restrictions can apply if the privileged process accesses the file through a shared logical file, as discussed in Section 19, “Using Shared Files.”

A process is automatically considered privileged if it is running under a privileged usercode. The usercode of a process is stored in the USERCODE task attribute. An operator can assign privileged status to a usercode by running the MAKEUSER utility or using the MU (Make User) system command. A usercode can also be assigned privileged status by a program that uses the USERDATA function in ALGOL, DCALGOL, or NEWP. For further information about these features, refer to the *Security Administration Guide*.

A process usually inherits the usercode of the session or process that initiated it. A different usercode can be assigned by task attribute assignment, use of the USERDATA function, or use of the WFL *USER* statement. However, in each of these cases, the statement that assigns the usercode must also specify a password, which is checked for validity. Only processes with special privileges can assign a usercode without specifying a password. Message control systems (MCSs) and processes with tasking status use this feature when assigning a usercode to a process initiated by a session.

If a process is not running under a privileged usercode, then the ability of a process to perform a privileged action is determined by the privilege status of the object code file that contains the request.

A process can execute code from several different object code files. This is the case if the process has entered either a library procedure or a passed external procedure. (For an introduction to external procedures, refer to Section 1, “Understanding Basic Tasking Concepts.”) The various object code files might not have the same privilege status. The current privilege status for the process is determined by the privilege status of the object code file containing the procedure that was most recently entered. This procedure contains the code that is currently being executed. For further details about this concept, refer to “Object Code File” earlier in this section.

Note that a privileged program has no special privileges when accessing files on a remote host. For example, suppose a process sets the HOSTNAME attribute of a file to specify a remote host, and then attempts to open that file. This action is executed with privilege on the remote host only if the process usercode is privileged on that host.

Granulated Privileges

Capabilities associated with privileged status can be individually recognized as granulated privileges. When it is undesirable or even dangerous for a process to acquire the full privileged status, granulated privileges can be alternatively used. Given a particular granulated privilege, a process is bounded by the limitation of such privilege. A security administrator can identify the security needs of a user or a program and delegate just the capabilities necessary for the performance of the job (the concept of least privilege). The description for each granulated privilege follows.

Privilege	**Description**
CHANGE	A process with this privilege can change titles of other users' disk files. This includes the file ownership. When a new file name is identical to another user's existing disk file, the file overwrite is not permitted if it is not accompanied by the REMOVE privilege.
CHANGESEC	A process with this privilege can modify security file attributes for files belonging to other users.
CREATEFILE	A process with this privilege can create disk files under another usercode without replacing existing disk files. The privilege does not include file creations through the WFL CHANGE command or through a library maintenance copy operation.
EXECUTE	A process with this privilege can execute other users' code files.
GETSTATUS	A process with this privilege can use the GETSTATUS intrinsic to retrieve information about jobs, tasks, status of peripherals, status of the operating system, and mainframe configuration. The privilege does not include those GETSTATUS directory and volume requests that currently require privileged-user status.
GSDIRECTORY	A process with this privilege can browse other users' private directories and files. In addition, this privilege enables a program to make GETSTATUS directory and volume requests that are typically restricted to a privileged-user status and enables a user to use the FILEDATA TAPEDIR request.
IDC	A process with this privilege can update the current DATACOMINFO file through DATACOMSUPPORT entry points, which are used by the SYSTEM/IDC utility.
LOCALCOPY	A process with this privilege can copy files and directories belonging to other users.This is done on the local host using library maintenance.
LOGINSTALL	A process with this privilege can access the MCSLOGGER intrinsic to create log installation records.
LOGOTHERS	A process with this privilege can access the MCSLOGGER intrinsic to create other log records for which privilege is currently required.
READ	A process with this privilege can have read access to other users' files, regardless of their security attributes.

Privilege	Description
REMOVE	A process with this privilege can remove files belonging to other users. When REMOVE is used with the CREATEFILE, LOCALCOPY, or CHANGE privileges, an existing disk file can be either replaced or removed. A close with purge operation on a non-owned file also requires the process to have the REMOVE privilege.
SETSTATUS	A process with this privilege can use the SETSTATUS intrinsic to control MCP mix, unit, and operational functions. The privilege does not include those SETSTATUS directory and volume requests that currently require privileged-user status.
SYSTEMUSER	A process with this privilege can make GETSTATUS, SETSTATUS, and DCKEYIN requests that are currently restricted to a system user.
USERDATA	A process with this privilege can access the USERDATA intrinsic. This includes all USERDATA functionality available to a privileged user on a system with security administrator status disabled and all USERDATA functionality available to a security administrator on a system with security administrator status enabled.
WRITE	A process with this privilege can have write access to other users' files, regardless of the file's security attributes. Processes with this privilege can also change all modifiable, non-security-related file attributes.

An operator assigns granulated privileges to a usercode by running the MAKEUSER utility or assigns granulated privileges to an object codefile by using the MP (Mark Program) system command. For more information about the MAKEUSER utility, refer to the *Security Administration Guide*. For more information about the MP command, refer to the *System Commands Operations Reference Manual*.

Nonusercoded Status

A nonusercoded process is one whose USERCODE task attribute value is a null string. By default, a process runs without a usercode if you initiate it from a nonusercoded MARC session.

In addition, a process initiated from an ODT is nonusercoded by default unless one of the following conditions is true:

- The ODT has been assigned a terminal usercode by the TERM (Terminal) system command. The terminal usercode is the default usercode for most processes initiated at that ODT.

  However, processes initiated at an ODT by a primitive system command default to a null usercode, even if there is a terminal usercode associated with the ODT. ??COPY (Copy Files) and ??RUN (Run Code File) are two primitive system commands that initiate processes.

- The process is a remote WFL job and the system has a host usercode. Host usercodes are assigned by the HU (Host Usercode) system command.

  Note, however, that a remote WFL job runs nonusercoded if the job is initiated by an *AT <hostname> START* command and the host usercode of the initiating system is defined with SYSTEMUSER status at the host where the job runs.

Processes initiated by a nonusercoded process are, by default, also nonusercoded.

Processes initiated by usercoded processes are, by definition, always usercoded. It is possible for a process to assign a null usercode to a task variable that is not in use, and then initiate a process with that task variable. However, the null usercode value in the task variable is overridden by task attribute inheritance, and the new process runs with the usercode of its initiator.

It is possible for a usercoded process to be assigned a null usercode after initiation. However, only a privileged process can assign a null usercode to an in-use process. Thus, for example, a privileged process can change its own usercode to a null usercode. When the usercode of a privileged process is changed to a null usercode, the process retains its privileged status.

A privileged process can also initiate a task with a nonprivileged usercode, and then change the usercode of the task to a null while the task is running. The task then assumes nonusercoded security status. Processes that are nonusercoded from the time they are first initiated also have nonusercoded security status.

A process with nonusercoded status has the same capabilities as a nonprivileged process, with the following additions:

- The ability to create nonusercoded files; that is, files whose TITLE file attributes begin with an asterisk (*) instead of a usercode, and whose OWNER file attribute is null.
- The ability to initiate a nonusercoded process; that is, a process whose USERCODE task attribute value is a null string.
- The ability to use the UNITNO file attribute, even on a system running with the security option NONPRIVUNITNO set.

Further, certain WFL statements are treated as privileged when submitted by a nonusercoded process. These statements, and other conditions affecting their privilege status, are shown in Table 5–1. This table refers to two concepts not discussed previously:

- Single-statement WFL inputs. These are single WFL statements entered directly at an ODT, entered in CANDE or MARC with the *WFL* prefix, or submitted in array form by a ZIP statement in a program.
- ODT status. This concept is defined under “ODT Status” later in this section.

Table 5–1. WFL Statements Executed with Privilege

WFL Statements	Conditions Granting Privilege
ADD, ARCHIVE, CATALOG, COPY, MOVE, RESTORE	Privileged if the process is nonusercoded
CHANGE, REMOVE, RERUN, SECURITY, START	Privileged if a nonusercoded, single-statement WFL input.
PRINT	Privileged if a nonusercoded, single-statement WFL input that does *not* have ODT status. .
VOLUME	Privileged if either of the following is true: • The process is nonusercoded and has ODT status. • The process has ODT status, only the VOLUME ADD or VOLUME DELETE form of the command is used, and the statement affects only volumes with the same usercode as the process.

ODT Status

A process is said to have *ODT status* if it was initiated from an ODT, or if it is descended from a process initiated from an ODT. The exception to this rule is that processes initiated with the ??RUN (Run Code File) primitive system command do not receive ODT status, nor do the descendants of such processes.

Processes initiated from an ODT frequently run without a usercode and receive nonusercoded status, as discussed under "Nonusercoded Status" earlier in this section.

Regardless of whether it has a usercode, a process with ODT status is granted access to all GETSTATUS calls in DCALGOL. This access includes the privileged form of the GETSTATUS directory call. (The privileged form of this call can return information about directories stored under any usercode.)

Certain WFL statements are treated as privileged when submitted by a process with ODT status. For a list of these statements, and other conditions affecting their privilege status, refer to Table 5-1, "WFL Statements Executed with Privilege.

SYSTEMUSER Status

A process receives SYSTEMUSER status if it is running under a usercode whose SYSTEMUSER usercode attribute is set. SYSTEMUSER status enables a process to use the DCKEYIN, GETSTATUS, and SETSTATUS functions in DCALGOL, even if the process does not have privileged status. A process can use these functions to submit system commands and perform other system operations functions.

By default, SYSTEMUSER status gives access to all the possible DCKEYIN, GETSTATUS, and SETSTATUS calls. However, certain restrictions can apply on a system running the Secure Identification Facility

Security Administrator Status

On a system where the Secure Identification Facility is installed, the system administrator can enable a special security administrator status. If security administrator status is enabled for the system, then certain system commands that would otherwise be available to any privileged or SYSTEMUSER process are instead reserved for use only by processes with security administrator status. The DCKEYIN and SETSTATUS functions corresponding to these system commands are similarly restricted. In addition, the ability to create or alter usercode definitions, which would otherwise be available to any privileged user, is restricted to processes with security administrator status.

The security administrator can also use the RESTRICT command to prevent or limit the use of certain system commands. For information about the RESTRICT command, refer to the *System Commands Operations Reference Manual.*

The system administrator can enable security administrator status on the system by setting the system SECADMIN option. This option is set using the *??SECAD* system command. Once the SECADMIN option is set, a process assumes security administrator status if either of the following conditions are true:

- The process is running with a usercode for which the SECADMIN attribute is set in the USERDATAFILE.
- The process is executing code from an object code file that has been marked with security administrator status. This concept is discussed further under "Object Code File" earlier in this section.

For further information about security administrator capabilities, refer to the *Security Administration Guide*.

Compiler Status

A process with compiler status is allowed to create an object code file or write to an existing object code file. You can mark an object code file with compiler status by using the *MP <file title> + COMPILER* form of the MP (Mark Program) system command. An operator can use this command to mark any program with compiler status, whether or not the program is really a compiler.

If a process without compiler status attempts to write to an object code file that is a permanent file, the write operation is not performed and the process is abnormally terminated. A process without compiler status can write to an object code file that is a temporary file. However, if the process attempts to lock the file, the system changes the file from an object code file into a data file. (For information about the concepts of permanent and temporary files, refer to the *I/O Subsystem Programming Guide*.)

Note that a compiler program has no special privileges when accessing object code files on a remote host. For example, suppose you initiate a compiler and file equate the HOSTNAME attribute of the CODE output file to a remote host. The compiler receives a file attribute error when it attempts to create the object code file. A compiler must create object code files on the host where the compiler is running.

Message Control System Status

Message control systems (MCSs) **Error! Bookmark not defined.**differ from other interactive programs in that they interface directly to the data comm subsystem (rather than opening a remote file) in order to send or receive messages from terminals. This interface is possible because MCSs are written in DCALGOL, an extended version of ALGOL with special data comm capabilities. The system extends a number of special privileges to MCSs.

How an MCS Acquires Its Privileges

The MCS security privileges and MCS priority are not granted to a program simply because it is written in DCALGOL; the system must also recognize the program as an MCS. Two things are necessary for the system to recognize a program as an MCS:

- Each MCS on a system must be named in the data comm network definition for that system. Note that the system disregards the family name portion of the MCS code file title when comparing the title with the data comm network definition. Only one MCS of a given name can be active.
- The MCS must invoke the DCALGOL *DCWRITE* function to initialize its primary queue. Every MCS must have such a queue and must initialize it in order to be recognized as an MCS.

The system removes MCS status from a process if either of the following events occurs:

- The process deactivates its primary queue, either by setting the QACTIVE queue attribute to FALSE or by exiting the block in which the primary queue is declared.
- A data comm quit takes place. A data comm quit can be caused by the *ID: QUIT* form of the ID (Initialize Data Comm) system command.

Priority of an MCS

An MCS automatically runs in the same priority category that control programs run in. This priority category gives the MCS higher priority than WFL jobs and application programs. However, the priority of an MCS is lower than that of any invisible independent runner. The priority of MCSs relative to each other is determined by the PRIORITY task attribute. For an explanation of process priority, refer to Section 7, “Controlling Processor Usage.” For a discussion of how and when this special priority can be inherited by offspring of an MCS, refer to “Inheritance of MCS Status” later in this section.

Privileges of an MCS

When an MCS first initializes its primary queue, the system grants that MCS all the privileges associated with privileged status, as discussed under “Privileged Status” earlier in this section. This is true even if the MCS is running under a nonprivileged usercode and has not been marked as a privileged program.

However, if the MCS changes its usercode, the system reevaluates the privileged status of the MCS. Thereafter, the MCS receives privileged status only if one of the following conditions is true:

- The MCS is running without a usercode.
- The MCS is running under a privileged usercode, and the MCS did not request lesser privileges when changing to that usercode. (Refer to the discussion of USERDATA function 3 later under this heading.)
- The MCS object code file has been marked with privileged status.

Additionally, an MCS receives a number of privileges that are unique to MCSs. These privileges are retained by the MCS regardless of the usercode under which the MCS runs. The following paragraphs describe these unique privileges and features of MCS status.

An MCS is allowed the following privileges with regard to the USERDATA function:

- Ability to use USERDATA function 3 to assume a usercode without supplying the corresponding password.

  When an MCS uses USERDATA function 3 to temporarily assume a usercode, the MCS does not appear in GETSTATUS mix request calls that request mix entries with that usercode. The MCS also does not appear in the output from system commands that display the mix and that request mix entries with that usercode.

  By default, the MCS inherits any of the following types of privileges that are associated with the new usercode: privileged status, security administrator status, and SYSTEMUSER status. However, the MCS can use the USERDATA locator parameter to limit the privileges the MCS can receive from the new usercode. The locator parameter can specify separate limits for each of these types of privileges.

  For an example of a program that uses USERDATA function 3, refer to "Temporarily Assuming an Identity," later in this section.

- Ability to change a user's password with USERDATA function 6, subfunction 1, which is normally disallowed on a system using password generation.
- Ability to call the USERDATA function to validate a usercode/password combination or, optionally, a usercode without the password. This USERDATA function allows the MCS to run with the specified usercode so the MCS can perform a function on behalf of that usercode.
- Ability to call the USERDATA function to validate a usercode/chargecode combination.
- Ability to call the USERDATA function to validate an accesscode/accesscode password combination.
- Ability to call USERDATA function 9 (Privileged Fetch and Examine).
- Ability to specify that the last log-on information for a usercode should be updated as a result of the current USERDATA call.
- Ability to survive USERDATA errors that would normally be fatal. The errors are returned in the USERDATA error result field.

An MCS receives the following special privileges with regard to other restricted DCALGOL functions:

- Ability to call the DCWRITE function, which handles station message traffic.
- Ability to call the MCSLOGGER function, which creates sessions or logs session activity.
- Ability to survive SETSTATUS errors that would otherwise be fatal. The SETSTATUS error reporting mechanism returns the error to the MCS process.
- Trusted status that causes the operating system not to perform validation on any mix numbers specified by the MCS in SETSTATUS calls.

An MCS receives the following special privileges with regard to initiating processes:

- Ability to survive errors in initiating an external object code file, such as security errors, that would otherwise be fatal. Also, the ability to attempt to initiate a missing external code file without becoming suspended with a NO FILE condition. The MCS can determine if initiation was successful by inspecting the task variable used in the process initiation statement. If initiation failed, the STATUS task attribute has a value of BADINITIATE. The reason for the failure is reported in the HISTORYTYPE, HISTORYCAUSE, and HISTORYREASON task attributes.
- Ability to pass a single array parameter to an offspring process by value instead of by reference.
- Trusted status that causes the operating system not to perform validation of any sheet array parameter the MCS passes when initiating a compiler.

An MCS receives the following special privileges with regard to task attribute access:

- Ability to survive task attribute errors that are normally fatal. The MCS can determine whether an error occurred, and the type of error, by interrogating the TASKERROR task attribute of the task variable that was accessed.
- Ability to commit task attribute errors in reading a task attribute without any error messages resulting.
- Ability to assign values to the JOBNUMBER and SOURCESTATION task attributes.
- Ability to modify the NAME task attribute value of a running process.
- Ability to modify the CHARGE task attribute of a running process. Note that the MCS should check the validity of the new CHARGE value before assigning that value.

An MCS has the following special privileges with regard to file access:

- Ability to invoke the exported MCP procedure CHANGESECURITY, which changes the security attributes of files.
- Ability to survive security errors that occur while changing file security.
- Ability to call the exported MCP procedure DRCDETERMINEUSERLIMITS, which reports on the file usage limits imposed on a user by the disk resource control (DRC) system.
- Ability to exceed DRC limits at file open time without any error messages being displayed.
- Ability to set the HSFILECOPY file attribute, which then allows an MCS to modify various "read only" disk file attributes such as CREATIONDATE, CREATIONTIME, and TIMESTAMP.

An MCS has sufficient privilege to access library objects that have a linkage class of 3. For information about library linkage classes, refer to "Security Considerations" in Section 18, "Using Libraries."

Inheritance of MCS Status

In some cases, it can be useful for an MCS to initiate one or more tasks to handle some of its work. By default, these tasks do not receive any special privileges as a result of being initiated by an MCS. These tasks also do not receive the special MCS priority category, and by default they receive a PRIORITY task attribute value of 50 rather than inheriting the PRIORITY value of the MCS.

However, the MCS can grant MCS privileges and priority to any of its offspring by assigning a value of TRUE to the INHERITMCSSTATUS task attribute of the offspring.

Tasking Status

Tasking status grants a process a subset of the privileges associated with MCS status, without the process actually having to be an MCS. Tasking status is well suited to interactive programs that service multiple users and need to be able to assume the identity of those users temporarily.

Tasking status allows programs access to some system software features that must be used with care. For example, if a tasking program calls an MCP intrinsic, the MCP omits some of the checking it would perform for other callers. If the tasking program formats a parameter incorrectly, the error might not be detected immediately and could cause later errors or system dumps. Therefore, you should grant tasking status only to programs that use these privileges properly.

A process receives tasking status when the process is executing code from an object code file that has been marked with tasking status. You can mark object code files with tasking status by using the MP (Mark Program) system command. For information about how processes receive privileges from their object code files, refer to "Object Code File" earlier in this section.

Tasking status provides the same privileges and restrictions as MCS status, with the following exceptions:

- When a process first becomes an MCS, the process automatically receives privileged status in addition to its other MCS privileges. However, when a tasking process is first initiated, tasking status confers privileged status only if the process is nonusercoded. If a tasking process is initiated under a usercode, then privileged status is determined by the privileged status of the usercode and the object code file.

  Once an MCS or tasking process changes its own usercode, the privileged status of the process is affected by the new usercode, as previously discussed under "Privileges of an MCS" in this section.

- Tasking status does not grant access to the DCALGOL functions DCWRITE and SETUPINTERCOM.
- Tasking status does not allow a process to make assignments to the SOURCESTATION task attribute.
- Tasking status does not enable a process to set the HSFILECOPY file attribute to TRUE and then modify certain disk file attributes, such as CREATIONDATE and CREATIONTIME, which are normally read-only.
- Tasking status does not cause the process to run in the MCS priority category.
- Multiple instances of the same tasking program can be running at the same time.
- The INHERITMCSSTATUS task attribute does not cause tasking status to be inherited.

Process Identity and Process Initiation

When a process initiates an external task, the NAME task attribute of the task specifies the title of the object code file to be initiated. The USERCODE and FAMILY attributes of the task have no effect on the search for the object code file; instead, the USERCODE and the FAMILY of the initiating process affect the search for the object code file. For example, consider the following WFL job:

```
BEGIN JOB;
  USERCODE = DADA/DALI;
  FAMILY DISK = MYPACK OTHERWISE YOURPACK;

  RUN OBJECT/EPSILON;
      USERCODE = RENE/MAX;
      FAMILY DISK = AMAST OTHERWISE BMAST;
END JOB
```

In this job, the phrase *RUN OBJECT/EPSILON* implicitly assigns "OBJECT/EPSILON" to the NAME attribute of the task. When executing this RUN statement, the system uses the USERCODE and FAMILY of the job, not those of the task. Therefore, the system searches for OBJECT/EPSILON in the following locations, in the order listed:

```
(DADA)OBJECT/EPSILON ON MYPACK;
*OBJECT/EPSILON ON MYPACK;
(DADA)OBJECT/EPSILON ON YOURPACK;
*OBJECT/EPSILON ON YOURPACK;
```

However, once OBJECT/EPSILON is initiated, it runs under usercode RENE. When OBJECT/EPSILON attempts to access files, the system applies the value FAMILY DISK = AMAST OTHERWISE BMAST to the file search.

Temporarily Assuming an Identity

Some interactive programs, such as MCSs, temporarily assume the identity of users when performing actions on behalf of those users. To effectively behave with a particular user's identity, the interactive program might need to do more than assume a usercode. For example, if an interactive program performs actions on files, the interactive program might need to assume the FAMILY statement of the user in order to find the correct files. A program can achieve this effect indirectly by taking the following steps:

- Temporarily assuming the usercode of the user by calling USERDATA function 3 (Validate Usercode/Password). To use this call without supplying a password, the process must have MCS status or tasking status.

  To assume a usercode without specifying a password, the process must pass the usercode to USERDATA function 3 in standard form and set bits [1:2] of the qualification byte to 0. To assume a usercode when the password is also supplied, the process can pass the usercode/password combination to USERDATA function 3 in either display form or standard form.

To make it possible to interrogate the FAMILY attribute, the process must also set bit 5 of the ACTION parameter when making the USERDATA function 3 call. Bit 5 causes USERDATA to return a copy of the USERDATA entry in the data output parameter.

- Retrieving the FAMILY value of the user by calling USERDATA function 1 (Examine User Entry), with the data input parameter set to the USERDATA entry previously returned, and the locator parameter set to USERDATALOCATOR ("FAMILY"). This call returns the FAMILY attribute of the usercode in the data output parameter.
- Assigning the value from the data output parameter to the FAMILY attribute of MYSELF.

The following DCALGOL example demonstrates these steps.

```
$INSTALLATION 1
BEGIN
  BOOLEAN BRSLT;
  REAL RSLT;
  ARRAY UDENTRY[0:255];
  ARRAY UCSTANDARD[0:14];
  EBCDIC ARRAY UCDISPLAY[0:2],
               FAM[0:89];
  REPLACE UCDISPLAY BY "JSMITH.";
  BRSLT:= DISPLAYTOSTANDARD (UCDISPLAY, UCSTANDARD);
  UCSTANDARD[0].[33:2]:= 0; % Indicates no password is supplied

  % Assume other usercode
  RSLT:= USERDATA(     3  % Function 3:
                             %     Validate Usercode/Password
                 & 1[5:1]  % Copy USERDATA entry to UDENTRY
                 & 1[6:1], % Use standard form
                   MYSELF,
                        7, % Assume privileges of the new usercode
                  UDENTRY, % Returns copy of USERDATA entry
               UCSTANDARD);
  IF BOOLEAN(RSLT) THEN
     DISPLAY("USERDATA ERROR:" CAT STRING(RSLT.[07:07],*));

  % Interrogate FAMILY of usercode
  RSLT:= USERDATA(     1,   % Function 1:
                              %     Examine User-Provided Entry
                        ,   % No task variable
  USERDATALOCATOR("FAMILY"),   % Fetch FAMILY attribute
                      FAM,   % Return FAMILY value in this array
                  UDENTRY);  % Search this USERDATA entry
  IF BOOLEAN(RSLT) THEN
     DISPLAY("USERDATA ERROR:" CAT STRING(RSLT.[07:07],*));

  % Assume new FAMILY value
  REPLACE MYSELF.FAMILY BY FAM;
END.
```

For the preceding example to work, the object code file must be marked with tasking status.

The $INSTALLATION 1 option in the example is necessary in order to give the DCALGOL compiler access to the DISPLAYTOSTANDARD function.

To retrieve the values of usercode attributes other than FAMILY, you could add further Function 1 calls, with different values specified for the third parameter. For example, to interrogate the IDENTITY usercode attribute, you could set the third parameter to USERDATALOCATOR ("IDENTITY").

Real, Saved, and Effective Process Identities

You can use the SECURITYMODE file attribute to specify that a code file should run under the usercode or group code of the code file itself, instead of the usercode or group code received from the initiator. Once the code file is initiated, it can freely toggle back and forth between the usercode and group code of the code file and the usercode and group code of the initiator.

Using SETUSERCODE and SETGROUPCODE

The usercode of a code file is the usercode located at the start of the TITLE file attribute. This usercode is also stored in the OWNER file attribute. To cause a code file to run under the usercode of the code file, you can assign the SETUSERCODE subattribute of the SECURITYMODE file attribute. The following ALGOL statement makes such an assignment to file F:

```
F.SETUSERCODE := TRUE;
```

The group code of a code file is the value specified by the GROUP file attribute. To cause a code file to run under the group code of the code file, you can set the SETGROUPCODE subattribute of the SECURITYMODE file attribute. The following ALGOL statement makes such an assignment to file F:

```
F.SETGROUPCODE := TRUE;
```

Understanding Real and Saved Identities

The system uses multiple task attributes to distinguish between the identity a process receives from its code file and the identity it receives from its initiator. These task attributes are as follows:

- REALUSERCODE

  This attribute records the USERCODE attribute value the process received from its initiator, either through inheritance or task equation.

- SAVEDUSERCODE

  This attribute records the USERCODE attribute value the process received from its code file, if SETUSERCODE was set for the code file. If SETUSERCODE was not set, then SAVEDUSERCODE is a copy of REALUSERCODE.

- USERCODE

  This attribute records the effective usercode that is currently in use by the process. When the process is first initiated, this value is the same as SAVEDUSERCODE. Thereafter, the process could change its USERCODE to different values, either through assignments to the USERCODE attribute, or through the method described in the following subsection “Toggling Between Identities.”

- REALGROUPCODE

  This attribute records the GROUPCODE attribute value the process received from its initiator, either through inheritance or task equation.

- SAVEDGROUPCODE

  This attribute records the GROUPCODE attribute value the process received from its code file, if SETGROUPCODE was set for the code file. If SETGROUPCODE was not set, then SAVEDGROUPCODE is a copy of REALGROUPCODE.

- GROUPCODE

  This attribute records the effective usercode that is currently in use by the process. When the process is first initiated, this value is the same as SAVEDGROUPCODE. Thereafter, the process could change its GROUPCODE to different values through the method described in the following subsection “Toggling Between Identities.”

Toggling between Identities

A process can change its current USERCODE to either of those stored in the REALUSERCODE or SAVEDUSERCODE attribute, without having to specify a password. However, the process cannot do this toggling with ordinary task attribute assignments. Instead, the process must use certain specialized functions that are available in three forms:

- As ALGOL procedures in the file SYMBOL/POSIX/ALGOL/PROPERTIES. These procedures are documented in the *ALGOL and MCP Interfaces to POSIX Features Programming Reference Manual*.

- As C language functions accessible through a header file. These functions are documented in the *C Programming Reference Manual, Volume 2: Headers and Functions.*
- As library procedures exported by the MCPSUPPORT library. These procedures are documented in the *ALGOL and MCP Interfaces to POSIX Features Programming Reference Manual*.

These functions and procedures refer to user identities by numeric values that correspond to the UID usercode attribute. Therefore, changing to the REALUSERCODE or SAVEDUSERCODE is a two-step process:

1. Retrieve the user ID associated with the REALUSERCODE or SAVEDUSERCODE.
2. Change the process's current user ID to the user ID that was retrieved.

Similarly, a process can change its current GROUPCODE to either of those stored in the REALGROUPCODE or SAVEDGROUPCODE attribute. The process uses procedures or functions that are available from the SYMBOL/POSIX/ALGOL/PROPERTIES file, the C header file, or the MCPSUPPORT library.

The following table lists the procedures and functions used to toggle between the real and saved usercode or group code:

Action to Perform	**ALGOL Procedure**	**C Function**	**MCPSUPPORT Procedure**
Retrieve user ID of REALUSERCODE	GETUID	getuid()	POSIX_INTEGERIDS
Retrieve user ID of USERCODE	GETEUID	geteuid()	POSIX_INTEGERIDS
Change current user ID	SETUID	setuid()	POSIX_SETIDS
Retrieve group ID of REALGROUPCODE	GETGID	getgid()	POSIX_INTEGERIDS
Retrieve group ID of GROUPCODE	GETEGID	getegid()	POSIX_INTEGERIDS
Change current group ID	SETGID	setgid()	POSIX_SETIDS

The following ALGOL program uses procedures from SYMBOL/POSIX/ALGOL/PROPERTIES to change its usercode and group code between the "real" and "saved" values:

```
BEGIN
$INCLUDE "SYMBOL/POSIX/ALGOL/PROPERTIES."
INTEGER ERRNO,
        REAL_GID,
        REAL_UID,
        RSLT,
        SAVED_GID,
        SAVED_UID;

% RECORD SAVED USERCODE AND GROUPCODE
% Because process has not yet changed its identity, these values
% are the same as the effective usercode and group code.
SAVED_UID := GETEUID(ERRNO);
SAVED_GID := GETEGID(ERRNO);

% RECORD REAL USERCODE AND GROUPCODE
REAL_UID := GETUID(ERRNO);
REAL_GID := GETGID(ERRNO);

% CHANGE USERCODE TO REALUSERCODE
RSLT := SETUID(REAL_UID, ERRNO);

% CHANGE GROUPCODE TO REALGROUPCODE
RSLT := SETGID(REAL_GID, ERRNO);

% CHANGE USERCODE TO SAVEDUSERCODE
RSLT := SETUID(SAVED_UID, ERRNO);

% CHANGE GROUPCODE TO SAVEDGROUPCODE
RSLT := SETGID(SAVED_GID, ERRNO);

END.
```

Note that the system performs security checking on the SETUID function. The system permits the SETUID to succeed only if the requested user ID corresponds to the USERCODE, REALUSERCODE, or SAVEDUSERCODE of the process.

Similarly, the system performs checking on the SETGID function. The system permits the SETGID to succeed only if the requested group ID corresponds to the value of the GROUPCODE, REALGROUPCODE, or SAVEDGROUPCODE of the process.

Section 6
Monitoring and Controlling Process Status

During its lifetime, a process can pass through several distinct states. These states characterize whether the process is currently executing, and if not, why not. You can design programs to monitor and modify process states through the use of task attributes and related expressions. You can also monitor process status through the use of system commands.

Understanding Process Status

From the time a process is initiated until it terminates, it is considered an *in-use* process. An in-use process can pass through several process states. These process states indicate whether the process is currently executing, and if not, why not.

When a process is not in use, the task variable for that process stores any of several process states. These process states specify if the process has not been initiated, if initiation failed, or if the process has terminated. It is possible for the task variable to store this information because the task variable of a process exists before the process is initiated and continues to exist after the process terminates. In the following WFL example, task variable T is created when the system executes the declaration TASK T, and continues to exist until the system executes the END JOB statement:

```
?BEGIN JOB;
TASK T;
RUN OBJECT/PROG [T];
.
.
.
?END JOB
```

Information about process status is available through several mechanisms, including the STATUS task attribute, the task state expression in WFL, mix display commands, and the STACK STATE line of the Y (Status Interrogate) system command output. Each of these programming constructs and system commands uses a slightly different terminology to portray process status. Table 6–1 shows the possible values of the STATUS task attribute and the corresponding status values returned by the other process monitoring methods. The meanings of the various process states are discussed in the subsections following the table.

Table 6–1. Process States

STATUS Task Attribute	WFL Task State	Mix Display Commands	STACK STATE in Y Display
NEVERUSED	None	None	None
SCHEDULED	Both of these: • SCHEDULED • INUSE	Both of these: • S (Scheduled Mix Entries) • MX (Mix Entries)	One of these: • Scheduled • Selected
ACTIVE	Both of these: • ACTIVE • INUSE	All of these: • A (Active Mix Entries) • J (Job and Task Display) • MX (Mix Entries) • DBS (Database Stack Entries)	One of these: • Alive • Holding • Ready • To be continued • Waiting on an event
SUSPENDED	Both of these: • STOPPED • INUSE	Both of these: • W (Waiting Mix Entries) • MX (Mix Entries)	Waiting on an event
FROZEN	Both of these: • ACTIVE • INUSE	LIBS (Library Task Entries)	Frozen
GOINGAWAY	None	None	None
BADINITIATE	ABORTED	None	None
TERMINATED	One or more of these: • COMPLETED • COMPLETEDOK • COMPILEDOK • ABORTED	C (Completed Mix Entries)	None

Note: *GOINGAWAY is a write-only value. That is, assigning GOINGAWAY actually changes the STATUS value to ACTIVE, with additional effects that are described under "Thawing a Library" later in this section.*

STATUS Task Attribute

The following are explanations of the STATUS task attribute values shown in Table 6–1.

- NEVERUSED

  The task variable being interrogated has never been used in a process initiation statement, or has been reinitialized since it was last used. Refer to "Preparing a Task Variable for Reuse" later in this section.

- SCHEDULED

  A process initiation statement has been executed, but the system is delaying initiation of the process. For further information, refer to "Preventing Process Scheduling" later in this section.

- ACTIVE

  Process execution is proceeding normally. In the Y command output, this status is expressed through any of several more specific values, which are described under "Y (Status Interrogate) Stack States" later in this section.

- SUSPENDED

  The process is waiting on an event that might require operator intervention. For example, the process might be trying to open a tape file, and the operator might need to mount the appropriate tape on a drive. Or the process might have executed an ACCEPT statement, which requires a response from an operator. For further information, refer to the discussion of responding to waiting entries in the *System Operations Guide*.

- FROZEN

  The process is a frozen server library process. It might be a permanent or temporary library. For information about library processes, refer to Section 18, "Using Libraries."

- GOINGAWAY

  You can assign this value to a frozen server library process to cause it to resume execution as soon as possible. For details, refer to "Thawing a Library" later in this section.

- BADINITIATE

  An unsuccessful attempt was made to initiate the process. Process initiation can fail, for example, if the specified object code file is missing or if the object code file expects different parameters than are passed by the initiator. Note that BADINITIATE is only one of the terminations that can cause the WFL task state of ABORTED to return a value of TRUE. See the following description of TERMINATED.

- TERMINATED

  The process initiated successfully and later terminated. You cannot tell from the STATUS value whether the process completed normally or whether some circumstance caused the process to fail. For further information about process terminations, refer to "WFL Task State Expression" later in this section and to Section 10, "Determining Process History."

WFL Task State Expression

The task state expression in WFL returns a Boolean value indicating whether a process is in a specified state. For example, the following statement fragment takes a specified action if the process with task variable T1 has completed execution.

```
IF T1 IS COMPLETED THEN...
```

Some of the task states that can be queried correspond to single STATUS task attribute values. Other task states correspond to two or more STATUS task attribute values. Thus, at any given time, it is possible that more than one of the possible task state expressions will return a value of TRUE. The following are values that can be used in a WFL task state expression, and the conditions that cause them to evaluate to TRUE:

- SCHEDULED

  The system is delaying initiation of the process. The STATUS task attribute value is SCHEDULED.

- ACTIVE

  The process is executing normally. The STATUS task attribute value is ACTIVE.

- STOPPED

  The process is waiting on an event that might require operator action. The STATUS task attribute value is SUSPENDED.

- INUSE

  The process is in use; that is, it has been initiated but has not yet terminated. The STATUS task attribute value is SCHEDULED, ACTIVE, or SUSPENDED.

- COMPLETED

  The process terminated. The STATUS task attribute value is TERMINATED or BADINITIATE.

- COMPLETEDOK

  The process completed execution normally, but if it was a compilation, it might not have compiled the program successfully. The STATUS task attribute value is TERMINATED and the HISTORYTYPE task attribute value is NORMALEOTV or SYNTAXERRORV.

- COMPILEDOK

  The process completed execution normally. If the process was a compilation, it compiled the program successfully. The STATUS task attribute value is TERMINATED and the HISTORYTYPE task attribute value is NORMALEOTV.

- ABORTED

  The process terminated abnormally, for example, because of a fault or because of operator entry of a DS (Discontinue) system command. The STATUS task attribute value is TERMINATED or BADINITIATE, and the HISTORYTYPE task attribute value is DSEDV.

The COMPLETEDOK, COMPILEDOK, and ABORTED values give you the ability to determine whether a process completed successfully. For further information on determining how and why a process terminated, refer to Section 10, "Determining Process History."

Mix Display Commands

Several different system commands are available for displaying all the processes that are in a particular state. The following are the system mix commands and the process states they display:

- A (Active Mix Entries)

  Displays processes that are running normally. The STATUS task attribute value is ACTIVE.

- C (Completed Mix Entries)

  Displays processes that have recently terminated. The STATUS task attribute value is TERMINATED or BADINITIATE.

- DBS (Database Stack Entries)

  Displays all active database stacks.

- J (Job and Task Display)

  Displays all in-use processes. The STATUS task attribute value is SCHEDULED, ACTIVE, or SUSPENDED.

- LIBS (Library Task Entries)

  Displays frozen library processes. The STATUS task attribute value is FROZEN.

- MX (Mix Entries)

  Displays all in-use processes. The STATUS task attribute value is SCHEDULED, ACTIVE, or SUSPENDED.

- S (Scheduled Mix Entries)

  Displays processes that the system is delaying initiating. The STATUS task attribute value is SCHEDULED.

- W (Waiting Mix Entries)

  Displays processes that are waiting on an event that might require operator intervention. The STATUS task attribute value is SUSPENDED.

In addition to being available through individual system commands, these displays are available as part of the ADM (Automatic Display Mode) output. For information about using ADM to track processes, refer to the *System Operations Guide*.

Y (Status Interrogate) Stack States

The Y (Status Interrogate) system command returns several types of information about a process, including the mix number, usercode, program name, and stack state. In the following example, the stack state is WAITING ON AN EVENT:

```
Status of Task 3251\4441 AT 16:15:28
Program name: *OBJECT/ED ON DOCPK
Priority: 50
Origination: SA154/CANDE/3 (LSN 288)
MCS: SYSTEM/CANDE
Usercode: JASMITH
Chargecode: 6825
Stack State: Waiting on an event
```

If the process is a library, then the Y display also includes a list of the processes linked to the library. Refer to "Determining Which Users Are Linked to a Library" in Section 18, "Using Libraries."

The following are explanations of the STACK STATE values shown in Table 6–1:

- Alive

  The process is currently bound to a processor. That is, it is actually being processed rather than being in any type of waiting state. You can see this state displayed only on a multiprocessor system, because on a single processor system, the CONTROLLER independent runner has to take over the only processor to execute the Y command.

- Frozen

  The process is a frozen library.

- Holding

  The process is waiting on interrupts. This type of waiting is described in Section 16, "Using Events and Interlocks."

- Ready

  The process is in the ready queue and will proceed as soon as a processor is available. It is not unusual for a process to be in this state, because each central processor on the system can be executing only one process at a time, and the mix can contain many processes. The priority of a process can affect the amount of time it spends in the ready queue; refer to Section 7, "Controlling Processor Usage."

- Scheduled

  The system is delaying initiation of the process for any of various reasons. For information about process scheduling, refer to "Preventing Process Scheduling" later in this section.

- Selected

  The process is being initiated.

- To be continued

  This value indicates a process that has initiated a synchronous task that has not yet completed, or a process that has executed a CONTINUE statement and is waiting on its coroutine.

- Waiting on an event

  If no RSVP line appears in the Y command output, then this value means the process is waiting for an I/O operation to be completed or for a particular event to be caused. For a general discussion of events, refer to Section 16, "Using Events and Interlocks."

  If an RSVP line appears in the Y command output, then the WAITING ON AN EVENT value means that the process is waiting on an event that might require operator intervention. The RSVP line lists system commands that might be helpful responses to the situation. For more information about responding to waiting entries, refer to the *System Operations Guide*.

If the process is taking a program dump, executing a SORT procedure, or scrubbing disk areas, then the Stack State value includes a second component that states *Programdumping*, *Sorting*, *scrubbing disk areas*, or some combination of them. The following example shows the output for a process that is taking a program dump:

```
Status of Task 6408/6513 at 16:39:09
Program name: *OBJECT/ED ON SYS00
Priority: 50
Origination: OCDWH_1/CANDE/2  (LSN 295)
MCS:SYSTEM/CANDE
Usercode: DEBS
Chargecode: 6893
Stack State: Waiting on an event, Programdumping
Display: ED:OPERATOR DSED @ (47312200)
```

Monitoring Changes in Process Status

A process can monitor the status of its offspring by waiting on its own EXCEPTIONEVENT task attribute. This method works because, when the status of a process changes, the system automatically causes the EXCEPTIONEVENT of the parent of the process, unless the EXCEPTIONTASK of the offspring has been changed.

If you design a process to wait on its own EXCEPTIONEVENT, it can resume execution and check the status of its offspring each time the EXCEPTIONEVENT is caused. For example, to wait for a task to terminate, the parent process can execute the following ALGOL statement:

```
WHILE T.STATUS GTR VALUE(TERMINATED) DO
   WAITANDRESET(MYSELF.EXCEPTIONEVENT);
```

Note that a WAITANDRESET statement is used rather than a simple WAIT statement. If a simple WAIT statement were used, then the WHILE loop would execute an infinite number of times after the first time the EXCEPTIONEVENT was caused.

The most typical reason for using such WAIT statements is to prevent critical block exits for ALGOL or COBOL programs that initiate asynchronous tasks. Critical block exits are discussed in Section 2, "Understanding Interprocess Relationships."

It is not necessary to take steps to prevent critical block exits in WFL. WFL automatically waits at the end of each block if any processes initiated by statements in the block are still in use. However, there can be other reasons for a WFL job to wait on the termination of an asynchronous task. For example, suppose you have an application consisting of three programs. The first two programs create files that are used as input by the third program. The following WFL job runs the first two programs in parallel, and waits for them to complete before initiating the third program:

```
100 ?BEGIN JOB;
110   TASK T1, T2, T3;
120   PROCESS RUN OBJECT/RUNEX [T1];
130   PROCESS RUN OBJECT/TADCOM [T2];
140   WHILE T1 ISNT COMPLETED OR T2 ISNT COMPLETED DO
150     WAIT;
160   PROCESS RUN OBJECT/DIALUP [T3];
170 ?END JOB
```

Note that the statement at line 150 is simply WAIT, rather than WAITANDRESET (MYSELF.EXCEPTIONEVENT) as it would be in ALGOL. This difference arises because WFL has no syntax for directly accessing the EXCEPTIONEVENT task attribute, or events in general for that matter. However, the simple WAIT in WFL has the effect of implicitly waiting on and resetting the EXCEPTIONEVENT.

WFL provides some other useful expressions for monitoring process status. You can design a WFL job to wait for a task to terminate, to wait for the task to assume a particular status, or to wait for any attribute of the task to assume a desired value. For details, refer to the discussion of the WAIT statement in the *Work Flow Language (WFL) Programming Reference Manual*.

Controlling Process Status

You can accomplish some changes to process status through programmatic assignments to the STATUS task attribute. Additionally, you can prevent many instances of process scheduling or suspension through careful program design.

Terminating a Process

You can interactively terminate a process by entering the DS (Discontinue) system command. You can design a program to terminate a process by assigning the STATUS task attribute a value of TERMINATED. The following is a WFL program that terminates a task if it becomes suspended:

```
100 ?BEGIN JOB ACCOUNTS/JOB;
110   JOBSUMMARY = SUPPRESSED;
120   CLASS = 2;
130 TASK T;
135 BOOLEAN DONE;
140 PROCESS RUN OBJECT/DAILY/ACCOUNTS [T];
150 WHILE NOT DONE DO
160 BEGIN
170   WAIT;
180   IF T IS STOPPED THEN
190      BEGIN
200        T(STATUS = TERMINATED);
210        MYSELF(JOBSUMMARY = UNCONDITIONAL);
220      END;
230   IF T IS COMPLETED THEN
240      DONE:= TRUE;
250 END;
260 ?END JOB
```

The presumption behind this WFL program is that OBJECT/DAILY/ACCOUNTS is a program that does not normally become suspended at any point in its run. If this particular program becomes suspended, it means that something has gone wrong and it is something that an operator cannot easily fix. Further, it is assumed that job queue 2, which this job is initiated from, has a mix limit of 1. Thus, if OBJECT/DAILY/ACCOUNTS becomes suspended, it is impossible for any more WFL jobs to be initiated from that job queue until an operator notices the situation and discontinues the process.

The WHILE statement at lines 150 to 250 is included to prevent this job from ever uselessly blocking up the job queue. Within the WHILE statement, the WAIT statement at line 170 causes the WFL job to wait until its own EXCEPTIONEVENT is caused. The system automatically causes the job's EXCEPTIONEVENT when the STATUS value of any of the job's offspring changes. When the status value of the offspring changes, the statement at line 180 uses the task state expression to determine if OBJECT/DAILY/ACCOUNTS is suspended; if so, then the statement at line 200 assigns a STATUS of TERMINATED to discontinue OBJECT/DAILY/ACCOUNTS. The statement at line 210 causes printing of the job summary. For information about job summaries, refer to Section 10, “Determining Process History.”

The statement at lines 230 to 240 causes the loop to be exited when OBJECT/DAILY/ACCOUNTS terminates (whether it terminated normally or was discontinued).

Thawing a Library

Thawing a library is the act of changing a permanent library into a temporary library. A temporary library automatically resumes execution as soon as it has no more users. By contrast, a permanent library remains frozen indefinitely. Thawing a permanent library is thus a first step toward removing the library process from usage (for example, because you want a newer version of the library program to be used). Thawing the library is less drastic than discontinuing the library process with a DS (Discontinue) system command or a STATUS assignment of TERMINATED.

You can design a program to thaw a library process by assigning either of two values to the STATUS task attribute: ACTIVE or GOINGAWAY. Table 6–2 summarizes the differences in the effects of these two assignments.

Table 6–2. Effects of GOINGAWAY and ACTIVE Assignments

Effects	GOINGAWAY Assignment	ACTIVE Assignment
Time Execution Resumes	When there are no more users	When there are no more users
New Users of Shared Libraries	Are linked to a new invocation of the latest version of the library object code file	Are linked to the existing library process
STATUS Task Attribute	ACTIVE	FROZEN
WFL Task State	ACTIVE, INUSE	ACTIVE, INUSE
Mix Commands	A, J, MX	LIBS
Y Stack State	WAITING ON AN EVENT	FROZEN

The key difference between GOINGAWAY and ACTIVE assignments is that the GOINGAWAY assignment prevents any additional user processes from linking to the library process. For libraries that are shared by many users, this can make a big difference in how long the library process takes to resume execution. If you use an assignment of ACTIVE instead, new processes can continue linking to the (newly temporary) library, with the result that the library remains frozen indefinitely.

Note that neither the GOINGAWAY assignment nor the ACTIVE assignment actually changes the STATUS task attribute to the requested value. After a GOINGAWAY assignment, the STATUS value is ACTIVE. GOINGAWAY is therefore never returned as a value when a process reads the STATUS task attribute. By contrast, after an ACTIVE assignment, the STATUS remains FROZEN until there are no more users of the library. Then the library STATUS changes to ACTIVE and the process resumes execution.

You can thaw a library interactively with the THAW (Thaw Frozen Library) system command. This command has the same effect as assigning a value of ACTIVE to the STATUS task attribute.

Suspending and Resuming Processes

You can interactively suspend execution of a process with the ST (Stop) system command, and resume execution of the process with an OK (Reactivate) system command. You can design a program to achieve the same result by assigning the STATUS task attribute a value of SUSPENDED or ACTIVE. However, note that if the process is suspended by the system, the OK command or ACTIVE assignment frequently is not enough to resolve the cause of the suspension. In this case, the process is suspended again by the system without progressing any further in its execution.

One reason to suspend a process programmatically is for testing. For example, you can add a statement in a program that causes it to be suspended at a certain point where a problem has been occurring. Then you can force a memory dump or program dump through the DUMP (Dump Memory) system command, with the knowledge that the dump will reflect the state of the process at a selected point in its execution.

Parallel processes can also use assignments of SUSPENDED or ACTIVE as a means of coordinating their activities. However, the system provides special variables called *events* that are better suited to coordinating parallel processes. For details, refer to Section 16, "Using Events and Interlocks."

Preparing a Task Variable for Reuse

As was stated earlier in this section, a task variable can be in use by only one process at a time. However, it is possible to reuse a task variable so long as the first process terminates before the task variable is used in another process initiation statement. The side effects that can result from such reuse of task variables are discussed in the *Task Attributes Programming Reference Manual*.

Suffice it to say here that the side effects involve task attribute values that are retained from one use of the task variable to the next. To restore all the task attributes of the task variable to their default values, you can assign the STATUS task attribute a value of NEVERUSED. In WFL, you also have the option of using an INITIALIZE statement, which has the same effect as the STATUS assignment.

Preventing Process Scheduling

A process is SCHEDULED when it has been submitted for initiation, but the system is delaying initiation of the process. Scheduling can have any of several causes, the most common of which is a lack of available memory on the system. If the system estimates that a particular process will require more memory for efficient execution than is currently available, the system places the process in a scheduled state until more memory becomes available.

Two methods that help prevent a process from being scheduled because of a shortage of available memory are:

- Override the system's memory estimate for the process through assignments to the CORE and STACKSIZE task attributes. For details, refer to Section 8, “Controlling Process Memory Usage.”
- Assign the process with control program status by marking its object code file with the *MP <file title> + CONTROL* form of the MP (Mark Program) system command. For information about control program status, refer to Section 7, “Controlling Processor Usage.”

For further information about the causes of scheduling, refer to the process scheduling discussion in the *System Administration Guide*.

Preventing Process Suspension

Many cases where a process becomes suspended by the system can be prevented with a little planning. To be sure, there are situations that cannot be anticipated that might make it necessary for the system to suspend a process. One such example is an extreme shortage of available memory. However, a good many cases of process suspension result from such causes as failed attempts to open files or ACCEPT statements that require immediate input from the operator.

In many of these cases, it might be preferable to allow the process to become suspended. The advantage to this is that the process appears in the W (Waiting Mix Entries) display with a message explaining why it is suspended. This is desirable if the situation is one that an operator can easily remedy. A common example of such a situation is one where a process is attempting to open a tape file. When the process appears in the W display, the operator is prompted to mount the appropriate tape.

However, if you are interested in automating operations at your site as much as possible, then you might find the techniques discussed in the following subsections to be useful. For further information about the file attributes mentioned in the following discussions, refer to the *File Attributes Programming Reference Manual*.

Checking File Residence

You can design a program to read the AVAILABLE or RESIDENT attributes of a file before attempting to open the file. RESIDENT returns a value of TRUE or FALSE to indicate whether the file is available. AVAILABLE returns a numeric value indicating whether the file can be opened, and if not, why not.

If the file is available, the program can execute an OPEN statement. If the file is not available, the program can skip the OPEN statement and take whatever recovery actions are deemed appropriate by the programmer.

Using AUTORESTORE for Disk Files

You can use the AUTORESTORE task attribute to request that the system automatically attempt to restore any missing disk file requested by a process. Automatic restoration can prevent the process from becoming suspended with a NO FILE condition. Refer to the discussion of disk file usage in Section 9, "Controlling Process I/O Usage."

Identifying Tape Files

When a process opens a tape file, the process can become suspended even if the requested tape is already mounted on an available tape drive. The suspension occurs if the process does not give the system sufficient information to identify the particular tape to search for the file. If the process becomes suspended, the operator can use system commands such as IL (Ignore Label) or OU (Output Unit) to specify the correct tape drive so process execution can resume.

Nothing you can do removes the need for someone to mount a tape containing the tape file on a tape drive. However, you can ensure that an operator does not have to take any further action beyond mounting the tape and later removing it.

You can use the SERIALNO attribute to specify the particular tape that should be opened. When the process attempts to open the file, the system checks to see whether a tape with that SERIALNO value is mounted on any of the available drives. If the tape is mounted, then the system looks for the requested file on that tape, without ever suspending the process.

SERIALNO is also available as an option in the WFL *COPY* statement. The following is an example:

```
COPY (JASMITH)= FROM SYSPK(PACK) TO LABCON(TAPE,SERIALNO="LABIN");
```

This example creates a tape named LABCON with a SERIALNO value of *LABIN*. The SERIALNO value can include letters as well as digits. Any letters in the string must be capitalized.

Sometimes you want the program to write output to a tape, but you do not really care which tape, as long as it goes to a tape that is not otherwise in use. In this case, you can leave the SERIALNO value empty and set the MYUSE file attribute to OUT. If the SERIALNUMBER operating system option is not set, then the system writes the file to any scratch tape that is mounted and not in use. If you specify the SCRATCHPOOL or DENSITY file attributes, the system restricts the selection to tapes with the requested scratch pool or density.

An operator can set or reset the SERIALNUMBER option with the OP (Options) system command.

Using UNITNO and OMITTEDEOF for Unlabeled Tape Files

By default, any tapes created by an A Series system have ANSI-standard tape labels. These tape labels store identification information for the tape. However, you might have occasion at some time to use a tape on an A Series system that was created by a different type of computer system. If the different computer system did not create an ANSI-standard tape label, you must design your program to read the tape as an unlabeled tape. You can also use this technique to enable a program to read a tape whose label has become corrupted.

When a process attempts to open an unlabeled tape, the process typically becomes suspended until an operator enters a UL (Unlabeled) system command. This command specifies the tape drive to use for the file. You can prevent the need for the operator to enter this command. However, you must use a different technique than was previously described for labeled tapes. The SERIALNO attribute has no meaning for unlabeled tapes.

Instead, if you know the physical unit number of the drive where the correct tape will be mounted, you can design the program to assign the physical unit number of that tape drive to the UNITNO file attribute. A file open operation then opens any tape that happens to be on the specified tape drive. This method should not be used unless you can ensure that the correct tape will be mounted on the tape drive when the program runs. Note also that access to unlabeled tapes and to the UNITNO file attribute are restricted on systems running Security Services for ClearPath MCP security enhancement software when the security options UnLabeledTapes and NonPrivUnitNo have the values NOTOK; refer to the *Security Administration Guide* for details.

A process can also be suspended when it reaches the end of an unlabeled tape file. This happens because, depending on the circumstances, a tape mark can indicate the end of the file or simply the end of a reel. If the LABEL file attribute value is OMITTED, the tape mark is interpreted to mean that the file continues on another tape reel. The process becomes suspended until the operator enters a UL command (to specify where the next reel is located) or an FR (Final Reel) system command.

If you know in advance that the unlabeled tape file will be confined to a single reel, you can prevent the process from suspending at the end of the file. To do this, you must declare the file with a LABEL value of OMITTEDEOF. In this case, when the process reads to the end of the file, the system returns an end-of-file condition on the read operation. The process can check the result of the read operation and take appropriate action. This method saves the operator the trouble of entering the FR command.

Using the AUTORM Option

A process can become suspended if it attempts to enter a file into the disk directory and a file of the same title already exists. The system displays a “DUP LIBRARY” RSVP message for the process. The process does not proceed any further until an operator enters an RM (Remove) system command. The RM command causes the system to remove the existing file. You can save the operator from having to enter an RM command by setting the AUTORM option. AUTORM can be set for a process through assignments to the OPTION task attribute, or for the whole system through the OP (Options) system command. The AUTORM option causes the system to automatically remove any old duplicate files that a process encounters. For further information about disk directories and the AUTORM option, refer to “Entering a File in the Directory” in Section 19, “Using Shared Files.”

Using the ORGUNIT Value for ODT Files

A process can become suspended when it executes a statement that opens an ODT file. For information about how to prevent this process suspension from occurring, refer to the discussion of ODT terminal communications in the Section 3, “Tasking from Interactive Sources.”

Using Conditional ACCEPT Statements

A process can become suspended when it executes an ACCEPT statement to prompt the operator for input. For information about how to prevent this suspension from occurring, refer to the discussions of the conditional ACCEPT statement and the ACCEPTEVENT task attribute in Section 3, “Tasking from Interactive Sources.”

Section 7 **Controlling Processor Usage**

You can control two aspects of the processor usage for a process: process priority and total processor usage. In addition, you can monitor the processor usage of a particular process to gain an understanding of the resource usage patterns of the process.

Controlling Process Priority

The system is designed to efficiently execute large numbers of processes simultaneously. However, each system incorporates a limited number of processors, including central processors, I/O processors, and data link processors (DLPs). Each system also has a finite amount of main memory. On a heavily used system, all the processes in the mix are competing for the use of these system resources.

However, it may be that not all these processes are equally urgent from the user's point of view. The system provides the concept of *priority* to allow you to specify which processes should receive preference in the competition for system resources.

The primary effect of process priority occurs in cases where more than one process is ready to use a central processor. Each central processor executes only one process at a time, but divides its time among all the processes on the system. The system maintains a list, called the *ready queue*, of all processes that are waiting for a processor, arranged in priority order.

A processor continues executing a particular process until one of three things happens: the process reaches a natural stopping point (for example, because it is waiting for an I/O to complete), a higher-priority process appears in the ready queue, or the process exceeds its time slice and a process of equal priority is present in the ready queue. The processor then retrieves the higher-priority process from the ready queue and begins executing it.

The priority of a process is determined by several factors, only some of which can be controlled by the user. For example, some system software processes have a higher priority than can be assigned to an ordinary application process. For a complete overview of factors affecting process priority, refer to the *System Administration Guide*.

One aspect of priority that you can control, within certain limits, is the PRIORITY task attribute value. The PRIORITY task attribute has a range of values from 0 to 99, with the higher values indicating higher priority. The default value is 50. You can assign a PRIORITY value to a process anytime before initiation, either through task equations or assignments to a task variable. Once a process is initiated, any programmatic assignments to the PRIORITY task attribute change the task attribute value without affecting the actual priority at which the process executes. The new PRIORITY task attribute value is returned when the task attribute is read, and displayed in the output of various system commands.

The only way to effectively change the priority of an in-use process is with the PR (Priority) system command. This command changes the PRIORITY task attribute value and also causes the system to enforce the new priority value.

One point to bear in mind about this attribute is that its effects are absolute rather than proportional. That is to say, the system always gives the processor to the highest-priority process that is ready to use it. A PRIORITY value of 51 gives as much advantage over a PRIORITY of 50 as a PRIORITY value of 99 does. If the process with the PRIORITY of 51 is very processor-intensive, it could prevent the process with PRIORITY 50 from receiving any processor time at all. For this reason, you should be cautious about raising the PRIORITY value of a processor-intensive process.

On the other hand, it is sometimes helpful and appropriate to raise the priority of interactive processes. An interactive process is one that is largely driven by input from a user at a terminal. Such a process typically spends most of its time waiting for the user to enter commands. Once the user does enter a command, the user typically has to wait for a response before being able to accomplish any further useful work. If the processor usage of the process is small and occasional, you can improve response time by raising the priority with relatively little impact on overall system performance.

For information about how to determine whether a process is processor-intensive, refer to “Understanding Processor Usage Accounting” later in this section.

The system administrator can place some constraints on the values you are able to assign to the PRIORITY task attribute. For example, the administrator can assign a PRIORITY limit to a job queue. If you write a WFL job that is initiated from that job queue, the job cannot request a PRIORITY value higher than the job queue PRIORITY limit. Similarly, the system administrator can assign a value to the PRIORITY attribute of your usercode. CANDE and MARC read the PRIORITY attribute of your usercode when you log on. When you initiate a task from a CANDE or MARC session, CANDE and MARC do not allow you to assign the PRIORITY task attribute a value higher than your PRIORITY usercode attribute.

Aside from the PRIORITY task attribute, the major feature you can use to manipulate process priority is the *MP <file title> + CONTROL* form of the MP (Mark Program) system command. This option marks an object code file as a control program. Thereafter, whenever that program is initiated, it runs in the same priority category that message control systems (MCSs) do. This category gives higher priority than WFL jobs or application programs have, but lower priority than invisible independent runners.

An additional effect of control program status is that it prevents the system from *scheduling* a process (that is, delaying initiation of the process) when there is a shortage of available memory. If you mark too many programs with control program status, the result can be that system memory becomes overloaded, with a resulting adverse effect on system performance. Therefore, you should use caution in marking programs with control program status.

The system also places WFL jobs in a special priority category. WFL jobs receive higher priority than all application programs, but lower priority than control programs, MCSs, and invisible independent runners.

The system uses the PRIORITY task attribute only when comparing processes that are in the same priority class. Thus, a WFL job running with a PRIORITY value of 1 still has a higher priority than an ordinary process with a PRIORITY of 99.

The system places WFL jobs in the high priority class because their only purpose, in most cases, is to initiate tasks; the sooner the job initiates each task, the sooner the system can evaluate the priority of each task on its own merits. However, it would not be possible to rewrite a typical application in WFL to take advantage of its priority. WFL is specialized for tasking functions and has no ability to read from or write to files.

The high priority class is limited to a maximum of 60 seconds of processor time. After a WFL job has used 60 seconds of processor time, its priority bias is automatically removed and the job competes for processor resources based on its visible priority value, in the same way most other tasks compete for processor resources.

Running a looping WFL job at priority 99 can still seriously impact system performance in the same way as any looping task running at priority 99 impacts the system performance. In general, the priority of a WFL job should be about the same as the priority of the tasks that it initiates.

Because the PRIORITY task attribute and control program assignments have a potential to affect overall system performance, you should generally consult with the administrator of your system before raising the priority of any particular process.

Limiting Processor Usage

You can use the MAXPROCTIME task attribute to set a limit on the amount of processor time that a process can use. The accumulated processor time for a process is stored in the ACCUMPROCTIME task attribute. When ACCUMPROCTIME reaches a value equal to that of MAXPROCTIME, the system discontinues the process and displays the error message PROCESSOR TIME EXCEEDED.

The main use of the MAXPROCTIME task attribute is to ensure that WFL jobs are placed in the proper job queues. For example, suppose there is a high-priority job queue that is intended for short jobs. The system administrator can use the PROCESSTIME job queue attribute to provide default and limiting values for the MAXPROCTIME task attribute of all WFL jobs that use the job queue. If you submit an extremely processor-intensive job through that job queue, the system discontinues the job when it exceeds the MAXPROCTIME value. This gives you an incentive to resubmit the job through a different job queue. For an introduction to the subject of job queues, refer to the discussion of WFL in Section 4, “Tasking from Programming Languages.”

Understanding Processor Usage Accounting

Programs vary a lot in terms of their patterns of processor usage. Understanding the processor usage of a program can help you to decide the priority at which it should run. It can also help you to diagnose inefficiencies in program design or problems in overall system performance.

The system divides the processor usage of a process into several categories, which can be displayed through system commands, examined through task attributes, or read in the system log.

The system command that displays processor usage information is the TI (Times) command. The following is an example of the output:

```
5825 TI

       TIMES FOR 5825
         PROCESS   = 00:00:37 LIMIT 0:01:20
         IO        = 00:00:01 LIMIT 0:02:40
         READYQ    = 00:00:56
         INITPBIT  = 00:00:06 3217 OPERATIONS
         OTHERPBIT = 00:00:02 1521 OPERATIONS
         ELAPSED   = 00:11:40
```

In the TI command output, all the times are expressed in a format of <hours>:<minutes>:<seconds>. The following are the meanings of these fields in the TI command output:

- PROCESS

  The accumulated processor usage of the process, with the exception of the process time spent on presence-bit operations. (See the following descriptions of INITPBIT and OTHERPBIT.) The LIMIT time, if displayed, corresponds to the MAXPROCTIME task attribute value.

- IO

  The accumulated I/O usage for the process. The LIMIT time, if displayed, corresponds to the MAXIOTIME task attribute value.

- READYQ

  The accumulated ready queue time for the process. Ready queue time is the time spent waiting for the processor to become available. If this value is excessive, it indicates either that the processor is overloaded or that other higher priority processes are dominating the processor.

- INITPBIT

  The amount of processor time spent on initial presence-bit operations. These are operations that create arrays, files, and code segments for this process. This value is followed by a count of the number of presence-bit operations.

  If the value of INITPBIT is high compared to the value of PROCESS, this can be a symptom of poor program structure. For example, if a large local array is declared in a procedure that is entered repeatedly, then much processor time is spent recreating the array each time the procedure is entered, thus resulting in a high INITPBIT value. You can prevent this problem by declaring the array globally to the procedure, or by declaring the array with an OWN clause (in ALGOL programs only).

- OTHERPBIT

  The amount of processor time spent on noninitial presence-bit operations for this process. Noninitial presence bit operations read arrays and code segments back into main memory after they have been overlaid. The value of OTHERPBIT can vary widely for different runs of the same program, depending on the memory demands that are made by other active processes. If this value is very high, it might indicate that memory is overloaded and the system is thrashing.

- ELAPSED

  The amount of real time that has passed since the process was initiated. This value is stored in a different form in the ELAPSEDTIME task attribute.

  The value of ELAPSED can be greater than or less than the sum of the other values listed in the TI display. The ELAPSED value can be greater because it includes time spent waiting on events, and this waiting time is not displayed separately. The ELAPSED value can be less because, in some cases, a process might be using the processor and performing one or more I/O operations at the same time.

Most of the resource usage information that can be displayed for a process can also be interrogated through task attributes.

The ACCUMPROCTIME task attribute returns the accumulated processor time. The value does not include processor time spent on presence-bit operations. The INITPBITTIME, INITPBITCOUNT, OTHERPBITTIME, and OTHERPBITCOUNT task attributes return the times and counts for presence-bit operations. The ACCUMIOTIME task attribute returns the accumulated I/O time for the process. The ELAPSEDTIME task attribute returns the total elapsed time.

The values these task attributes return are expressed in units of 2.4 microseconds, except if the attributes are read from WFL, which expresses the values in units of seconds.

The system log (SUMLOG) records several categories of processor usage for each process. This information includes the processor time, I/O time, ready queue time, and p-bit times and counts. This information is stored in the Major Type 1, Minor Type 2 (EOJ) and Minor Type 4 (EOT) log entries. For a description of these log entry types, refer to the *System Log Programming Reference Manual.*

Section 8
Controlling Process Memory Usage

Process execution takes place in a memory environment that is shared with all the other processes in the mix. Understanding that environment can help you to improve process performance and prevent a process from impairing overall system performance.

This section is aimed at programmers, and concentrates on the aspects of process memory usage that can be affected by task attributes and object code file location.

Understanding Process Memory Usage

A process consists of several distinct components, some of which reside in main memory and some of which can reside in virtual memory. The system uses presence-bit operations to create or re-create some of the process components in main memory.

Main Memory and Virtual Memory

The effective memory capacity of a system consists of the following two components:

- Main memory

  This is the total amount of memory that is physically present.

- Virtual memory

  This is an additional amount of memory whose existence is simulated by temporarily copying segments of main memory out to disk. The use of virtual memory enables the system to handle more processes than can fit into main memory at the same time.

To facilitate memory management, the system classifies each of the segments of main memory into one of the following three categories:

- Available memory

  This is memory that is not assigned to an in-use process. The system is free to allocate this memory as the need arises.

- Overlayable memory

  This is memory that is assigned to in-use processes, but which can nevertheless be overwritten if necessary. For data segments, the system must copy the data to a different location in main memory or to an overlay disk file before reusing the memory segment. For code segments, the system can simply overwrite the code segment with other code or data. The system can read the code segment back in from the object code file the next time it is needed.

- Save memory

  Save memory consists of structures that, for performance reasons, must be kept in main memory at all times. The system never copies these segments out to disk, nor does it move them around in main memory except for stack stretches. (Refer to "Preventing Stack Stretches" later in this section.)

When the processes in the mix require far more memory than exists as main memory, the processor is forced to spend a lot of time performing overlays. When the time spent performing overlays begins to significantly impair system performance, the situation is called *thrashing*.

Process Components

Every running process includes the following basic structures in memory:

- Process information block (PIB)

  This structure contains process control information visible only to the operating system. The PIB also contains a reference to the TAB.

- Task attribute block (TAB)

  This structure stores the task variable for the process and includes the values of all task attributes. In addition to the TAB of the process, the system creates a separate TAB for each task variable the process declares. Thus, reusing a task variable can slightly reduce the memory usage of a process. For cautions related to task variable reuse, refer to the *Task Attributes Programming Reference Manual*.

- Process stack

  This structure includes storage areas, or descriptors pointing to outside storage areas, for all variables declared by the process. The top of the process stack also serves as a working storage area that the processor can use when evaluating expressions. For information about estimating and limiting process stack size, refer to "Controlling Process Scheduling," "Preventing Stack Stretches," "Protecting against Stack Overflow," and "Restricting Save Memory Usage" later in this section.

- Code segment dictionary

  This structure includes descriptors pointing to the locations of

  – The code segments used by the process.

  – Constant data used by the process, such as value arrays and translate tables.

  – Sequence numbers for all the code segments, if the program was compiled with the LINEINFO compiler options set. (For information about LINEINFO, refer to Section 10, "Determining Process History.")

For further information about code segment dictionaries, refer to "Controlling Code Segment Dictionary Sharing" later in this section.

Presence-Bit Operations

When the processor constructs an array, a logical file, or a code segment for a process in memory, this action is referred to as a *presence-bit operation.* An initial presence-bit operation is one that creates an array or a code segment because the related procedure has just been invoked. A noninitial presence-bit operation is one that copies an array or a code segment back into main memory from disk.

The number of initial presence-bit operations performed by a process, and the processor time they take, are relatively stable from one run of a program to the next (provided that each run results in the same sequence of procedure entrances). However, the number of noninitial presence-bit operations performed by a process depends to a large extent on how much memory is being used by all the other processes in the mix. When memory is crowded, more noninitial presence-bit operations are performed.

You can monitor the number of presence-bit operations for a process, and the processor time spent on them, by using the TI (Times) system command, by interrogating task attributes, or by reading system log entries. Refer to the discussion of processor usage accounting in Section 7, "Controlling Processor Usage."

Controlling Code Segment Dictionary Sharing

The system generally causes processes to share the same code segment dictionary if the processes are executions of the same program. This technique reduces total memory usage and thus reduces the system overhead for memory management. The result is that all the processes in the mix are able to run more quickly.

There are a few situations in which the system does not use the same code segment dictionary for processes that are executing the same program. Understanding these situations can help you to conserve memory and control process privileges.

To decide whether two processes are executions of the same program, the system compares the object code file title for each process. Suppose you have one copy of OBJECT/PROG on the family SYSPK, and another copy on a family called DOCPK. In this case, the family part of the object code file title is different. The system therefore regards these as two different programs. If people are using both programs simultaneously, the system has to create two separate code segment dictionaries. You can eliminate this duplication, and thus reduce system overhead, by placing a single object code file in a central location where all the users have access to it.

Even if two processes have the same object code file title, the system still assigns them different code segment dictionaries in the following cases:

- If either of the processes is running in Test and Debug System (TADS) mode. A process runs in this mode if you compile the program with the TADS compiler option set and run the program with the TADS task attribute set. TADS mode gives ALGOL, C, COBOL74, COBOL85, or FORTRAN77 processes access to the TADS facility for debugging programs. You can prevent unnecessary duplication of code segment dictionaries by using TADS mode only for the rare cases when you are actually doing debugging.
- If the object code file is overwritten. An object code file can be overwritten if, for example, you recompile the program or use a COPY statement to replace it with a different program having the same title. If the object code file of a running process is overwritten, the system retains the old object code file as a temporary file. The running process continues to use its code segment dictionary and the old object code file. However, any new processes that are initiated with the same object code file title receive a code segment dictionary reflecting the new object code file. The main point to bear in mind is that updating or removing an object code file has no effect on processes that are already running.

The MP (Mark Program) system command can be used to assign various options to object code files. These options are described in Section 5, "Establishing Process Identity and Privileges." Be aware that these options do not affect instances of the program that are already running when the MP command is entered.

Controlling Process Scheduling

A process is considered scheduled when it has been submitted for initiation, but the system is delaying initiation of the process. Scheduling can occur for any of a number of reasons, most of which are not preventable by the programmer. For an explanation of these reasons, refer to the discussion of process scheduling in the *System Administration Guide*.

One type of process scheduling that you can prevent, to some extent, is scheduling due to a lack of available memory. The system performs this type of scheduling if it estimates that a particular process requires more memory for efficient execution than is currently available. The system places the process in a scheduled state until more memory becomes available.

The initial memory estimate for a process is created by the compiler and stored in the object code file. The memory estimate is an estimate of the average amount of memory that must be available for the process to run efficiently (that is, without excessive overlays). This ideal amount of memory is referred to as the *working set* of the process.

Each time the object code file is executed, the system writes an updated memory estimate into the object code file. The updated estimate is based on the average of the existing estimate and the memory usage during the current run. The effect is gradually refined and improved the accuracy of the memory estimate each time the object code file is run.

The memory estimate for a process consists of two separate statistics: the estimated process stack size, and the estimated memory usage for data and code segments. You can override the process stack size estimate through an assignment to the STACKSIZE task attribute. You can override the data and code estimate through an assignment to the CORE task attribute. By assigning large or small values to the STACKSIZE and CORE attributes, you can make it more or less likely that the system will schedule a process when it is submitted for initiation.

It is rarely necessary or desirable for you to make assignments to the STACKSIZE and CORE task attributes. It is true, for example, that you can help ensure that a process will not be scheduled by setting STACKSIZE and CORE to artificially low values. However, doing so could cause a system to begin thrashing, with the result that system performance could dramatically worsen.

The following are situations in which it might make sense to assign STACKSIZE and CORE values:

- When initiating a program that is newly compiled. The memory estimate in such an object code file has not been refined through repeated use.
- When initiating a program that is stored on a read-only disk. Many types of disk drives have a switch that enables an operator to put the disk drive in read-only mode. If an object code file is stored on a read-only disk, the system is not able to update the memory estimate in the object code file after each run.
- When initiating a program whose memory usage varies widely from one run to the next. This can be the case if the memory usage depends on the type and quantity of the data passed to the program for processing.

Even in these situations, there is no point in your assigning a CORE or STACKSIZE value unless you have some information about what the working set of the program really is. You can get some general idea of the memory usage of a program by running it and examining statistics with the LOGANALYZER utility. You can use the LOGANALYZER *MIX* option to return log entries for a particular process. In the Major Type 1, Minor Types 2 and 4 (EOJ and EOT) log entries, you can find figures for the average memory usage of a process.

The average memory usage is not necessarily the ideal memory usage for a process; if memory is very crowded, the process might be constrained to use a less-than-optimum amount. Still, these statistics might be helpful in some cases.

For example, suppose you run the same reporting program twice a day: once to report on the payroll for your whole department, and once to summarize the time charged to certain accounts. Using LOGANALYZER, you might discover that the program consistently uses less memory for the time-accounting run than it does for the payroll run. In this example, the memory estimate in the object code file tends to reflect an average of the two types of runs. You can improve on this memory estimate for individual runs by assigning one set of CORE and STACKSIZE values for the time accounting run and another set of CORE and STACKSIZE values for the payroll run.

Preventing Stack Stretches

The process stack for a process can vary in size significantly during the execution of the process. Each time a process enters a procedure, the system adds a storage area called an *activation record* on top of the process stack. In turn, the system removes an activation record from the process stack when a procedure is exited. The size of the activation record, in turn, is affected by the number and type of objects that are declared in a procedure, as well as by the nature of the computations made in the procedure.

The process stack resides entirely in a contiguous region of save memory. To leave room for the process stack to grow, the system sets aside a region of memory larger than the initial size of the process stack. The amount of memory set aside is based on the stack memory estimate or the STACKSIZE task attribute value, as described under "Controlling Process Scheduling" earlier in this section.

In addition to its effects on scheduling, the stack estimate has implications on the performance of the process. If the process stack grows to a greater size than the stack estimate, the system must stretch the process stack. The stack stretch is an expensive operation because it requires moving the entire process stack to a different location and updating all ASD table entries that are used by the process.

You can tell if stack stretches occurred for a process by checking the job summary or, in some cases, the system log. (By default, stack stretches are recorded only in the job summary.) If a stack stretch occurred, an entry such as the following appears:

```
  18:01:19        3211  STACK EXTENDED FROM 511 TO 704 WORDS.
```

If a process experiences stack stretches, the system changes the stack estimate in the object code file to the average of the previous estimate and the final stack area allocated in the current run. Thus, if the program is fairly consistent in the amount of stack space it needs, then the stack estimate can become very accurate after several runs. The result is that stack stretches are no longer necessary and the program executes faster.

You can consider making STACKSIZE assignments to improve performance in the same situations where you might want to use STACKSIZE assignments to regulate scheduling, that is, for newly compiled programs, programs stored on read-only disk units, or programs whose memory usage varies widely from one run to the next. You can help decide on an appropriate STACKSIZE value by observing the STACK EXTENDED log entries for various program runs.

Note that you can assign values to the STACKSIZE only before process initiation (for example, through task equations). You cannot use STACKSIZE to change the stack space allocated for an in-use process.

Protecting against Stack Overflow

Another task attribute related to process stack size is STACKLIMIT. The STACKLIMIT value places a limit on the size to which the process stack can be stretched. If a process cannot proceed further without exceeding this limit, the system discontinues it and returns the error message "STACK OVERFLOW."

The STACKLIMIT task attribute defaults to a value of 6000 words for most programs. (The system uses a higher default for certain types of programs that are likely to need a larger stack size.) It is unlikely that any process stack will reach the default STACKLIMIT size if it is running as intended. The "STACK OVERFLOW" error usually indicates that a process has entered into an infinite loop of recursive procedure calls. Because each procedure call adds an activation record to the process stack, the process stack quickly exceeds the STACKLIMIT value.

The PERFORM statement in COBOL also adds to the size of the process stack, and an infinite loop of recursive PERFORM statements also causes a "STACK OVERFLOW" error. (This is true even though a PERFORM statement does not create an activation record. A PERFORM statement adds a different type of structure to the process stack.) By contrast, GO TO statements do not increase the process stack size and so cannot cause a "STACK OVERFLOW" error.

If a process receives a "STACK OVERFLOW" error, and you determine that the process was running as intended, then you can remedy the problem by assigning a higher value to the STACKLIMIT task attribute before initiating the process. The highest value STACKLIMIT can be set to is about 64000 words.

Because a process stack is built exclusively in save memory, the save memory restrictions discussed in the next subsection also effectively limit the size of the process stack.

Restricting Save Memory Usage

The amount of save memory in use on the system is of particular concern to a system administrator because it is one of the factors that can have a big effect on system thrashing. As the percentage of memory allocated to save memory increases, the probability that the system will begin thrashing also increases.

A number of memory structures associated with a process are stored in save memory. The total save memory usage of a process varies, depending on factors such as process stack size, code segment dictionary sharing, and file and array declarations.

Since processes use varying amounts of save memory, and save memory usage impacts system performance, it is possible for an individual process to harm system performance by using excessive save memory. This might happen, for example, if the process enters an infinite loop of procedure calls, as discussed under "Protecting against Stack Overflow" earlier in this section. This might also happen if the process uses large numbers of arrays and files.

You can prevent a process from exceeding a planned level of save memory usage by assigning a value to the SAVEMEMORYLIMIT task attribute. If the save memory usage of the process exceeds the limit set by this attribute, the system discontinues the process and displays the message "USER SAVE MEMORY LIMIT EXCEEDED."

If a process is discontinued with the "USER SAVE MEMORY LIMIT EXCEEDED" error, you should check to see if it was running normally or looping. If it was running normally, you can consider program design measures to reduce the save memory usage. Alternatively, you can raise the SAVEMEMORYLIMIT value and plan to run the process at a time when the system is not very busy.

The system administrator can place some limits on the SAVEMEMORYLIMIT value your processes can have. For example, the system administrator can assign a SAVEMEMORYLIMIT value to your usercode. This value becomes the maximum SAVEMEMORYLIMIT value for all processes initiated with your usercode. If you assign a different SAVEMEMORYLIMIT value to a process, the system uses the lower of your SAVEMEMORYLIMIT assignment and the usercode SAVEMEMORYLIMIT assignment.

You might also find that the system administrator has assigned a SAVEMEMORYLIMIT value to a job queue you use for your WFL jobs. If the SAVEMEMORYLIMIT is assigned as a job queue default, you can override it with a different SAVEMEMORYLIMIT assignment in the job header of your WFL job. If the SAVEMEMORYLIMIT is assigned as a job-queue limit, your WFL job is rejected from the job queue if the job header includes a higher SAVEMEMORYLIMIT assignment. For more information about job queues, refer to the discussion of WFL in Section 4, "Tasking from Programming Languages."

Section 9
Controlling Process I/O Usage

The I/O activity of a process is primarily determined by various I/O statements that the process executes. These include statements for reading from, writing to, opening, and closing files. For an overview of I/O features available in programming languages, refer to the *I/O Subsystem Programming Guide.*

There are also a number of task attributes that affect various global aspects of process I/O activity. For example, you can use task attributes to establish default locations for files used by a process, or to specify defaults for handling printer output produced by a process. This section introduces the functions of task attributes that affect process I/O activity and some related system commands.

Establishing the Default Usercode for Disk Files

One of the effects of the USERCODE task attribute is to supply a default usercode for all disk files used by a process. For example, suppose a process runs with a USERCODE value of FERMAT. Suppose also that this process attempts to open a file with a TITLE file attribute of *INPUT/DATA ON DBFAM*. In this case,

- If the NEWFILE file attribute is TRUE, the system creates the file under usercode FERMAT and changes the TITLE file attribute to *(FERMAT)INPUT/DATA ON DBFAM*.
- If the NEWFILE file attribute is FALSE, the system searches for the file first under the title *(FERMAT)INPUT/DATA ON DBFAM*. If no file of that title exists, the system searches for the file under the title **INPUT/DATA ON DBFAM*.

A process can override the default behavior by assigning a usercode as part of the TITLE file attribute before attempting to open the file. For example, a process could assign TITLE the value (LUANN)INPUT/DATA ON DBFAM. In this case, the system searches for the file only under usercode LUANN.

Note: *The preceding description of default usercodes applies only to files for which the SEARCHRULE file attribute value is NATIVE. For an explanation of the file searching rules used when SEARCHRULE = POSIX, refer to "Specifying a Current Directory" later in this section.*

Modifying File Attributes

File attributes are entities that describe the properties of files. For example, file attributes specify the title of the file and the physical device type on which it resides (such as disk or tape). Programs can specify attributes for a file in the file declaration. Programs can also add to or change file attributes with file attribute assignment statements later in the program.

After you have written and compiled a program, you might later find that you would like the program to start using a different set of file attributes than were originally specified in the program. One method for doing this is to rewrite and recompile the program. This method can be time consuming for the programmer, and can make heavy use of system resources such as processor time and memory.

Alternatively, you can modify the file attributes used by a program through constructs called *file equations*. For example, suppose a program uses a file called IN and another file called OUT. In a CANDE *RUN* command, you could use file equations to specify different titles for these files in the RUN statement that initiates the program. The following is an example:

```
RUN REPORT1;FILE IN = (HKANE)INDATA, OUT = (HKANE)OUTDATA
```

File equations thus enable you to modify the file attributes used by a program without having to rewrite or recompile the program. However, in order to use a file equation you first have to know the internal name of the file. The internal name of the file is determined by the value of the INTNAME file attribute. If the program does not specify a value for INTNAME, then INTNAME defaults to the value of the file identifier used for the file in the program. You can determine the internal name of a file by looking at the file declaration in the program source file. Thus, either of the following ALGOL declarations creates a file with an internal name of SOURCE:

```
FILE CUSTDATA(INTNAME = "SOURCE.");
FILE SOURCE;
```

The syntax for file equations in CANDE, MARC, and WFL is almost identical. For example, to change the device kind of the file with the internal name of SOURCE, you can append the following to a RUN statement submitted through any of these sources:

```
FILE SOURCE(KIND = REMOTE);
```

The flexibility provided by file equations can be so convenient that programmers sometimes design a program with the intention that the user will use file equations. For example, in the documentation for various compilers and utilities, you can find descriptions of the internal names of files used by these compilers and utilities. These internal names are documented so that you can use them in file equations.

Note that the same file attribute can be assigned different values by file declarations, file attribute assignment statements, and file equations. In these cases, the values assigned through file equations override those specified in the file declaration. File equations are in turn overridden by any conflicting file attribute assignment statements executed by the program. A programmer can prevent file equations from having effect simply by specifying file attributes through file attribute assignment statements rather than through attribute assignments in the file declaration.

When you specify file equations for a process, the system stores the equations in the FILECARDS task attribute of the process. For further information about FILECARDS, refer to the *Task Attributes Programming Reference Manual*.

One of the file attributes that it is frequently useful to change at run time is the FAMILYNAME file attribute. You can save yourself the trouble of including FAMILYNAME equations for each disk file in the program by using the FAMILY task attribute instead. Refer to "Specifying Family Substitution" later in this section. Also, you can establish default values for the file attributes related to printing by using the PRINTDEFAULTS task attribute, as described under "Programmatic Control Over Printing" later in this section.

You might find occasionally that you initiated a process and forgot to specify the correct file equations. The system suspends the process if both the following conditions are true:

- The process attempts a open operation with the WAIT option specified or with no specific open option.
- The process is unable to open the specified file because of a missing or incorrect file attribute value.

You cannot use file equations to remedy this problem, because file equations must be specified at process initiation. Instead, you can use the FA system command to supply the needed file attribute values. For example, suppose a process is suspended because it tried to open a file SOURCE with KIND = TAPE, and the file is a disk file. The Y system command output looks like this:

```
STATUS OF TASK 38057\23046 at 07:28:51
Program name: *SYSTEM/DUMPALL ON DISK
    Codefile created: Monday, July 2, 2001 (2001183) at 11:07:28
Priority: 50
Origination: SB154/CANDE/3 (LSN 320)
MCS: SYSTEM/CANDE
Usercode: JASMITH
Stack State: Waiting on an event, Assigning a file
RSVP: NO FILE SOURCE (MT) #1
Reply: NF, FA, UL, IL, OK, DS
```

Note that the name SOURCE, which appears on the RSVP line, is the file title rather than the internal name. However, it does not matter if you do not know the internal name in this case. When you specify file attribute assignments in an FA command, the system automatically applies the assignments to the file the process is trying to open. The following FA command enables the process to open the file and resume running normally:

```
  5692 FA KIND = DISK
```

For detailed descriptions of all the file attributes available on the system, refer to the *File Attributes Programming Reference Manual*.

Controlling Disk File Usage

You can use task attributes to take advantage of several disk storage features, including the concept of disk families, disk directories, and the disk resource control system.

Specifying Family Substitution

Disk *families* are groups of disk units that are labeled with a common name and treated as a logical unit. Disk families are defined through system configuration and system commands. Once a family has been defined, a program can use the FAMILYNAME file attribute to specify that a file is located on that family.

It is quite often the case that all the input and output files used by a process are located on one, or possibly two, disk families. Now, suppose that you include FAMILYNAME file attribute assignments in the program for each file used by the program. The system administrator might later decide to change the name of a disk family, or might ask you to place your files on a different family. Further, you might need to run your program on a different host system, where no family of the original name exists. For any of these reasons, it might become desirable for the program to look for its files on a different family than is specified in the program code.

The simplest way to make a process use a different family is by assigning the FAMILY task attribute. This task attribute specifies a target family and one or two substitute families to be searched for files. For example, suppose a process expects to find its files on the family SYSPK. This is considered the target family. To make the process look for its files on the family PARTS instead, you could use the assignment *FAMILY SYSPK = PARTS ONLY*.

Note that this FAMILY value affects only files with a FAMILYNAME value of SYSPK. For example, if the file has a FAMILYNAME of DBFAM, then the process still looks for the file on DBFAM.

Note also that only one FAMILY value can be in effect at a time. For example, suppose the existing FAMILY value of a process is *FAMILY SYSPK = PARTS ONLY*. In this case, an assignment such as *FAMILY DBFAM = PACK ONLY* disables family substitution for the SYSPK family and enables substitution for the DBFAM family.

If a program does not specify a FAMILYNAME for a disk file, the system searches for the file on the family named DISK. If you want the users of a program to specify a FAMILY value, you can leave the FAMILYNAME unspecified for all the files. The user can override the default FAMILYNAME of DISK with a FAMILY task attribute assignment such as *FAMILY DISK = DBFAM ONLY*.

Sometimes it is useful to specify two substitute families in the FAMILY value. For example, you might have a WFL job that runs utilities stored on the family named DBFAM, which in turn use data files stored on the family named SYSPK. In this case, you can use a FAMILY statement like the one in the following WFL job:

```
?BEGIN JOB;
  FAMILY DISK = DBFAM OTHERWISE SYSPK;
RUN OBJECT/DAILY/RUN;
RUN OBJECT/REPORT/GENERATOR;
?END JOB
```

Because the FAMILY assignment is in the job header, the system searches for OBJECT/DAILY/RUN and OBJECT/REPORT/GENERATOR on DBFAM family and then on SYSPK family. The FAMILY task attribute value is inherited by both tasks, which search for their data files on DBFAM and SYSPK families.

When a family statement specifies two substitute families, the first is referred to as the *primary family* and the second as the *alternate family*. In the previous example, DBFAM is the primary family and SYSPK is the alternate family.

When a process attempts to create a new file on the target family, the system creates the file on the primary family instead. When a process attempts to open or execute an existing file on the target family, the process searches for the file first on the primary family and then on the alternate family. If the TITLE file attribute of the existing file does not begin with a usercode or an asterisk, then the system searches for the file in the following locations, in the order shown:

1. On the primary family, under the usercode of the process
2. On the primary family, as a nonusercoded file
3. On the alternate family, under the usercode of the process
4. On the alternate family, as a nonusercoded file

If the TITLE attribute of a file does not specify a usercode, and the NEWFILE file attribute is TRUE, the system creates the file on the primary family under the usercode of the process.

Another method for overriding the FAMILYNAME file attribute is through file equations, as described under "Modifying File Attributes" earlier in this section. The following are two advantages to using FAMILY instead of file equations for this purpose:

- A single FAMILY assignment affects all the files in the program that have the specified target FAMILYNAME. Using file equations, you must specify each file individually.
- The FAMILY assignment overrides the target FAMILYNAME wherever it is mentioned in the program. By contrast, file equations are applied when a file is first declared. The program can later use file attribute assignment statements to override the values supplied through file equations.

Note: *The preceding description of family substitution applies only to files for which the SEARCHRULE file attribute value is NATIVE. For an explanation of the file searching rules used when SEARCHRULE = POSIX, refer to "Specifying a Current Directory" later in this section.*

Preventing File Duplications

The system does not allow two permanent disk files with the same title to exist on the same disk family. In order to handle attempts to duplicate disk file titles, most system administrators set the system option AUTORM. If a process attempts to enter a file in the disk directory for a family, but a file with the same name already exists in that family's disk directory, then the AUTORM option causes the existing file to be removed. For further information, refer to the discussion of preventing process suspension in Section 6, "Monitoring and Controlling Process Status."

Automatically Restoring Missing Disk Files

If your site uses the archiving subsystem to perform system backups, you can use the AUTORESTORE task attribute to reduce the likelihood that a process will be suspended for attempting to open a nonresident disk file.

The system checks to see if there is an archive record specifying the location of a backup copy of a file if a process attempts to open the file and all the following conditions are true:

- The process has an AUTORESTORE value of TRUE.
- The file title is under the usercode of the process.
- The disk file is not present on the requested family because the file was removed by one of the following WFL statements: ARCHIVE RELEASE or ARCHIVE ROLLOUT.

If there is an archive record for the file, the system issues a request for an operator to mount the necessary tape. When the tape is mounted, the system copies the file back onto disk. At this point, the process that was attempting to use the tape resumes execution.

If the system is unable to restore the file for any reason, the process becomes suspended and appears in the W (Waiting Entries) system command display with a "NO FILE" RSVP message.

For an overview of the system archiving and AUTORESTORE features, refer to the *System Administration Guide*.

Limiting Disk Usage

The system administrator can use the disk resource control (DRC) system to limit the disk usage of each user. For each usercode, the administrator can establish the maximum amount of space the user can use on each family. The limits are applied in a somewhat different manner for permanent and temporary disk files.

For permanent disk files, the limits imposed by the system administrator apply to the total of all the user's files on a given family. Any process that attempts to increase the total file usage beyond the limit receives an I/O error. For example, suppose the system administrator has established a limit of 2 megabytes on the disk usage for usercode CHAN on DBFAM family. Suppose there are already 1999999 bytes of permanent files under CHAN usercode on DBFAM, and a process attempts a write operation that requires another area to be allocated for one of these files. In this case, the write operation fails.

For temporary disk files, the limits imposed by the system administrator apply to individual processes running under the specified usercode. The administrator specifies the limit by assigning a TEMPFILELIMIT attribute to the usercode. This in turn sets a limit on the value that can be stored by the TEMPFILELIMIT task attribute of processes running under the usercode. If a process attempts to increase its temporary file usage beyond the number of megabytes specified by TEMPFILELIMIT, the process receives an I/O error.

For example, if the TEMPFILELIMIT for usercode CHAN is 3 megabytes, there can be two different processes running with CHAN usercode that each use 2 megabytes for temporary files. The total temporary file usage is thus 4 megabytes. This is not a violation of the TEMPFILELIMIT because the limit is enforced on a process-by-process basis.

Note also that, unlike the permanent disk file limits, the TEMPFILELIMIT cannot be linked to a particular disk family. The process might allocate its temporary files on any family. For example, if the TEMPFILELIMIT is 3 megabytes, and the process has allocated 2 megabytes of temporary files on DBFAM, the process can allocate no more than 1 megabyte on SYSPK.

At any given time, the TEMPFILEMBYTES task attribute records the total number of disk megabytes in use by the process for temporary files. The process can interrogate this task attribute to determine the process is nearing the TEMPFILELIMIT value. Alternatively, you can design the process to include I/O error handling that enables the process to recover from temporary file limit errors.

For information about permanent and temporary disk files, and about I/O error handling, refer to the *I/O Subsystem Programming Guide*. For more information about the DRC system, refer to the *System Administration Guide* and the *System Operations Guide*.

Specifying a Current Directory

The system supports two different methods of file searching: native rules and POSIX rules. You can select the file searching method by setting the SEARCHRULE file attribute to NATIVE or POSIX. Native rules are the default.

An advantage to using POSIX rules for file searching is that POSIX allows you to define the current working directory for a process. You define this directory with the CURRENTDIRECTORY task attribute. Once you have defined a current directory, the process can specify file names using an incomplete form known as a *relative pathname.* The system automatically combines the relative pathname with the CURRENTDIRECTORY value to determine the *absolute pathname.* The absolute pathname is the complete title of the file, encoded in a POSIX-defined format. The pathname is stored in the PATHNAME file attribute.

For example, consider the following WFL statements:

```
MYSELF (CURRENTDIRECTORY = "/-/MYFAM/USERCODE/JASMITH/DATA");
FILE F1(SEARCHRULE = POSIX, PATHNAME = "TEST/ONE");
FILE F2(SEARCHRULE = POSIX, PATHNAME = "REPORT/BRIEF");
```

The preceding statements have the same effect as the following statements:

```
FILE F1(SEARCHRULE = POSIX,
        PATHNAME = "/-/MYFAM/USERCODE/JASMITH/DATA/TEST/ONE");
FILE F2(SEARCHRULE = POSIX,
        PATHNAME = "/-/MYFAM/USERCODE/JASMITH/DATA/REPORT/BRIEF");
```

You can use a SEARCHRULE value of POSIX only if the DL (Disk Location) command has been used to define a DL ROOT family on your system.

A SEARCHRULE value of POSIX has two side effects of which you should be aware:

- Family substitution is disabled.
- For usercoded files, the usercode must be explicitly included in the PATHNAME attribute or the CURRENTDIRECTORY attribute. The system does not implicitly add a usercode when none is specified.

For more details about absolute and relative pathnames, refer to the CURRENTDIRECTORY task attribute description in the *Task Attributes Programming Reference Manual*. Refer also to the descriptions of the SEARCHRULE and PATHNAME file attributes in the *File Attributes Programming Reference Manual*.

Controlling Printing

One aspect of process control is the ability to direct the printer output generated by a task. A process can control the printer output of an offspring by using relevant task attributes, such as PRINTDEFAULTS and FILECARDS.

The following subsections briefly introduce the printing control features of the system and the role that task attributes play in printing control. For complete details about controlling printer output, refer to the *Print System and Remote Print System Administration, Operations, and Programming Guide*.

For information about printing job summaries, refer to Section 10, "Determining Process History."

Default Handling of Printer Output

The system handles printer output in certain typical ways if operators, programmers, and users do not use printing-related statements to request different treatment. The following subsections describe the typical handling of printer output.

Storing Printer Backup Files Temporarily

When a program creates a printer file, the system either routes the output directly to a printer or stores the output in a printer backup file for later printing. If the output is routed directly to a printer, the program editing the file either controls all aspects of printing or allows the Print System to control printing.

If the BDNAME task attribute has a non-null value, then the system always creates a backup file. Other effects of the BDNAME task attribute are discussed under "Other Print-Related Task Attributes," later in this section.

If the BDNAME task attribute is null (the default), then the PRINTDISPOSITION file attribute determines whether output is sent directly to a printer or to a printer backup file.

Output is sent directly to a printer and all aspects of printing are controlled by the program that created the file if the PRINTDISPOSITION file attribute is set to one of the following values:

- DIRECTDLP
- DIRECT

  This value is used when the value of the DIRECTPRINTER option of the SYSOPS system command is DIRECTDLP.

Note: *The DIRECTDLP value works only on systems with printers directly attached to them.*

Output is sent directly to a printer under the control of the Print System if the PRINTDISPOSITION file attribute is set to one of the following values:

- DIRECTPS
- DIRECT

  This value is used when the value of the DIRECTPRINTER option of the SYSOPS system command is DIRECTPS.

- NOW

Note: *The NOW value provides additional functionality from the Print System.*

If the PRINTDISPOSITION value is FILEOPEN, FILECLOSE, EOT, EOJ, or DONTPRINT, then the system creates a backup file. FILEOPEN automatically creates a print request when the file is opened. For FILECLOSE, EOT, or EOJ, the system automatically creates a print request when the file is closed, at end of task, or at end of job, respectively. For DONTPRINT, the system creates a backup file but does not create a print request; you can print the backup file later with a WFL *PRINT* statement.

If the program does not assign a PRINTDISPOSITION value for the file, and no PRINTDISPOSITION is file-equated for the file when the program is initiated, then the default is inherited from the PRINTDEFAULTS task attribute. The PRINTDEFAULTS task attribute can itself inherit its value from any of several sources, as discussed in the *Task Attributes Programming Reference Manual.*

If the PRINTDEFAULTS task attribute does not specify a PRINTDISPOSITION value, then the following factors determine the PRINTDISPOSITION value:

- If either the LPBDONLY system option or the BACKUP option of the OPTION task attribute is enabled, then the PRINTDISPOSITION file attribute inherits the system default PRINTDISPOSITION. The LPBDONLY system option is set by the OP (Options) system command. The system default PRINTDISPOSITION is set by the PS DEFAULTS PRINTDISPOSITION system command. If the PS DEFAULTS PRINTDISPOSITION command is never used, the system default PRINTDISPOSITION is EOJ.
- If neither LPBDONLY nor BACKUP is enabled, then the PRINTDISPOSITION defaults to DIRECT. The PS DEFAULTS PRINTDISPOSITION value is ignored.

If the printer file is a backup file, the BACKUPKIND file attribute specifies the kind of medium on which the backup file is to be created. If the BACKUPKIND is DISK or DONTCARE, the backup file is created on the family named DISK. If the BACKUPKIND is PACK, the backup file is created on the family named PACK.

If the BACKUPKIND value is TAPE, then the process is suspended and displays an RSVP message asking the operator to mount a tape. When the tape is mounted, the backup file is created on the tape. The system does not print the backup file automatically. However, you can later use SYSTEM/BACKUP or a WFL *PRINT* command to cause the file to be printed.

The operator can use the SB (Substitute Backup) system command to specify a substitute backup medium for each possible BACKUPKIND value. The SB setting can convert any BACKUPKIND value to any other BACKUPKIND value. For example, SB can specify that all files with a BACKUPKIND of DISK be created on PACK instead.

The SB command can also convert any BACKUPKIND value to DLBACKUP. This value cannot be specified directly by the BACKUPKIND file attribute; only the SB setting can cause this value to be applied to a backup file. The DLBACKUP value causes the backup file to be created on the family specified by the FAMILYNAME file attribute. If the FAMILYNAME file attribute is null, the backup file is created on the family specified by the BACKUPFAMILY task attribute. If BACKUPFAMILY is also null, the backup file is created on the family specified by the *DL BACKUP ON <family name>* form of the DL (Disk Location) system command.

Titling of Printer Backup Files

MOREBACKUPFILES Option Is Set

If the MOREBACKUPFILES option of the OP (Options) system command is set and no title is specified by the process that created a backup file, the system automatically assigns the backup file a title of the following form:

```
*BD/<mix number>/..../<12-digit file number>/
<internal name> ON <backup family>
```

In this title, the **BD* node indicates that this is a printer backup file. On a system running the Secure Accountability Facility with the USERCODEDBACKUP option set to TRUE, the backup file titles are prefixed with the usercode of the process, rather than an asterisk (*).

The first node of the title is followed by one or more nodes that store mix numbers. The first of these nodes contains the mix number of the originating job or the session number of the originating session. Other mix number nodes, if there are any, contain the mix numbers for other ancestors of the process, in order, from eldest to youngest. The last of the mix number nodes contains the mix number of the process itself. If there is only one mix number node, the backup file was created directly by a job or session. Each mix number node is padded on the left with enough zeros to bring the node to 7 digits in length.

The file number node is a 12-digit number that indicates the chronological order of this backup file compared to other backup files produced by the same process. For example, suppose a process declares a backup file with an INTNAME of A and another with an INTNAME of B. If the process opens and closes A three times, the system creates multiple backup files whose titles end with 000000000000/A, 000000000001/A, and 000000000002/A. If the process then opens B, the system creates a backup file whose title ends with 000000000003/B.

The last node of the title stores a file name. The name is up to 17 characters of the FILENAME file attribute, which can be assigned by the process. If FILENAME is not assigned, FILENAME defaults to the value of the INTNAME file attribute. If INTNAME is not assigned, its value defaults to the file identifier used in the file declaration.

MOREBACKUPFILES Option Is Not Set

If the MOREBACKUPFILES option of the OP (Options) system command is not set and no title is specified by the process that created a backup file, the system automatically assigns the backup file a title of the following form:

```
  *BD/<mix number>/..../<file number><internal name> ON <backup family>
```

In this title, the **BD* node indicates that this is a printer backup file. On a system running the Secure Accountability Facility with the USERCODEDBACKUP option set to TRUE, the backup file titles are prefixed with the usercode of the process, rather than an asterisk (*).

The first node of the title is followed by one or more nodes that store mix numbers. The first of these nodes contains the mix number of the originating job or the session number of the originating session. Other mix number nodes, if there are any, contain the mix numbers for other ancestors of the process, in order, from eldest to youngest. The last of the mix number nodes contains the mix number of the process itself. If there is only one mix number node, the backup file was created directly by a job or session. Each mix number node is padded on the left with enough zeros to bring the node to 7 digits in length.

The last node of the file name stores a file number and a name. The name is the first fourteen characters of the FILENAME file attribute, which can be assigned by the process. If FILENAME is not assigned, FILENAME defaults to the value of the INTNAME file attribute. If INTNAME is not assigned, its value defaults to the file identifier used in the file declaration.

The file number is a 3-digit number that indicates the chronological order of this backup file compared to other backup files produced by the same process. For example, suppose a process declares a backup file with an INTNAME of A and another with an INTNAME of B. If the process opens and closes A three times, the system creates multiple backup files whose titles end with 000A, 001A, and 002A. If the process then opens B, the system creates a backup file whose title ends with 003B.

Note that if a process attempts to create a backup file that would be numbered higher than 999, the system discontinues the process with the error *TOO MANY BACKUP FILES.* The HISTORYTYPE value is 4 (DSEDV), the HISTORYCAUSE value is 8 (SOFTIOERRCAUSEV), and the HISTORYREASON value is 115.

For more information on the OP (Options) system command refer to the *System Commands Operations Reference Manual.*

The backup family is the family determined by the rules discussed under "Storing Printer Backup Files Temporarily" earlier in this section.

Note that the backup file title can be affected by the task attributes discussed under "Other Print-Related Task Attributes" later in this section.

Submitting Print Requests

The PRINTDISPOSITION file attribute controls when the system generates a print request for any given backup file, as discussed under "Storing Printer Backup Files Temporarily" earlier in this section. If PRINTDISPOSITION has the default value of EOJ, then the system creates print requests in the following manner:

- When a job terminates, the system groups the backup files produced by the job and its tasks into print requests. These print requests are groups of all the backup files that can be printed on the same device. The system then queues all the print requests for printing. In this context, a session is treated like a job, and backup files produced by tasks of the session are queued for printing when you end the session.
- For WFL jobs submitted through a MARC or CANDE *WFL* command, the backup files are associated with the session and are queued for printing when the session ends. However, for WFL jobs submitted through a MARC or CANDE *START* command, the backup files are associated with the job and are queued for printing when the job terminates.

Selecting Print Requests

When one of the default printers becomes available, the system chooses one of the queued print requests to be the next print request printed. By default, short print requests are chosen before longer print requests. However, if the BACKUPBYJOBNR operating system option is set, then backup files are printed in order according to job number. The operator can set or reset this system option by using the OP (Options) system command.

Normally, the system removes any backup file from disk once the backup files has been printed. However, the system does not delete the backup file in the following circumstances:

- The SAVEPRINTFILE file attribute is assigned the value TRUE.
- The LOCKEDFILE file attribute is assigned the value TRUE.
- The file resides on a CD-ROM disk.
- The file resides on a disk that is write-protected.

Programmatic Control Over Printing

A program can control the handling of printer output by specifying print attributes and print modifiers. Using these attributes and modifiers, the program can control such issues as

- The location and the device kind of the backup file
- When the backup file is queued for printing
- The printer used
- The time the print request is considered for printing
- The number of copies that are printed

- The portions of the backup file to be printed
- The formatting and translation of printed output

The final values of the print attributes and print modifiers for a backup file are the result of several different factors, most of which are controlled by a programmer. To begin with, each print attribute and modifier has an ultimate default value that is used if no other factor affects the value. The ultimate defaults can be overridden by process defaults. The process defaults are established by the PRINTDEFAULTS task attribute. The PRINTDEFAULTS value consists of a list of print attributes and modifiers and their associated values. The system applies these values to all backup files produced by the process, unless the values are overridden for particular backup files.

The PRINTDEFAULTS value is itself the outcome of several layers of possible assignments. The sources of these assignments include the PRINTDEFAULTS usercode attribute in the USERDATAFILE, the PRINTDEFAULTS attribute of a session, inheritance from a parent process, assignments to the object code file, run-time task equations, and assignments to an active process.

You can override the process defaults for particular backup files by assigning print attributes to the backup file. *Print attributes* is the name given to file attributes that are related to printing, and they are assigned in the same way as any other file attribute. Using print attributes, a process can cause each backup file to be handled differently.

Another option for printing files is to use the WFL *PRINT* statement. You can enter this statement in WFL jobs, in MARC or CANDE sessions, or at an ODT. The PRINT statement is used mainly to print permanent backup files that were created on an earlier occasion. The backup files remain on disk when printing is completed, so they can be reused later.

The PRINT statement can assign print attributes and modifiers for any or all of the backup files printed. These assignments override all previous assignments for the backup files.

A process can affect the print handling for another process by making assignments to the PRINTDEFAULTS task attribute of the process. Where more specific control is needed, you can use the FILECARDS task attribute to specify print attributes for each backup file.

However, the PRINTDEFAULTS and FILECARDS values that are assigned externally can be overridden internally. A process can assign a different value to its own PRINTDEFAULTS task attribute after initiation. Also, file attribute assignments made by the process outside the file declaration override any conflicting assignments made by way of the FILECARDS task attribute.

Other Print-Related Task Attributes

Aside from PRINTDEFAULTS and FILECARDS, the following task attributes are related to printing: BACKUPFAMILY, BDNAME, and OPTION (BACKUP, BDBASE, and NOSUMMARY options only). However, these task attributes were implemented before the current Print System. You can now use various print attributes to achieve effects similar to the effects of most of these task attributes. Print attributes are the preferred method for achieving such print control.

The BDNAME task attribute, if assigned, prevents a backup file from being automatically printed; instead, the file is saved on disk. In addition, BDNAME causes the backup file to be stored under the usercode of the process. The BDNAME value replaces **BD* as the beginning of the file name. However, the remainder of the file name follows the standard backup-file naming conventions.

As we have seen, BDNAME has several effects. You can achieve some of the same effects through the use of several attributes. You can prevent automatic printing by setting the PRINTDISPOSITION print attribute to DONTPRINT. You can assign a file name by setting the USERBACKUPNAME print attribute to TRUE and assigning the desired name to the FILENAME file attribute. The following example shows what these assignments look like in WFL:

```
FILE OUT (PRINTDISPOSITION=DONTPRINT, USERBACKUPNAME=TRUE,
          FILENAME= <file name>)
```

An advantage to using file attributes instead of BDNAME is that the file attributes give you complete control over the backup file name, whereas BDNAME only affects the prefix. On the other hand, this method is admittedly somewhat more complex than using BDNAME. A single BDNAME assignment affects all backup files used by a process, whereas when file attributes are used, separate FILENAME assignments must be made for each backup file. For example, if a process creates multiple backup files by opening and closing the same logical file repeatedly, then the FILENAME value should be changed before each file open operation; otherwise, each time the file is opened, the previous backup file with the same FILENAME is removed.

If BDNAME is assigned a non-null value, the backup file is saved and not printed, regardless of the PRINTDISPOSITION and SAVEPRINTFILE values.

If BDNAME has a non-null value and USERBACKUPNAME is FALSE, then the FILENAME value is ignored. However, if both BDNAME and USERBACKUPNAME are TRUE, then the FILENAME value is used as the file title. If FILENAME was not assigned, then the INTNAME file attribute value is used as the title. If INTNAME was not assigned, then the file identifier is used as the title.

You can use the BACKUPFAMILY task attribute to specify the family where backup files produced by a process are to be stored. You can also assign the family for a backup file by using the FAMILYNAME file attribute. If there is a conflict between FAMILYNAME and BACKUPFAMILY, the FAMILYNAME value takes precedence over the BACKUPFAMILY value.

You can use the BDBASE option of the OPTION task attribute to cause the task to assume some of the characteristics of a job. One of the effects of this option is to cause task backup files to be submitted for printing when the task terminates. If BDBASE is not set, the backup files are not submitted for printing until the task's job terminates. Another method of controlling the timing of print requests is to use the PRINTDISPOSITION print attribute. Assigning PRINTDISPOSITION a value of EOT has the same effect on printing as setting the BDBASE option.

If you set BDBASE, then the PRINTDISPOSITION value is treated as it would be for a job. PRINTDISPOSITION values of EOT and EOJ are synonyms in this case, and both cause backup files to be printed when the task terminates. Other PRINTDISPOSITION values have their usual effect, regardless of whether BDBASE is set.

The BACKUP option of the OPTION task attribute is discussed earlier in this section under "Storing Printer Backup Files Temporarily." The NOSUMMARY option of the OPTION task attribute is discussed under "Controlling Job Summary Printing" in the "Determining Process History" section.

Controlling Message Tanking

Processes can communicate with terminals by way of remote files. *Tanking* is a method the system can use to temporarily store messages that a process writes to a remote file. You can use the TANKING task attribute to specify the default tanking mode for all remote files used by a process. The effects of this task attribute vary, depending on whether or not the terminal that the process writes to is controlled by the Transaction Server.

When a process writes a message to a remote file, the system inserts the message in an output queue. The system transfers messages from the output queue to the remote device as fast as the remote device is able to accept them. If the process writes messages to the remote file faster than the remote device can receive them, then the output queue can become full. If the output queue is full, and the process writes another message to the remote file, then the system can respond by tanking the output. Tanked output is stored in a file called the *tank file* on disk. The system retrieves messages from the tank file and places them in the output queue when space becomes available.

If the output queue is full and tanking is not enabled for the remote file, and the process attempts to write to the remote file, then the process must wait for room to become available in the output queue before the write operation can complete. The result can be a delay in the execution of the process. However, the process does not actually become suspended and does not appear in the W (Waiting Mix Entries) system command display.

The tanking mode for a particular remote file is determined primarily by the file attribute TANKING. To prevent tanking from occurring, you can assign TANKING a value of NONE. To enable tanking, you can assign TANKING a value of SYNC. To enable a process to close the remote file and continue execution while tanked output exists, you can assign TANKING a value of ASYNC. When ASYNC is used and the process closes the remote file, the system continues to transfer messages from the tank file to the output queue until the tank file is empty. For details about the TANKING file attribute, refer to the *File Attributes Programming Reference Manual*.

The default value of the TANKING file attribute is UNSPECIFIED. If the file attribute has this value, then tanking is determined by the TANKING task attribute and the MCS. The TANKING task attribute has the same possible range of values as the TANKING file attribute. Thus, setting the TANKING task attribute to NONE, SYNC, or ASYNC causes these values to be applied to all remote files whose TANKING file attribute is UNSPECIFIED.

If the TANKING file attribute and the TANKING task attribute are both UNSPECIFIED, the MCS controlling the station can set the tanking mode for the remote file. The MCS can do this by way of a parameter to the Station Assignment to File *DCWRITE*. The DCWRITE statement is described in the *DCALGOL Programming Reference Manual*.

For remote files that communicate with terminals controlled by the CANDE MCS, an operator can use the ?TANKING network control command to specify the default tanking mode. The ?TANKING command can specify default values of UNSPECIFIED, NONE, SYNC, or ASYNC.

For remote files that communicate with terminals controlled by the Transaction Server, the effects of the TANKING file and task attributes vary depending on the type of program involved. Three types of application programs can run under the Transaction Server: direct window programs, remote-file programs, and MCS window programs.

Direct window programs communicate with terminals through special Transaction Server structures rather than through remote files. Consequently, the TANKING file attribute and task attribute have no meaning for these programs.

Remote-file programs are programs that communicate through declared or dynamic remote-file windows. Declared remote-file windows are windows that appear in the Transaction Server configuration file and have particular programs associated with them. Dynamic remote-file windows are created by the Transaction Server at run time when a program initiated from a MARC session opens a remote file.

For remote-file programs with a TANKING value of NONE or UNSPECIFIED, the system does not perform tanking for the remote file. If the TANKING value is SYNC or ASYNC, the system performs tanking as if the TANKING value were ASYNC.

It is recommend that you enable tanking for a remote-file program unless the program services only a single terminal. If a remote-file program services multiple terminals and uses a TANKING value of NONE, the program can go into a waiting state when writing output to a terminal. While the program is in a waiting state, it is unable to service input from other terminals. On the other hand, if a remote-file program services a single terminal, it can be reasonable for the program to wait for all output to be displayed before accepting any further input.

An MCS window program is a program that you initiate from a Transaction Server window devoted to a subsidiary MCS. For example, any programs you initiate by entering a RUN command in a CANDE window are considered MCS window programs. For such programs, the system supports the full range of TANKING file attribute and task attribute values: NONE, SYNC, ASYNC, and UNSPECIFIED. The subsidiary MCS, such as CANDE, can specify a tanking mode if the TANKING file and task attribute are both UNSPECIFIED.

In addition to the system-level tanking that has been described up to this point, the Transaction Server-level tanking is provided for the programs that run in a Transaction Server environment. The Transaction Server-level tanking affects direct window programs, remote-file programs, and MCS window programs. The Transaction Server places output messages in the Transaction Server tank file if the messages are being written faster than the station can receive them, or if the messages are sent to a window dialogue that is suspended.

By default, only messages generated for the user's current window dialogue are displayed at the terminal, and all other window dialogues are considered suspended. The user can resume another dialogue by using an ON command to transfer to the dialogue, or by entering a RESUME command that specifies the dialogue. When the window dialogue is resumed, the Transaction Server retrieves tanked messages and sends them to the station.

The Transaction Server-level tanking is a necessary feature in the Transaction Server windowing environment and is performed regardless of the value of the TANKING file and task attributes.

Suppressing Unwanted System Messages

Although system messages are intended to be helpful, there can be situations where you might find it more convenient to suppress the display of certain messages.

Deimplementation warning messages are a good example of this principle. The system issues a deimplementation-warning message for a process when the process uses a feature that has been scheduled for future deimplementation. These warning messages can be very valuable because they help you to identify programs that need to be modified before you can migrate your system to a new release.

However, the system displays these deimplementation warnings each time the program is run. If you run the program frequently, you may see the warning messages more often than you care to be reminded of the pending deimplementation. You can suppress the messages by using the SUPPRESSWARNING task attribute. This attribute enables you to specify a list of warning message numbers or number ranges, as in the following example:

```
RUN OBJECT/PROG;SUPPRESSWARNING = "1,4,8-10";
```

You can learn the identifying number for a message in either of two ways. First, you can note the warning number when it appears in the message itself. For example, after seeing the following message, you might assign SUPPRESSWARNING a value of *156*.

```
WARNING 156: The TCPNATIVESERVICE mnemonic of the SERVICE file attribute
will be deimplemented in software released after April 30, 2002.
```

Second, you can interrogate the TASKWARNINGS task attribute. This task attribute returns the value of the WARNINGS file attribute of the object code file that is being executed. The WARNINGS file attribute, in turn, stores the message numbers for all the warning messages that the system has ever displayed for processes executing code from that object code file. The following is an example of a declaration and statements you can use in an ALGOL program to suppress all previously displayed warning messages:

```
EBCDIC ARRAY WARN[0:999];
REPLACE WARN BY MYSELF.TASKWARNINGS;
REPLACE MYSELF.SUPPRESSWARNING BY WARN;
```

The SUPPRESSWARNING task attribute suppresses warnings for only a single process. You can suppress selected warnings for all processes on the system by using the SUPPRESSWARNING system command. A particular warning is suppressed for a process if either the SUPPRESSWARNING system command or the SUPPRESSWARNING task attribute indicates that the warning is to be suppressed.

You might also find it useful to suppress DISPLAY messages. A process issues a DISPLAY message by executing a DISPLAY statement. DISPLAY messages are used to enable a process to communicate information to the user without actually opening a remote file or ODT file. DISPLAY messages appear in the MSG (System Messages) system command display, at the terminal of the user that initiated the process, and in the system log.

If a process is initiated by a user at a data comm terminal, the DISPLAY messages issued by the process are probably of interest only to that user. The appearance of these messages in the MSG display can be a needless distraction to the system operator. You can eliminate this distraction by setting the DISPLAYONLYTOMCS task attribute to TRUE for the process. When this attribute is TRUE, if the process is initiated from a data comm terminal, DISPLAY messages appear at the originating terminal but do not appear in the MSG display at the ODT.

The DISPLAYONLYTOMCS task attribute affects only the DISPLAY messages generated by a single process. However, you can use the command ?SO DISPLAYONLYTOMCS to establish a default of DISPLAYONLYTOMCS = TRUE for the current CANDE session.

Further, you can use the CANDE control command *?OP + DISPLAYONLYTOMCS* to set a global default of DISPLAYONLYTOMCS = TRUE for all CANDE sessions. For information about CANDE control commands, refer to the *CANDE Operations Reference Manual.*

The SUPPRESSWARNING and DISPLAYONLYTOMCS task attributes enable you to control certain message displays. You can control a broader range of message displays by using the MSC (Message Control) command in MARC. A user with SYSTEMUSER status can use the MSC system command to suppress messages for all processes with a particular usercode, charge code, mix number, or job number. The MSC system command can also suppress selected types of messages, including DISPLAY messages, particular warning messages, or other common messages. For details, refer to the *Menu-Assisted Resource Control (MARC) Operations Guide.* The MSC system command can also be entered from the ODT.

Localization

Localization is the process of tailoring the user interface of a program to users of a particular nation or culture. Two task attributes can assist you in the localization process: the LANGUAGE task attribute and the CONVENTION task attribute.

You can use the LANGUAGE task attribute to specify the language that is used for a process. This task attribute has effects on two levels:

- The system attempts to use the specified language when displaying any system messages generated for the process, such as BOT, EOT, and RSVP messages. The LANGUAGE value has effect only if system messages in the specified language have been installed on your system.
- The specified language becomes the default language for any messages that are displayed by MESSAGESEARCHER statements in an ALGOL or NEWP program. The LANGUAGE value has effect only if a version of the OUTPUTMESSAGEARRAY using the specified language has been bound to the object code file.

You can use the CONVENTION task attribute to specify the conventions for dates, times, and currency used by a process. This task attribute affects processes that use the CENTRALSUPPORT library to format data according to requested conventions. The CONVENTION task attribute specifies a default convention to be used for CENTRALSUPPORT procedure calls. The user process can selectively override this default through parameters to the CENTRALSUPPORT procedures.

For further information about localization, refer to the *MultiLingual System (MLS) Administration, Operations, and Programming Guide*.

Limiting I/O Usage

When a process executes an I/O statement, the central processor must execute some operating system code to initiate the I/O operation. Thereafter, I/O processors (IOPs) and various peripheral devices such as disk drives might all devote varying amounts of time to executing the I/O operation. At the completion of the I/O operation, the central processor executes some I/O finish code.

The I/O time recorded by the system for individual processes includes I/O initiation time as well as actual device time. The accumulated I/O time for a process is stored in the ACCUMIOTIME task attribute. The I/O time for a process is also visible in the output from the TI (Times) system command and in the Major Type 1, Minor Types 2 and 4 (EOJ and EOT) system log entries.

You can use the MAXIOTIME task attribute to set a limit on the amount of I/O time that a process can use. When the ACCUMIOTIME task attribute reaches a value equal to that of MAXIOTIME, the system discontinues the process and displays the error message I/O TIME EXCEEDED.

The main use of the MAXIOTIME task attribute is to ensure that WFL jobs are placed in the proper job queues. For example, suppose there is a high-priority job queue that is intended for jobs that are not very I/O intensive. The system administrator can use the

IOTIME job queue attribute to provide default and limiting values for the MAXIOTIME task attribute of all WFL jobs that use the job queue. If you submit an extremely I/O intensive job through the job queue, the system discontinues the job when it exceeds the MAXIOTIME value. This enforcement of the MAXIOTIME value gives you an incentive to resubmit the job through a different job queue. For an introduction to the subject of job queues, refer to the discussion of WFL in Section 4, “Tasking from Programming Languages.”

It is also possible for the system administrator to limit each person's usage of disk space. Refer to “Limiting Disk Usage” earlier in this section.

Section 10
Determining Process History

Process history consists of information about how a process terminated, the accumulated resource usage of the process, and what actions the process took while it was active. Process history information can help you determine if a program is running as intended, and can help you to locate the source of any problems that arise.

This section describes the uses of various sources of process history information, including termination messages, job summaries, system log entries, history-related task attributes, and program dumps.

Understanding Termination Messages

You can quickly find out how a process terminated by examining the C (Completed Mix Entries) system command display. The following is an example of this display:

```
---Job-Task-Time-Hist---- COMPLETED ENTRIES -----------------
* 1962\3430 11:43 EOT  (LANJ) *LIBRARY/MAINTENANCE
* 2619\3368 11:43 EOT  (ELMER) *OBJECT/MAIL ON PACK
* 3353\3354 11:43 SNTX (ORDS) *BINDER ON SYS37 MCP/FIXSBP ON DPMAST
  3384\3422 11:42 O-DS (JAS) (JAS)MARC WFL
  3384\3423 11:42 P-DS (JAS) (JAS)WFLCODE
  3327\3327 11:42 EOJ  (RALPH) JOB (RALPH)OBJECT/BNATEST ON DPMAST
```

For each entry, the following information is displayed: the job number, the mix number, the time the process terminated, the type of termination, the usercode of the process, and the name of the process (which is usually the object code file title).

If you know the mix number of the completed process, you can use the *<mix number> Y* form of the Y (Status Interrogate) system command to display the completion status of that process only. The following is an example of such a command and the response to the command:

```
7800 Y

   Task 7800 is completed
     7784\7800 15:08 F-DS (JASMITH) (JASMITH)OBJECT/JUNK ON PACK
```

If the process was initiated from a Menu-Assisted Resource Control (MARC) session, then a similar termination message is automatically displayed on the TASKSTATUS screen. The following is an example:

```
12:10 3384\3718 EOT  (ROLLINS)MARC  WFL
```

For a process initiated from a Command and Edit (CANDE) session, abnormal terminations result in a display of the termination type and other process history information. The following is an example:

```
#2316 OPERATOR DSED @ (00000120)
#O-DS @ 00000120.
#ET=3.2 PT=0.1 IO=0.1
```

The first two lines shown in the preceding example would be displayed only for an abnormal termination. These lines give the mix number, the cause of the termination, and the sequence number of the statement the process was executing when it terminated. (The sequence number is replaced by a code address if the program was compiled without the LINEINFO compiler option being set. For information about how to interpret the code address, refer to “Determining Where a Fault Occurred” later in this section.)

The third line in the preceding example is displayed for all terminations (normal or abnormal) of tasks initiated from CANDE. This line gives statistics on the elapsed time, accumulated processor time, and accumulated I/O time for the process.

The CANDE, MARC, and ODT termination messages make use of the same termination type abbreviations. Of these, the following indicate normal terminations:

EOJ	The process was a job that terminated normally.
EOT	The process was a task that terminated normally.
SNTX	The process was a compilation that encountered syntax errors. The process terminated normally, but no object code file was created.

For processes that terminated abnormally, you can get a general idea of the cause of the termination by examining the DS message (P–DS, O–DS, and so on). Table 10–1 lists the abnormal termination messages and their meanings.

For a more specific indication of why a process terminated abnormally, you need to examine the values of the history-related task attributes (HISTORYTYPE, HISTORYCAUSE, and HISTORYREASON). Table 10–1 lists the HISTORYTYPE and HISTORYCAUSE values corresponding to each DS message. For an introduction to history-related task attributes, refer to “Determining the Type of Termination” later in this section. For a list of the possible values for history-related task attributes, refer to the *Task Attributes Programming Reference Manual.*

Table 10–1. Abnormal Termination Messages

Message	HISTORYTYPE	HISTORYCAUSE	Meaning
A-DS	8	0	The process was a Work Flow Language (WFL) job whose initiation failed because the job attribute list included an invalid task attribute assignment; or, the process was discontinued, but is now executing an EPILOG procedure.
D-DS	4	6	The process encountered a data comm error.
E-DS	4	12	The process encountered an Enterprise Database Server error.
F-DS	4	4	The process requested a machine operation that could not be executed. Examples are dividing by zero or reading past the end of an array.
I-DS	4	7-9	The process encountered an I/O error.
N-DS	4	13	The process encountered a BNA error.
O-DS	4	1	The process was discontinued by an operator command.
P-DS	4	2	The process attempted an illegal action or deliberately set its STATUS task attribute to TERMINATED, or was terminated because its parent terminated.
Q-DS	7	0	The process was a job that did not qualify for any job queue.
R-DS	4	3	The process exceeded a resource limit, such as MAXPROCTIME.
S-DS	4	5	The process violated system parameters or a system hardware fault occurred.
U-DS	4	10-11	The process was discontinued by an unknown cause.
?-DS	4	0	The process was discontinued by an unknown cause.

Using Log Information

The system records the activity of each process in two types of logs:

- The system log (SUMLOG)

  This central log stores information about all kinds of actions on the system.

- Job logs

  A separate job log is created for each job on the system and is stored in the job's job file. The system creates a job log for WFL jobs and other independent processes, as well as for CANDE and MARC sessions. The job log contains information about the job (or session) and its descendant tasks. Depending on the values of various task attributes and system options, the system might create a printout of the job log, called the *job summary*.

The following subsections explain how the programmer and system operator can control the contents of these logs and the generation of reports from these logs.

Specifying the Information to Be Logged

An operator can use the LOGGING (Logging Options) system command to select the major and minor log entry types that are to be logged. You can specify that a particular type of log entry is to appear in the job log, in the system log, in both, or in neither. The following LOGGING command causes Major Type 1, Minor Type 5 (File Open) entries to appear in job logs, and Major Type 1, Minor Type 6 (File Close) entries to appear in the system log:

```
LOGGING 1,5 JOBFILE ALL;1,6 SUMLOG ALL;
```

You can use the DEPTASKACCOUNTING task attribute and the FILEACCOUNTING task attribute to control certain types of logging. These task attributes affect the system log and the job log equally. You can use DEPTASKACCOUNTING to prevent the system from generating log entries to record the initiation and termination of a dependent process. You can use the FILEACCOUNTING task attribute to prevent the system from generating log entries to record file open and close actions. You can create defaults for these task attributes on a system wide basis with the ACCOUNTING (Resource Accounting) system command. You can create defaults for these task attributes on a usercode basis through assignments to the usercode attributes with the same names in the USERDATAFILE.

You can use either of two system commands to log comments about the history of a particular process. The LC (Log Comment) system command enters a comment in the system log only. The LJ (Log to Job) system command enters a comment in both the system log and the job log of a particular job. The following is an example of this command:

```
3335 LJ JOB RAN NORMALLY
```

You can use the NOJOBSUMMARYIO task attribute to suppress the logging of information in the job log. If NOJOBSUMMARYIO is set, no entries are written to the job log, except for the Major Type 1, Minor Type 1 (BOJ) entry or the Major Type 4, Minor Type 1 (Log-on) entry. NOJOBSUMMARYIO can also be set and reset throughout a job to prevent selected parts of the job from appearing in the job log. Using NOJOBSUMMARYIO saves I/O time and thus allows the job to run more efficiently.

You can use the LG (Log for Mix Number) system command and the LOGSELECT usercode attribute to enable logging of selected types of events for a particular usercode. These features enable the system administrator to monitor the activities of a particular user who might be committing some type of security breach. These features affect the system log only.

Controlling Job Summary Printing

The printing of job summaries is controlled primarily by the JOBSUMMARY task attribute. To cause job summary printing, you can assign a value of UNCONDITIONAL, and to prevent job summary printing, you can assign a value of SUPPRESSED. To cause conditional printing of job summaries, you can use either of two values: ABORTONLY or CONDITIONAL. Either of these values causes job summary printing if the job or any of its tasks terminate abnormally. The difference between the two values is that the CONDITIONAL value also causes job summary printing if the job has any printer backup files associated with it or if a compiler task encounters syntax errors.

If the JOBSUMMARY task attribute has a value of DEFAULT, then job summary printing is controlled by either of two types of defaults.

- If the NOSUMMARY option of the OPTION task attribute is set, then a JOBSUMMARY task attribute of DEFAULT is interpreted as CONDITIONAL.
- If the NOSUMMARY option of the OPTION task attribute is reset, then the Print System JOBSUMMARY option controls the job summary printing. The Print System JOBSUMMARY option is set or reset through the PS DEFAULT system command. The JOBSUMMARY option can specify any of the following values: CONDITIONAL, UNCONDITIONAL, SUPPRESSED, or ABORTONLY.

A job summary can be printed for any WFL job that compiled successfully, as well as a job that failed to compile. This is true even if the job never ran because no job queue would accept it or because an operator discontinued the job while it was queued.

The conversion of job logs into job summaries for printing is handled by visible independent runners with the name JOBFILE/CONVERTER. These processes are visible in the mix so that you can monitor their resource usage.

Saving the Job Summary File

You can use the JOBSUMMARYTITLE task attribute to cause the job summary file to be saved as a permanent disk file.

If the JOBSUMMARYTITLE value is a null string (the default value), then the system creates a job summary file only if a job summary is to be printed. If the system does create a job summary file, the system removes the file once it is printed. The job summary file title can have two forms. If the MOREBACKUPFILES option of the OP (Options) system command is set the following form is used:

```
*BD/<7-digit job number>/000000000000/SUMMARY
```

If the MOREBACKUPFILES option of the OP (Options) system command is not set, the following form is used:

```
*BD/<7-digit job number>/000SUMMARY
```

For more information on the OP (Options) system command, refer to the *System Commands Operations Reference Manual*.

If you assign a file title to JOBSUMMARYTITLE, then the system creates a job summary file with the specified file title. The job summary file remains on disk, whether or not the system prints the job summary. You can use a Command and Edit (CANDE), Menu-Assisted Resource Control (MARC), or WFL *PRINT* command to print out the job summary file later. For a description of the PRINT command, refer to the *Print System and Remote Print System Administration, Operations, and Programming Guide*.

Analyzing the System Log

You can use the LOGANALYZER utility to obtain a detailed report of the history of a particular process. You can invoke LOGANALYZER by using the LOG command from CANDE, MARC, WFL, or an ODT. The LOG command causes the current system log to be searched, unless the title of an old system log is specified.

The following command displays all log entries for the job with mix number 7483 and for all descendants of that job:

```
LOG JOB 7483
```

The following command displays all log entries for the process with mix number 8923:

```
LOG MIX 8923
```

You can specify various options to limit the types of entries that are displayed for the process and to direct the output to an ODT, a remote terminal, or a printer. For details, refer to the discussion of LOGANALYZER in the *System Software Utilities Operations Reference Manual.*

Programmatically Interrogating Process History

After a task terminates, the task variable associated with it continues to exist until the parent exits the block that contains the task variable declaration. As long as the task variable exists, the parent can use it to interrogate the final task attribute values of the task. By interrogating history-related task attributes, the parent can find out whether the task terminated normally.

On the other hand, the history of a job cannot be interrogated through task attributes. The task variable of a job can be accessed only by the job itself and its descendants, and the descendants cannot survive the termination of the job.

Determining the Type of Termination

Several task attributes and a special WFL expression are available for determining how a task terminated. The relevant WFL expression is the *task state inquiry*. This expression can be used to determine whether termination was normal or abnormal. For example, the following WFL statement causes a specified action to be taken if the task terminated abnormally:

```
IF TSK ISNT COMPLETEDOK THEN
```

Another way to determine whether a task terminated normally is to inspect the HISTORYTYPE task attribute. A HISTORYTYPE value of NORMALEOTV indicates that the termination was normal. A value of DSEDV indicates that termination was abnormal. Most of the other values indicate that the process has not yet terminated and give an indication of its current state.

If termination was abnormal, the HISTORYCAUSE task attribute can be interrogated to determine the general type of abnormal termination that occurred. For example, a value of OPERATORCAUSEV indicates that an operator command discontinued the process and a value of DCERRV means that the process was discontinued because of a data comm error.

A more detailed account of why a task terminated abnormally is stored in the HISTORYREASON task attribute. For example, suppose the HISTORYCAUSE value is RESOURCECAUSE, meaning that a resource limit was exceeded. The HISTORYREASON value might be PROCESSEXCEEDEDV, which means specifically that the processor time limit was exceeded.

The system uses the HISTORYTYPE and HISTORYCAUSE values to determine what termination message to display for a process. The correspondence between these task attributes and the termination messages is shown in Table 10–1.

Determining Whether a Compilation Was Successful

You can use any of several methods to determine programmatically whether a particular compilation uncovered syntax errors in the source program.

For a compilation initiated from WFL, the task state expression can be used to determine whether the compilation was successful. To use this expression, a task variable must first be associated with the compilation in the COMPILE statement. The following is an example:

```
COMPILE OBJECT/PROG WITH ALGOL [COMPILETASK] LIBRARY;
IF COMPILETASK IS COMPILEDOK THEN
   RUN SYSTEM/XREFANALYZER (0);
```

Another way to determine whether the compilation was successful is to interrogate the HISTORYTYPE task attribute of the compilation. A value of NORMALEOTV means that the compilation was successful, but a value of SYNTAXERRORV means that syntax errors were found.

Another method that can be used is to interrogate the TASKVALUE task attribute. TASKVALUE has a value of zero if the compilation was successful or one if syntax errors were found.

Responding to Task Failures

WFL includes a special statement that specifies actions to be taken if any offspring terminates abnormally. This is the *ON TASKFAULT* form of the ON statement. The *ON TASKFAULT* statement remains in effect for the remainder of the job unless overridden by another *ON TASKFAULT* statement. For example, a WFL job could include the statement *ON TASKFAULT, ABORT*. This statement causes the job to terminate abnormally when any of its offspring terminates abnormally.

Determining Where a Fault Occurred

You can use the STACKHISTORY task attribute to determine the statement that was being executed and the procedures that had been entered when a process terminated abnormally. To understand the value returned by this attribute, you must have compiled the program that was being executed with one or both of the following compiler options set: LINEINFO and LIST.

Setting LINEINFO causes the STACKHISTORY value to include the sequence number for each of the relevant statements in the source program. Setting LIST causes the compiler to produce a printout of the source program that includes code addresses for each line. The code addresses are needed to interpret the STACKHISTORY value if LINEINFO was not set.

The following is an example of the source program printout for a program that was compiled with the LIST and LINEINFO options set. (The example has been condensed horizontally to fit on the page.)

```
BEGIN                                             00000200  000:0000:0
                                                BLOCK#1 IS SEGMENT 0003
                                              1   00000300  003:0000:1
                                                  00000400  003:0000:1
REAL X, Y;                                        00000500  003:0000:1
PROCEDURE ONE;                                    00000600  003:0000:1
BEGIN                                             00000700  003:0000:1
  PROCEDURE TWO;                                  00000800  003:0000:1
                                                    ONE IS SEGMENT 0004
                                              2   00000900  004:0000:1
  BEGIN                                           00001000  004:0000:1
     Y:= X DIV 0;                                 00001100  004:0000:1
  END;                                        3   00001200  004:0001:2
                                              3   00001300  004:0001:3
  TWO;                                            00001400  004:0001:3
END;                                              00001500  004:0002:1
                                         ONE(004) LENGTH IN WORDS IS 0005
                                              2   00001600  003:0000:1
                                                  00001700  003:0000:1
ONE;                                              00001800  003:0000:1
                                                  00001900  003:0000:5
END.                                              00002000  003:0000:5
                                     BLOCK#1(003) LENGTH IN WORDS IS 0006
```

In this example, each line of source code ends with the sequence number and code address of the line. The code address is divided into three parts by colons; the first part is the code segment number, the second is the word number, and the third is the syllable number. The numbers in the code address are in hexadecimal format.

If a process terminates normally, the STACKHISTORY value is a null string. However, if the process terminates abnormally, STACKHISTORY returns a value such as the following:

```
004:0000:5 (00001100), 004:0002:1 (00001400), 003:0000:5 (00001800).
```

This value gives the code address and sequence number for the statement that was being executed when the process terminated and for each procedure invocation statement that was in effect when the process terminated. Thus, the value in this example indicates that the statement at line 1100 was being executed when the process terminated, and that procedure invocation statements at lines 1400 and 1800 were in effect. The sequence numbers are included in the STACKHISTORY value because the program was compiled with LINEINFO set.

If LINEINFO is *not* set, then STACKHISTORY returns the following value:

```
004:0000:5, 004:0002:1, 003:0000:5.
```

The numbers in this example give a somewhat less exact idea of the locations of the statements that were being executed when the process terminated. Each statement usually occurs on the line preceding the specified code address. The address 004:0000:5 does not appear in the printout, but the statement occurs on the next lower-numbered line: 004:0000:1.

Note that if the object code file was produced by the Binder, you must use some extra care in interpreting the code addresses or sequence numbers returned in the STACKHISTORY value. When the Binder produces a bound object code file, the Binder changes the code segment numbers for statements in the subprogram. Fortunately, if you use the Binder option LIST, the Binder produces a printout that lists such changes. The following is an example of such a printout:

```
                      O B J E C T / A L G O L / B I N D   O N   D I S K
                        = = = = = = = = = = = = = = = = = = = = = = = = = = = =

  HOST IS OBJECT/ALGOL/HOST;                                        00001270
  BIND PRINTIT FROM OBJECT/ALGOL/SUB;                               00001272
  STOP;                                                             00001274
  BEGIN BINDING PRINTIT OF BLOCK#1 FROM OBJECT/ALGOL/SUB
        PRINTIT  (02,0006) CHANGED TO (02,0006)
  K  <-- NEW GLOBAL ADDED TO HOST - WARNING ONLY
        K  (02,0004) CHANGED TO (02,0008)
        LINE  (02,0005) CHANGED TO (02,0003)
        J  (02,0003) CHANGED TO (02,0005)
        BUFFER (02,0002) CHANGED TO (02,0004)
        ?010  (01,0006) CHANGED TO (01,0006)
        <SEG DICT ITEM>   (01,0002) CHANGED TO (01,0005) = 03 000001300001
        <SEG DICT ITEM>   (01,0003) CHANGED TO (01,0007) = 05 07000000005F
        <SEG DICT ITEM>   (01,0004) CHANGED TO (01,0008) = 05 080000540002
  END OF BINDING PRINTIT
```

Note the three lines near the bottom of the list that begin with "<SEG DICT ITEM>." These list changes to the address couples for code segments in the subprogram. The second number in each address couple is the offset, which is the same as the code segment number for that code segment. The list informs us that code segment number 2 was changed to 5, 3 was changed to 7, and 4 was changed to 8. Therefore, you should look at the STACKHISTORY value for code addresses that begin with 5, 7, or 8, and make a note that they really begin with 2, 3, or 4, respectively. Then you can look for the code addresses in the compiler listing that was created when you originally compiled the subprogram. Suppose that the STACKHISTORY value is as follows:

```
  005:000F:1, 003:0017:3.
```

You should translate the first address into 002:000F:1, and then look at the compiler listing of the subprogram to determine which statement had that code address. The second address does not begin with 5, 7, or 8, so you don't need to translate it. You can find the procedure invocation referred to by the second address at 003:0017:3 in the compiler listing for the host program.

If the host program and the subprogram were compiled with the LINEINFO compiler option set, and the Binder was run with the LINEINFO Binder option set, then the bound object code file contains sequence numbers that appear in the STACKHISTORY value. The Binder does not change the sequence numbers of the host program or the subprogram. When interpreting the sequence number, beware of the possibility that the same sequence number occurred in both the host program and the subprogram file. For example, suppose that the following is the STACKHISTORY value:

```
005:000F:1 (00000750), 003:0017:3 (00001350).
```

The subprogram and the host program both might contain lines with the sequence number 750, and they also both might contain lines with the sequence number 1350. However, the last address in the STACKHISTORY value always refers to a statement in the host program. At line 1350 in the host program listing is a procedure invocation statement. If this statement invokes the bound-in procedure, then line 750 is found in the subprogram listing. Otherwise, line 750 is found in the host program listing.

Another task attribute that provides information related to process history is the STOPPOINT task attribute. This real-valued attribute has fields defined that store the fault reason and the code address. The fault reason is the same as the value returned by the HISTORYREASON task attribute, and the code address is the same as the first code address of the STACKHISTORY value.

Protecting a Process from Abnormal Terminations

You can use a variety of different methods to prevent a process from being terminated abnormally. Depending on the method you choose, you can

- Trap faults and recover from them.
- Protect a process from operator-entered DS (Discontinue) system commands.
- Retry a process that was terminated by a fault.
- Protect a process from all possible faults, errors, and termination conditions.

You can also specify code to perform cleanup functions when a process is forced to terminate abnormally.

Preventing All Abnormal Terminations

ALGOL and NEWP programs can use the TRY statement to protect processes against abnormal terminations.

Before the implementation of the TRY statement, the best-known method of protection was the ON statement in ALGOL and NEWP. The ON statement is discussed under "Protecting a Process from Faults" later in this section. Compared to the ON statement, the TRY statement provides the following advantages:

- Protection against a broader range of otherwise fatal conditions.
- Alternate sets of error handling code to be executed if the error handling code itself incurs an error.
- Faster execution.

Protection against Most Conditions

The basic form of the TRY statement is

```
TRY <statement> ELSE <statement>;
```

TRY executes the first statement or expression. If an error occurs during execution, TRY then executes the statement or expression in the ELSE clause. Control then continues to the next statement after the TRY statement.

This form of the TRY statement protects processes against most termination conditions, including the following:

- Faults detected by the hardware, such as divide-by-zero conditions. (These faults could also be trapped by an ON statement in ALGOL or NEWP.)
- Untrapped I/O errors.
- Security errors. However, the system ignores the TRY error-handling code and terminates the process if the SAVEVIOLCOUNT usercode attribute is set and the error causes the VIOLATIONCOUNT usercode attribute to exceed the value of the VIOLATIONLIMIT usercode attribute.
- Task attribute errors.
- Library linkage errors caused by an attempt to invoke an imported procedure that is not available for one of the following reasons:
  - The procedure is not supplied by the linked library.
  - The procedure is in a library that could not be linked.
  - The procedure is in an unlinked library with AUTOLINK set to FALSE.
- Delinkage of a library by a DELINKLIBRARY statement with the ABORT option. Normally, the ABORT option causes the system to discontinue any processes that are executing a procedure imported from the linked library. To protect processes against this condition, the TRY statement must be invoked before the library procedure call. For an example, refer to "TRY Statements and Library Delinkage" later in this section.

- Operator DS (Discontinue) commands applied to a process waiting on an RSVP, ACCEPT, or unit clear action. (An operator can still discontinue the process by entering a second DS command.)
- Assignments of MYSELF.STATUS to VALUE(TERMINATED).

The TRY statement does not handle exception conditions that would not result in process termination. Such conditions include bad GO TO statements and exponent underflow faults. Such conditions can be trapped by an ON statement or EXCEPTION procedure, however.

Protection against All Conditions

You can use the unsafe form of the TRY statement to extend protection against all possible causes of termination.

Note: *The unsafe form should be used with great care and only by programmers familiar with the operating system. In particular, it is important that unsafe TRY code execute quickly, so it does not monopolize system locks. It might also be necessary for the TRY code to cut back the process stack to prevent stack overflow errors. Misuse of the unsafe form can result in hung programs, system dumps, or halt/loads.*

The unsafe form of the TRY statement is distinguished by the use of the PROTECTED clause, as follows:

```
TRY [:PROTECTED ] <statement> ELSE <statement>;
```

The PROTECTED clause of the TRY statement is only available in NEWP and DMALGOL, and in NEWP is only permitted in blocks marked with the UNSAFE block directive. The compiler marks the resulting object code file as nonexecutable. To enable execution of the object code file, an operator can use either of the following commands:

- The *MP <file title> + EXECUTABLE form* of the MP (Mark Program) system command. This command removes the nonexecutable status so that the object code file can be initiated with the normal process initiation statements such as CALL, PROCESS, and RUN.
- The *SL <function name> = <file title>* form of the SL (Support Library) system command. This command enables a nonexecutable object code file to be initiated by the library linkage mechanism when the library is linked to by function.

Note: *An unsafe DMALGOL program is only nonexecutable if the security option DMALGOLUNSAFE is set.*

The conditions protected against by the PROTECTED clause include:

- Operator DS (Discontinue) commands applied to a normally executing process.
- PARENT PROCESS TERMINATED conditions.
- STATUS = VALUE(TERMINATED) assignments made by a different process.

- Cancellation of a library linkage through a DELINKLIBRARY statement with the ABORT option. Normally, the ABORT option causes the system to discontinue any processes that are executing a procedure imported from the linked library. The unsafe TRY statement protects processes against this type of condition, even if the TRY statement is invoked after the library procedure call. For an example, refer to "TRY Statements and Library Delinkage" later in this section.
- GO TO statements that attempt to exit outside the TRY statement.

Note: *By default, the error-handling statement in the ELSE clause is not protected. To protect the error-handling statement, you could use multiple ELSE clauses as discussed under "Providing Alternate Sets of Error Handling Code." Alternatively, you could protect the error-handling statement from interrupts with unsafe constructs such as the CONTROLSTATE block directive in NEWP or the DISALLOW statement in DMALGOL. You should use these unsafe constructs only to protect short code streams that do not include any procedure calls.*

Including Multiple Statements

The TRY statement can include compound statements or procedure calls. For example, the following statement form is valid:

```
TRY
   <procedure call>
ELSE
   BEGIN
    <multiple statements>
   END;
```

Determining the Cause of the Error

The error handling code in a TRY statement can determine the cause of the error by inspecting the values of the task attributes PRIORHISTORY, PRIORHISTORYTYPE, PRIORHISTORYCAUSE, and PRIORHISTORYREASON. These task attributes store the values that HISTORY, HISTORYTYPE, HISTORYCAUSE, and HISTORYREASON would have had if the process had been terminated by the error.

The following example shows how TRY statement error code could use CASE statements to interrogate the PRIORHISTORYCAUSE and PRIORHISTORYREASON task attributes:

```
TRY [: PROTECTED ]
  <statement or expression>
ELSE
  BEGIN
  CASE MYSELF.PRIORHISTORYCAUSE OF
    BEGIN
      0: % If PROTECTED, this value indicates a GO TO
         % is exiting to outside the TRY statement.
         % Perform any necessary cleanup.
         LIBERATE(E1);
         VALUE(OPERATORCAUSEV):
            CASE MYSELF.PRIORHISTORYTYPE OF
            BEGIN
            VALUE(JUSTDSEDV): DISPLAY("DO NOT ATTEMPT TO DS ME");
                 VALUE(RSVPV): DISPLAY("WRONG ANSWER");
            END;
         VALUE(PROGRAMCAUSEV):
            CASE MYSELF.PRIORHISTORYREASON OF
            BEGIN
                 VALUE(DEATHINFAMILYV):
                 VALUE(CRITICALBLOCKV): LIBERATE(E1);
            END;
         ELSE:
       END;
     END;
```

Using TRY as an Expression

When you use TRY to invoke an expression or to make a typed procedure call, TRY itself is treated as an expression that returns a value. For example, the following statement invokes typed procedure PROCA and stores the result in RSLT. If PROCA incurs an error, a value of -1 is stored in RSLT.

```
RSLT := TRY PROCA ELSE -1;
```

Providing Alternate Sets of Error Handling Code

You can protect the error handling code itself by specifying additional ELSE clauses. For example, a TRY statement could have the following form:

```
TRY <statement> ELSE <statement> ELSE <statement>;
```

The example first executes the statement following the TRY verb. If an error occurs in that statement, this example executes the statement following the first ELSE verb. If an error occurs in that statement, this example executes the statement following the second ELSE verb. If an error occurs in the final statement, the program is not protected from the error.

Optimizing Performance

On some machines, the TRY statement might be executed more quickly if you use the following limited form of the statement:

```
TRY [ <error procedure identifier> ] <procedure invocation or reference>
```

This limited form of the TRY statement provides more limited protection than the normal form. Unlike the normal form, the limited form does not begin to protect the process until the procedure begins execution. Thus, if an error occurs in passing parameters to the procedure, or if the procedure invoked is a null procedure reference, the process is not protected from the error. (However, on some machines the limited form does protect the process if the process invokes an imported procedure that is not available. This protection is provided on all machines *except* the A7, A2100, and NX4200.

If the procedure fails to finish normally, the error procedure is automatically invoked. No protection is provided if the error procedure also fails to finish normally.

Triggering Program Dumps

TRY error-handling code can include statements that invoke program dumps. However, such dumps might not be useful because the system cuts back the process stack to the point where the TRY statement occurs before executing the error-handling code. Thus, if the TRY statement invoked one or more procedures, then any variables declared by those procedures would be lost before the error-handling code invokes the program dump.

To obtain a more useful program dump, you can set the FAULT and DSED bits in the OPTION task attribute when the process is first initiated. (For an explanation of the difference between the FAULT and DSED bits, refer to "Understanding Internal and External Causes" later in this section.) When these bits are set, and a termination condition occurs, the system initiates a program dump. After the program dump finishes, the system cuts back the process stack to the TRY statement and invokes the error-handling code.

Protecting a Process from Faults

A fault is an invalid action that is detected by the hardware, such as an attempt to divide by zero. In general, a process is discontinued if it encounters a fault. However, ALGOL and NEWP provide a feature that can be used to allow the process to continue normal execution after most faults. The *ON* statement specifies actions to be taken if a fault occurs. In addition, the ON statement can be used to interrogate the type of fault and the stack history. The stack history value returned is identical in format to that returned by the STACKHISTORY task attribute.

The ON statement can specify which particular fault types it is to handle, or can include the ANYFAULT clause to handle all eligible faults. Certain faults, such as stack overflows, cannot be handled by the ON statement. If any fault occurs that can be handled, the following ON statement stores the stack history into array FAULTARRAY and the number into FAULTNO. The statement then invokes the procedure HANDLEFAULTS, passing the fault number to it as a parameter:

```
ON ANYFAULT [FAULTARRAY:FAULTNO], HANDLEFAULTS(FAULTNO);
```

In addition, the C language supports a *signals* mechanism that can be used for fault handling. The signals mechanism is among the C language extensions that support the POSIX environment. For an overview of signals, refer to the *POSIX User's Guide*.

Retrying a Failed Task

You can design a process to be restarted automatically if it is terminated because of an error. This effect is achieved by assigning a value to the RESTART task attribute. The value of this task attribute specifies the number of times the process is to be restarted following an error termination. Execution of the restarted process begins with the first statement in the outer block. After each restart, the RESTART task attribute value is reduced by one. When the RESTART value is zero, the next error termination is final.

When the process is restarted, no “EOJ” or “EOT” messages are displayed. Some elements of the process survive the error termination and they are reused. These elements are the PIB, the code segment dictionary, and the base of the process stack. All task attribute values of the original process are retained, including the mix number. In addition, the values of any parameters the process received from its initiator are saved.

However, the values of all objects, including all variables and arrays, declared by the process are lost. These objects must be re-created and reinitialized after the process restarts.

If the process has offspring tasks, they are discontinued with a “PARENT PROCESS TERMINATED” error each time the process has an error termination. However, each time the process is restarted, it can execute the task initiation statements again and create new tasks.

A process that was discontinued by an operator command does not restart, regardless of the value of the RESTART attribute. The RESTART value also does not cause a process to be restarted after a halt/load. (It is true that WFL jobs restart after a halt/load, but this feature is not related to the RESTART task attribute. For information about WFL job restarts, refer to Section 11, “Restarting Jobs and Tasks.”)

The RESTART task attribute is primarily useful in situations where the process might be discontinued by a temporary hardware fault or where the process will receive different input data after it restarts. If the process is attempting to do something invalid or contradictory, repeated restarts are not helpful. The process terminates abnormally each time.

If the process includes a statement that assigns a value to RESTART, make sure that the statement is not reexecuted after each restart. If the statement is always reexecuted, then the value of RESTART can never reach zero, and the process restarts infinitely. The following example shows an ALGOL program that would enter such a loop:

```
100 BEGIN
200 REAL X;
300 MYSELF.RESTART:= 4;
400 X:= X DIV 0;
500 END.
```

The following example shows how the program could be modified so that it would not enter an infinite loop. The SW1 task attribute is used as a flag to indicate whether the process has been executed at least once.

```
100 BEGIN
200 REAL X;
300 IF NOT MYSELF.SW1 THEN
400     MYSELF.RESTART:= 4;
500 MYSELF.SW1:= TRUE;
600 X:= X DIV 0;
700 END.
```

In ALGOL or NEWP, you can use the ON statement to prevent an abnormal termination from occurring after a program fault. The ON statement has fewer applications than the RESTART task attribute because it applies only to errors that would otherwise cause the process to be discontinued with HISTORYCAUSE = FAULTCAUSE. However, for these cases, the ON statement provides more flexible error handling than the RESTART task attribute.

For more information about the *ON* statement, refer to “Protecting a Process from Faults” earlier in this section.

Protecting a Process from Operator DS (Discontinue) Commands

You can protect a process from operator-entered DS (Discontinue) commands by using the following form of the MP (Mark Program) system command:

```
MP <object code file title> + LOCKED
```

The effects of this command carry over each time the program is run.

You can also use the TRY statement to protect a process against DS commands, as described under “Preventing All Abnormal Terminations” earlier in this section.

Protecting Procedures from DS and ST Commands

The ISOLATED procedure attribute specifies that when a procedure runs on a different process stack than the stack that declares the procedure, it is protected from DS (Discontinue) and ST (Stop) commands that are applied to the second stack.

The stack that declares the procedure is the task or library started when the code file containing the procedure code is initiated. In library procedures, this stack is called the library stack.

Since an isolated procedure can cause DS and ST commands to be delayed, ensure that the isolated procedure runs quickly and efficiently without consuming excess system resources.

For a full description of isolated procedures, refer to "Using Isolated Procedures in Libraries" in Section 18.

Performing Cleanup during an Abnormal Termination

You can use EPILOG or EXCEPTION procedures to perform cleanup if a process is terminated abnormally.

Using EPILOG Procedures

An EPILOG procedure is a special type of procedure that is available only in ALGOL, DCALGOL, DMALGOL, and NEWP. An EPILOG procedure is executed whenever the block that declares it is exited, even if the exit was caused by the process being discontinued. The EPILOG procedure can be designed to perform cleanup actions, such as liberating or causing an event.

EPILOG procedures can be declared in procedures, structure blocks, or connection blocks. The use of EPILOG procedures in connection blocks is discussed under "Using PROLOG and EPILOG Procedures" in Section 18, "Using Libraries." The use of EPILOG procedures in structure blocks is discussed in the *ALGOL Programming Reference Manual, Volume 1: Basic Implementation.* The remainder of this discussion assumes that the EPILOG procedure is declared in a procedure.

An EPILOG procedure can determine whether the procedure exit is normal, or whether the process is being discontinued, by inspecting the STATUS, HISTORYTYPE, HISTORYCAUSE, and HISTORYREASON task attributes of the MYSELF task variable. You can design the EPILOG procedure to take different actions, depending on whether the block exit is normal.

Note that the EPILOG procedure can be discontinued and, thus, prevented from completing all its cleanup functions. For example, if you enter two DS commands for a process, the first causes the EPILOG procedure to be entered. The second DS command discontinues the EPILOG procedure if it has not yet finished execution. This problem should rarely occur if the EPILOG procedure is kept brief.

Using EXCEPTION Procedures

If you need to ensure that certain actions are always performed when a procedure is exited abnormally, you can use an EXCEPTION procedure instead of an EPILOG procedure. EXCEPTION procedures are available in ALGOL, DCALGOL, DMALGOL, and NEWP. These procedures serve a similar function to EPILOG procedures. However, an EXCEPTION procedure is executed only if the block that declares it is exited abnormally, whereas an EPILOG procedure is executed even if the block exit is normal. Block exits are considered abnormal in either of the following cases:

- The block is exited because of a bad GO TO statement. A bad GO TO statement transfers control to a label outside the block and can occur in the block that declared the EXCEPTION procedure, or in a nested block within that block.
- The block is exited because the process was discontinued, because of either an operator DS (Discontinue) system command or an internal fault.

Another important feature of EXCEPTION procedures is that you can prevent them from being interrupted. To do this, you simply add the PROTECTED clause to the EXCEPTION procedure declaration. If the block that declares a protected EXCEPTION procedure is exited abnormally, then the EXCEPTION procedure executes in protected mode. A protected EXCEPTION procedure cannot be interrupted by the DS (Discontinue) or ST (Stop) system command, or by stack stretches.

The PROTECTED form of the EXCEPTION procedure is only available in NEWP and DMALGOL, and in NEWP is only permitted in blocks marked with the UNSAFE block directive. The compiler marks the resulting object code file as nonexecutable. To enable execution of the object code file, an operator can use either of the following commands:

- The *MP <file title> + EXECUTABLE* form of the MP (Mark Program) system command. This command removes nonexecutable status so that the object code file can be initiated with the normal process initiation statements such as CALL, PROCESS, and RUN.
- The *SL <function name> = <file title>* form of the SL (Support Library) system command. This command enables a nonexecutable object code file to be initiated by the library linkage mechanism when the object code file is linked to by function.

Note: *An unsafe DMALGOL program is only nonexecutable if the security option DMALGOLUNSAFE is set.*

If you want an EXCEPTION procedure to be executed before any block exit, normal or abnormal, you can include an explicit call on the EXCEPTION procedure in the block. The following is an example:

```
700 PROCEDURE P1;
710   BEGIN
720   FILE MYFILE(KIND=DISK);
730   PROTECTED EXCEPTION PROCEDURE CLEANUP;
740     BEGIN
750     CLOSE(MYFILE,LOCK);
760     END;
    .
    .
    .
900   CLEANUP;
910   END;
```

The vertical ellipses in this example denote lines that are omitted because they are not essential to the point being illustrated. If P1 exits normally, then the EXCEPTION procedure CLEANUP is explicitly invoked by the statement at line 900. Note that in this case, CLEANUP is executed without protected status. If P1 exits abnormally, the system automatically invokes CLEANUP and executes it with protected status.

Interactions between Error Recovery Statements

A single process could include two or more of the error recovery and cleanup mechanisms mentioned in this section. For example, a process might include ON statements, TRY statements, EPILOG procedures, and EXCEPTION procedures. A process can also contain multiple instances of each of these types of protection (for example, multiple TRY statements).

Multiple TRY Statements

It is possible for multiple TRY statements to be in effect at the same time (for example, because a TRY statement invoked a procedure that also invoked a TRY statement). If multiple TRY statements are in effect, and a termination condition occurs, the system invokes the error-handling code in the most recent TRY statement. If the termination condition can be handled only by an unsafe TRY statement, then the system invokes the most recent protected TRY statement and bypasses any safe TRY statements that are more recent.

For example, Figure 10–1 shows a process stack that has three TRY statements in effect.

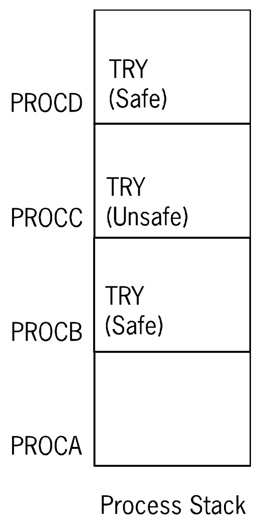

Figure 10–1. Multiple TRY Statements

In this example, if a condition occurs that can be handled by a safe TRY statement, the system then executes the TRY statement in procedure PROCD. If a condition occurs that can be handled only by an unsafe TRY statement, then the system executes the TRY statement in procedure PROCC instead.

TRY Statements and Library Delinkage

A TRY statement can be skipped if a process is executing a procedure imported from a library, and one of the following conditions occurs:

- The library process is discontinued.
- The library is a connection library and is delinked by a DELINKLIBRARY statement with the ABORT option.

The behavior that results depends on whether the library is a server library or a connection library, and whether the TRY statement is safe or unsafe. For example, in Figure 10–2, a process has invoked procedure PROCB in connection library CL.

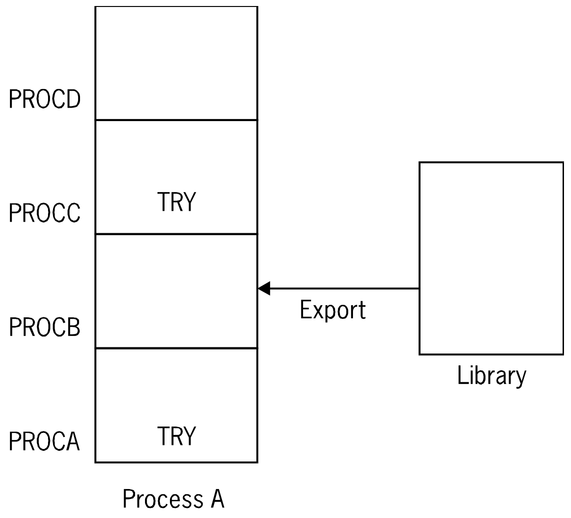

Figure 10–2. TRY Statements and Library Delinkages

The following table explains which TRY error-handling code is executed for the preceding example:

WHEN the TRY statement in PROCC is . . .	**AND the library is a . . .**	**THEN . . .**
Safe	Server library	No TRY statements are executed; the process is discontinued.
Safe	Connection library	The system deletes PROCD, PROCC, and CL.PROCB from the stack, and then executes the TRY code in PROCA.
Unsafe	Server library or connection library	The system executes the TRY error-handling code in PROCC before deleting PROCC from the stack. The TRY code is responsible for cutting back PROCB (for example, by means of a bad GO TO statement).

TRY Statements and ON Statements

If a TRY statement and an ON statement are both in effect, and a fault occurs, the system examines the more recent of the two statements. The system invokes the ON statement if ON is the more recent statement and it is coded to handle the fault in question. Otherwise, the system invokes the TRY statement.

For example, Figure 10–3 shows two process stacks that each includes an ON statement and a TRY statement.

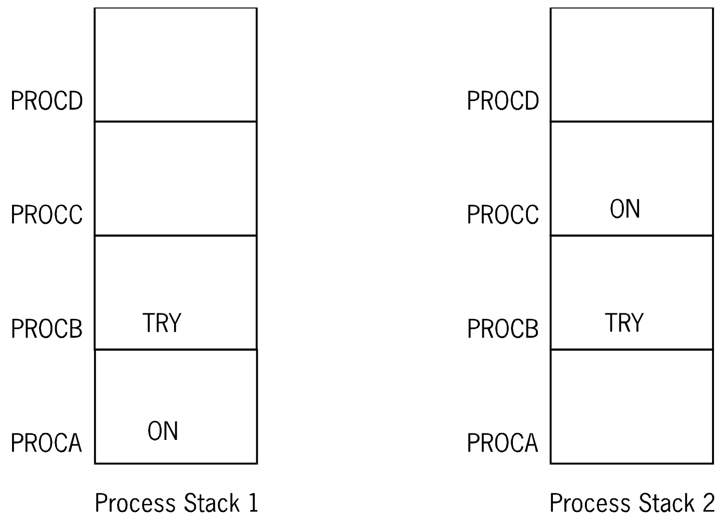

Figure 10–3. ON Statements and TRY Statements

In this example, if a termination condition occurs for process stack 1, the TRY statement is used instead of the ON statement, because the TRY statement is more recent. If a termination condition occurs for process stack 2, then the system checks the ON statement first, because the ON statement is more recent than the TRY statement. However, if the condition cannot be handled by the ON statement, the system executes the TRY statement instead.

TRY Statements, EPILOG Procedures, and EXCEPTION Procedures

If a TRY statement is used in a block that declares an EXCEPTION procedure, the system invokes the EXCEPTION procedure only if the TRY statement does not handle the condition.

If a block contains an EPILOG procedure, then the system invokes the EPILOG procedure at block exit, regardless of whether or not any termination conditions occurred, and regardless of whether or not such conditions were handled by other mechanisms.

While a TRY statement is in effect, a process might enter one or more procedures. If an error occurs, the system deletes the activation records for all procedures entered since the TRY statement. Any of these activation records might include an EXCEPTION procedure or an EPILOG procedure. Before deleting each activation record, the system executes any EXCEPTION or EPILOG procedures contained in that activation record. For example, if the process in Figure 10–4 incurs a termination condition, the system executes the EPILOG procedure in PROCD, then the EXCEPTION procedure in PROCC, and then the TRY statement in PROCB.

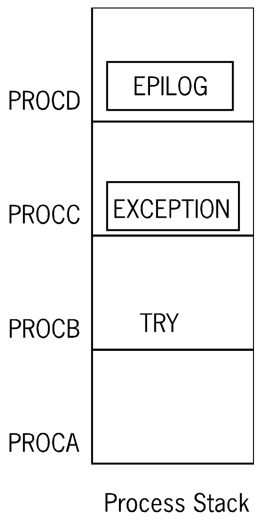

Figure 10–4. TRY Statement, EPILOG Procedure, and EXCEPTION Procedure

The system skips executing TRY error-handling code if an EXCEPTION or EPILOG procedure returns control to a point before the TRY statement. For example, in Figure 10–5, the EXCEPTION procedure in PROCC invokes procedure PROCE. Procedure PROCE, in turn, executes a bad GO TO statement that transfers control to PROCA. Therefore, the TRY statement in PROCB is never executed.

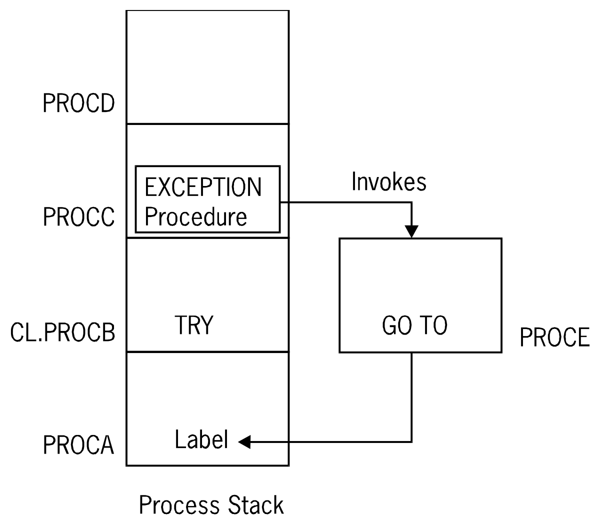

Figure 10–5. TRY Statements and Bad GO TO Statements

Controlling Program Dumps

A program dump is a printout of information about the current state of a process. You can use this information to help debug a defective program. The following subsections explain how to specify when program dumps are to occur, and how to specify which types of information should be included in the dump.

On a system running the Secure Accountability Facility, some security options can restrict the contents of program dumps and the ability to copy program dumps. Refer to the *Security Administration Guide* for details.

Using Program Statements to Control Program Dumps

You can initiate and control program dumps in either of two ways: by using program dump statements, or by using the OPTION task attribute.

The program dump statements that are available are the ALGOL *PROGRAMDUMP* statement, the COBOL74 and COBOL85 *CALL SYSTEM DUMP* statement, the FORTRAN77 *DEBUG PROGRAMDUMP* statement, and the Pascal *Programdump* procedure. Some languages provide other statements to dump process information, but these are language-specific features. The preceding statements call an operating system feature that is available from a variety of sources.

Alternatively, you can enable a program dump by setting certain options of the OPTION task attribute. If the FAULT option is set, then the process generates a program dump if it terminates abnormally because of an internal cause. If the DSED option is set, then the process generates a program dump if the process terminates abnormally because of an external cause. For a definition of internal and external causes, refer to "Understanding Internal and External Causes" later in this section.

You can also specify various dump options, which determine the types of information that are included in the program dump. These dump options can be accessed through assignments to the OPTION task attribute. In ALGOL, FORTRAN77, and Pascal, these options can also be set by parameters in a program dump statement.

If a program dump statement specifies dump options, then the dump options specified in that statement are used, and the value of the OPTION task attribute is ignored. If the program dump is caused by the DSED or FAULT option of the OPTION task attribute, or by a program dump statement that does not specify any dump options, then the dump options specified by the OPTION task attribute are used for the dump.

The possible dump options are ARRAY, BASE, CODE, DBS, FILE, LIBRARIES, PRESENTARRAYS, PRIVATELIBRARIES, TODISK, CRITICALBLOCK, and TOPRINTER. The effects of these options are explained in the discussion of the OPTION task attribute in the *Task Attributes Programming Reference Manual*. The effects of the TODISK and TOPRINTER options are also discussed under "Controlling the Program Dump Destination" later in this section.

Using Operator Commands to Control Program Dumps

If a process is behaving abnormally, you might want to invoke a program dump for the process. You can use the dump later to help debug the process.

One way you can invoke a dump is by using the DUMP (Dump Memory) system command. The *<mix number> DUMP* form of this command initiates a program dump for the specified process. The *<mix number> DUMP <option list>* form of this command assigns dump-related options to the OPTION task attribute and then initiates a program dump. The OPTION values are retained, and they affect any later program dumps for the process, unless overridden by later assignments. The following example dumps information about arrays and files for a process with the mix number 3457:

```
3457 DUMP ARRAYS, FILES
```

It is possible to view the program dump while the process is still running. Refer to "Analyzing a Program Dump from a Running Process" later in this section.

You can also trigger a dump by way of the DS (Discontinue) system command. The *<mix number> DS <option list>* form of this command initiates a program dump and discontinues the process. The option list in this command controls the contents of the dump by assigning options to the OPTION task attribute. The following example dumps arrays and code segments and discontinues the process with mix number 3457:

```
3457 DS ARRAYS, CODE
```

Note that the simple form of the DS command, *<mix number> DS*, causes a program dump if the DSED option of the OPTION task attribute was previously set through object code file assignments, task equations, or task attribute assignments executed by the process. You can prevent such a dump from occurring by using the *<mix number> DS NONE* form of the DS command.

A program dump requested by the DUMP system command might not be taken immediately if the process is running at a low priority or is in an MCP service routine. For example, if the process is waiting on an event, the program dump is not taken until the wait completes.

Furthermore, if an operator uses a DS command to discontinue the process while a program dump is pending for the process, then the program dump might not be taken at all. The DS command causes the pending program dump to be taken only if at least one of the following conditions is true:

- The DSED option of the OPTION task attribute is set.
- The previous DUMP command specified the DSED option.
- The DS command specifies one or more dump options.

Controlling the Program Dump Destination

You can direct a program dump to a printer backup file for printing, to a disk file for later analysis and printing, or both. You can control the program dump destination through two dump options: TOPRINTER and TODISK. These options are available in ALGOL, FORTRAN77, and Pascal through program dump statements. Languages that provide access to task attributes can also assign these options by way of the OPTION task attribute. Additionally, these options can be assigned in a DS (Discontinue) or DUMP (Dump Memory) system command.

The TOPRINTER option directs program dumps to a printer backup file called the *task file.* For details about the task file, refer to “Using the Task File” later in this section.

The TODISK option directs program dumps to a disk file, and causes a brief summary of each dump to be written to the task file. The contents of the program dump are determined by the other dump options, except for the BASE option. Whenever TODISK is set, the BASE option is treated as if it is also set.

The following are advantages to using the TODISK option instead of the TOPRINTER option:

- The dump is performed more rapidly, and less printer output is produced at the time of the dump. This factor makes it convenient for you to set the dump options to dump all possible information. By setting all the dump options, you reduce the likelihood of having to try to reproduce the problem later to obtain more information.
- The disk file stores dump information in a format that can be analyzed by the DUMPANALYZER utility. DUMPANALYZER also enables you to decide at analysis time what information to include in the report. You can even run DUMPANALYZER repeatedly to produce reports on different information from the same dump.

Another benefit is that DUMPANALYZER provides a detailed analysis of the process information block (PIB).

You can also use DUMPANALYZER to produce a report similar to one created by the TOPRINTER option. Like TOPRINTER reports, DUMPANALYZER reports include the names of all the identifiers used by the process. (However, identifiers are included in the report only if all object code files used by the process are present when DUMPANALYZER is run.) For information about running the DUMPANALYZER utility, refer to the *System Software Utilities Operations Reference Manual*.

When the TODISK option is used, the default title for the resulting disk file has the following format:

```
(<usercode>)PDUMP/<process name>/<date>/<time>/<mix number> ON <family>
```

The values of the various elements of this title are as follows:

Title Element	**Value**
<usercode>	The value of the USERCODE task attribute of the process.
<process name>	The value of the NAME task attribute of the process, except that any usercode or family name is omitted. If the resulting process name is more than eight nodes long, then only the first eight nodes are included.
<date>	The current date, in the form YYMMDD.
<time>	The current time, in the form HHMMSS.
<mix number>	The value of the MIXNUMBER task attribute of the process.
<family>	DISK, unless the FAMILY task attribute provides a primary family to be used in place of DISK.

The following is an example title:

```
(UC)PDUMP/OBJECT/TEST/PDUMP/961009/154444/0958 ON APACK
```

If you are running the process from CANDE, it will display the name of the final program, dump to disk. Only the final dump name is displayed even though the program may perform more than one.

```
Program Dump: (UC)PDUMP/OBJECT/TEST/PDUMP/961009/154444/0958 ON APACK
```

You can use file equations to specify a different file name or family name for a dump to disk. You can file-equate the FILENAME, FAMILYNAME, and TITLE file attributes. The file equations must specify PDUMP as the internal name of the file. For example, a WFL job can use the following statement to initiate a program and specify the title of any program dumps generated by the program. Note that the file equation has effect only if the TODISK option is specified, either in the OPTION task attribute or in the statement that invokes the program dump.

```
RUN OBJECT/JADCON;
    FILE PDUMP(TITLE = JADCON/DUMP ON PACK);
    OPTION = (FAULT, TODISK);
```

If a program dump occurs, the system adds a suffix to the file-equated title. The suffix is a 3-digit integer ranging from 000 to 999. The suffix is incremented by one for each program dump generated by the process. Thus, in the previous example, if OBJECT/JADCON runs under usercode BLAKE and generates three program dumps in a single run, the program dumps receive the following titles:

```
(BLAKE)JADCON/DUMP/000 ON PACK;
(BLAKE)JADCON/DUMP/001 ON PACK;
(BLAKE)JADCON/DUMP/002 ON PACK;
```

You can include a usercode in the PDUMP file equation, but only a privileged process can assign the program dump a usercode different from that of the process. If the process is nonprivileged, and PDUMP is equated to a different usercode, then a security violation results when a program dump occurs. The system deletes the program dump file rather than saving it under the requested usercode.

If neither the TODISK nor the TOPRINTER option is set, the operating system option PDTODISK determines whether the program dump is directed to a disk file or to the task file. If the PDTODISK option is set, program dumps are written by default to a disk file; otherwise, program dumps are directed by default to the task file. An operator can use the OP (Options) system command to set or reset the PDTODISK option.

If either the TODISK or TOPRINTER option is set for a process, the program dump is directed only to the destination specified by the option: a disk file for TODISK, or the task file for TOPRINTER. If both of these options are set, then two program dumps occur: the first is directed to disk and the second is directed to the task file.

If the TODISK and TOPRINTER options are both used, the two resulting dumps may differ slightly, because the act of directing a program dump to disk can cause some arrays used by the process to be made present or overlaid. The contents of the arrays are not affected, and both present and overlaid arrays are included in the dump. However, if you compare both of the dumps that were produced, you might see the same array indicated as present in one dump, and overlaid in the other dump.

Using the Task File

The task file is a predeclared printer backup file that is associated with each process. If a program dump is directed to a task file, the task file is automatically queued for printing, in the same way as other printer backup files produced by a process. If a process generates multiple program dumps, then by default, they are all stored in the same task file.

You can use the TASKFILE task attribute to write comments to the task file or interrogate the file attributes of the task file. You can also use this task attribute in a program to force multiple program dumps to be stored in separate backup files. The program can achieve this effect by closing the task file after each dump and then writing a comment to the task file. An example of this method is given in the TASKFILE task attribute description in the *Task Attributes Programming Reference Manual*.

A program can also use the TASKFILE task attribute to access the task file of an ancestor process.

You can assign file attributes to the task file through file equation. This task attribute can be assigned only before process initiation. The following is a WFL example of such an assignment:

```
RUN OBJECT/PROG;
  FILE TASKFILE (PRINTDISPOSITION=DONTPRINT,USERBACKUPNAME=TRUE,
                 FILENAME=PROG/DUMP);
```

You can also use the BDNAME task attribute to save the task file and assign a prefix other than *BD to the file title.

Some security restrictions apply if file equations or a BDNAME task attribute assignment is used to prefix the task file title with a usercode other than that of the process. The following are WFL examples of such statements:

```
RUN OBJECT/PROG;
  BDNAME = (FRAN)PROGDUMP;

RUN OBJECT/PROG;
  FILE TASKFILE (PRINTDISPOSITION=DONTPRINT,USERBACKUPNAME=TRUE,
                 FILENAME=(FRAN)PROGDUMP);
```

In general, a process must have privileged status to open a file under another usercode. The system enforces this rule even more strictly for task files by requiring that the process have a privileged usercode rather than merely being a privileged program. The purpose of this restriction is to prevent nonprivileged users of privileged programs from using a program dump to overwrite files under another usercode.

This restriction is not foolproof, however. If a privileged program is running under a nonprivileged usercode, and the program opens the task file with a write statement before the dump takes place, the program can successfully open the task file under another usercode. The following is an ALGOL example of such a write statement:

```
WRITE (MYSELF.TASKFILE,//,"DUMP NUMBER ONE");
```

When the program dump takes place later, the dump is directed to the already-opened task file. For this reason, if you are designing a privileged program intended for use by nonprivileged users, you should not include any statements that would cause the task file to be opened before the dump.

Analyzing a Program Dump from a Running Process

Some program dumps occur when a program is terminated, either by a fault or by a DS (Discontinue) system command. However, there can also be situations when it is useful to generate a program dump for a process while it is still running. Such a dump can be initiated by the DUMP (Dump Memory) system command or by a PROGRAMDUMP statement in the program.

By default, program dumps are directed to printer and do not print until the process and its job have terminated. The following paragraphs explain how you can gain access to the program dump while the process is still running.

One method of gaining immediate access to a program dump is by directing the program dump to disk. For information on directing dumps to disk, refer to “Controlling the Program Dump Destination” earlier in this section. If the program dump is directed to disk, then the dump file becomes available as soon as the dump is completed. You can then run the DUMPANALYZER utility to analyze the disk file. For a description of DUMPANALYZER, refer to the *System Software Utilities Operations Reference Manual*.

If the program dump is directed to printer, you can enable immediate printing by setting the PRINTDISPOSITION attribute of the task file to CLOSE. You can accomplish this assignment with a task equation in the statement that runs the program. The following is a WFL example:

```
RUN OBJECT/TEST/ALGOL/TASK;FILE TASKFILE(PRINTDISPOSITION= CLOSE)
```

Alternatively, you can assign the task file PRINTDISPOSITION through a FILECARDS task attribute assignment within the program. The following is an ALGOL example:

```
REPLACE MYSELF.FILECARDS BY
   "FILE TASKFILE(PRINTDISPOSITION = CLOSE);" 48"00";
```

If the program dump is initiated by the DUMP command, the system closes the task file at the end of the program dump. The PRINTDISPOSITION attribute then causes the program dump to be queued for printing.

If the program dump is initiated by a PROGRAMDUMP statement in the program, the task file is *not* closed automatically at the end of the dump. To cause immediate printing, the program must follow the PROGRAMDUMP statement with a statement that closes the task file. The following is an ALGOL example:

```
CLOSE(MYSELF.TASKFILE);
```

Determining Whether a Dump is In Progress

If a process is taking a program dump, then this fact is reflected in the Stack State display of the Y (Status Interrogate) system command. The following example shows the output from the Y command for a process that is taking a dump:

```
Status of Task 6408/6513 at 16:39:09
Program name: *OBJECT/ED ON SYS00
Priority: 50
Origination: OCDWH_1/CANDE/2  (LSN 295)
MCS: SYSTEM/CANDE
Usercode: DEBS
Chargecode: 6893
Stack State: Waiting on an event, Programdumping
Display: ED:OPERATOR DSED @ (47312200)
```

This command can be useful, for example, if you have entered a DS (Discontinue) system command for a process, and the process seems to be taking a long time to terminate. By entering the Y command, you can determine whether the delay is caused by a program dump. You can then enter a second DS command if you wish to discontinue the program dump.

Causing Symbolic Dumps for RPG Processes

The task file of an RPG process can store a *symbolic dump* instead of, or in addition to, a program dump. A symbolic dump provides much of the same information as a program dump, but is shorter and simpler to read. A symbolic dump can be produced in any of the following ways:

- The RPG process can execute a DUMP operation code. This operation produces a symbolic dump, but no program dump. By default, the symbolic dump is written to the task file. However, the RPG process can specify that the symbolic dump is to be written to another file previously declared by the process.
- The operator can enter the *AX DUMP* form of the AX (Accept) system command in response to a halted RPG program. This action produces a symbolic dump, but no program dump. The symbolic dump is always written to the task file.
- The RPG process can generate a program dump when the process terminates abnormally and dump options were specified in a DS (Discontinue) system command or the DSED or FAULT option was set in the OPTION task attribute. If an abnormal termination results in a program dump, a symbolic dump appears in the task file after the program dump. If there is no program dump, then no symbolic dump is produced either.

The DUMP (Dump Memory) system command, when applied to an RPG process, produces a program dump, but no symbolic dump.

For further information, refer to the discussion of the DUMP operation code in the *Report Program Generator (RPG) Programming Reference Manual, Volume 1: Basic Implementation*.

Effect of Resource Limits on Program Dumps

Resource limits imposed by task attributes are deliberately overridden by the system for a process that is generating a program dump. Task attributes that might be overridden include ELAPSEDLIMIT, MAXIOTIME, MAXLINES, MAXPROCTIME, MAXWAIT, SAVEMEMORYLIMIT, STACKLIMIT, TEMPFILELIMIT, and WAITLIMIT. This policy ensures that a process can generate a complete program dump, even when the termination is caused by the process exceeding one of these limits

Understanding Internal and External Causes

The causes of abnormal terminations are divided into two categories: internal and external. The difference between these two types of causes can determine whether a process generates a program dump, and whether the process restarts automatically. To be more specific, the FAULT option of the OPTION task attribute causes a program dump if a process is discontinued by an internal cause. The DSED option of the OPTION task attribute causes a program dump if a process is discontinued by an external cause. The RESTART task attribute causes a process to restart only if it is terminated by an internal cause. If the task has not executed any user code, the RESTART count is ignored. For example, RESTART is ignored if a task fails to run because the codefile is too old to run on the current release.

An abnormal termination is considered to be due to an external cause if the HISTORYCAUSE and HISTORYREASON task attributes have any of the following combinations of values:

HISTORYCAUSE	**HISTORYREASON**
OPERATORCAUSE	Any
RESOURCECAUSE	Any
FAULTCAUSE	DISKPARITYV
SYSTEMCAUSE	APPLICATIONTIMEOUTV or FORCIBLECLOSEV
PROGRAMCAUSE	DATABASEDIEDV, LIBRARYDIEDV, DEATHINFAMILYV, STEPPARENTDIEDV, INFANTICIDEV, or CLIENTDIEDINACRV

An abnormal termination is considered to be due to an internal cause if the HISTORYCAUSE and HISTORYREASON values are not any of the combinations listed in the preceding table.

Section 11
Restarting Jobs and Tasks

A process can be discontinued by any of a variety of causes, including system commands, program faults, or resource limits. In most situations, this discontinuation is permanent. The system does not attempt to continue execution of the process or restart it from the beginning. The system removes the process stack, process information block (PIB), task attribute block (TAB), and any temporary files that the process was using. Only the permanent files used by the process are preserved and reflect all changes made by the process before it terminated.

However, in certain cases you can cause various aspects of a process to be saved for later restarting. This section discusses the following methods of saving and restarting processes:

- Work Flow Language (WFL) job restarts. The system automatically stores information about WFL jobs that enables the jobs to restart after a halt/load. Further, you can terminate and restart a WFL job manually with the RESTART (Restart Jobs) system command.
- Checkpoint facility. This facility stores a complete copy of a process and its data items, so that the process can be re-created later from a given point in its execution.
- CHECKPOINTARRAY procedure. ALGOL and NEWP processes can use this procedure to copy an array to disk or recover the array from disk. CHECKPOINTARRAY has fewer restrictions than the checkpoint facility and thus is suitable for wider use.

An additional restart method is the RESTART task attribute, which restarts a process that was terminated due to a fault. For further information about the RESTART attribute, refer to "Retrying a Failed Task" in Section 10, "Determining Process History."

Designing WFL Jobs for Automatic Restarts

A WFL job is the only type of user process that automatically restarts if interrupted by a halt/load. If a halt/load occurs while a WFL job is executing, then the WFL job and its offspring are terminated. After the halt/load, the job recovers in one of two ways.

If the restarted WFL job was executing a checkpointed task at the time of the halt/load, then a process called JOBRESTART appears in the W (Waiting Entries) system command display. For information about how to respond to this waiting entry, refer to "Restarting a Checkpointed Task" later in this section.

If the job was not executing a checkpointed task at the time of the halt/load, the system begins execution of the job from the last point that a successful *job rollout* took place. A job rollout stores selected information about the job for use if the job is restarted. The system attempts a job rollout before each of the following statements:

ALTER	COPY	PRINT	RUN
ARCHIVE	LOG	PROCESS	START
CHANGE	MODIFY	PTD	WAIT
COMPILE	OPEN	REMOVE	

Also, if any of the following statements contains an ACCEPT function, the system attempts a job rollout before the statement:

Assignment Statements	DO	WHILE
CASE	IF	

Additionally, the system attempts job rollouts after each WAIT statement. The WAIT statement is the only statement for which the system attempts a rollout both before and after the statement.

Note that a job rollout can succeed only if no tasks are running. If a job rollout fails, and the system has to restart the job, then the system restarts the job from a previous job rollout. The following examples illustrate this point:

- Suppose that at the time of the halt/load the WFL job is waiting for a single synchronous task to complete. Therefore, the last successful job rollout took place before the initiation of that task. After the halt/load, the WFL job resumes by executing the task initiation statement again. This creates a new task that is an instance of the same program.
- Suppose that the WFL job initiates three asynchronous tasks before the halt/load, and all of these tasks are still in use when the halt/load occurs. Therefore, the last successful job rollout took place before the first asynchronous task was initiated. After the halt/load, execution resumes with the first of the three task initiation statements.

- Suppose the WFL job initiated an asynchronous task called *A* and then another asynchronous task called *B*. Task *A* terminates while task B is still in use. Then a halt/load occurs, also while task *B* is still in use. Therefore, the last successful job rollout took place before A was initiated. After the halt/load, execution of the job resumes with the statement that initiated task *A.* This was the last point at which no in-use task existed, because task *A* still existed when task *B* was initiated.

Preventing Job Side Effects

The values of Boolean, integer, real, and string identifiers are retained across a halt/load. The values of job parameters are also retained. However, the following types of values are lost:

- The values of the task attributes of the job, except for the MIXNUMBER task attribute and any task attributes assigned in the job attribute list. Thus, for example, values assigned to any task attribute using the MYJOB task variable are not retained.
- The task attribute values of task variables declared in the job. Only task attribute values assigned in task variable declarations are retained.
- The effects of the ST (Stop) system command. The effects of this command are not retained across a halt/load, because the ST system command simply assigns the STATUS task attribute a value of SUSPENDED.
- The effects of any file attribute assignments that were executed after the file declaration in the job.

The ON RESTART statement can be used to specify actions that are taken immediately after a halt/load. Typically, the ON RESTART statement is used to restore the values of file and task variables before job execution continues.

Task equations included in task initiation statements are reexecuted when the task initiation statement is reexecuted. Therefore, the ON RESTART statement does not need to restore attributes specified in task equations.

The job can determine whether it has been restarted by interrogating the RESTARTED task attribute. This task attribute returns a value of TRUE if the job has been restarted.

Preventing Task Side Effects

When the WFL job reinitiates a task, some of the physical files used by the new task might reflect changes made by the old task before the halt/load. When designing a program that is to be initiated by a WFL job, you must plan ahead for this possibility and provide a way for the program to produce appropriate audit trails.

A WFL task that opens a remote file might not be able to do so after a halt/load. Normally, a task equation such as the following is used to enable a WFL task to open a remote file:

```
RUN OBJECT/PROG;
   STATIONNAME = #MYSELF(SOURCENAME);
```

This task equation directs the task to open any remote files at the station that initiated the WFL job. However, the requested station might not exist after a halt/load. This is the case, for example, if the job was initiated from a pseudostation, such as a Command and Edit (CANDE) dialogue opened through the Transaction Server. This pseudostation is discarded during a halt/load and is not reestablished until you log on to the same CANDE dialogue again. The task terminates abnormally if it attempts to open a remote file at a nonexistent pseudostation.

Understanding Job Restart Failure

Any of the following circumstances can prevent a WFL job from restarting after a halt/load:

- The system switches to using a different job description file after the halt/load. The operator can use system commands to cause the switch to a different job description file. For further information, refer to the discussion of the job description file in the *System Administration Guide*.
- The operator physically transfers the pack containing the job description file to an incompatible type of system and attempts to make it the new job description file for that system.

  If the operator uses the *DL JOBS ON <family>* command to mark the pack as the location of the next job description file, then after the next halt/load, the system attempts to restart the jobs from the specified job description file. The jobs should restart successfully, provided that the pack was transferred to a compatible type of system with the same type of memory architecture. The following are the compatible classes of systems; transfers within each class are possible, but transfers between the classes are not:

  – A 7, A2100, NX4200

  – A11, A14, A16, A18, A19, A2410, NX4600

  – A2800, NX4800, NX5600, NX5620, NX5800, NX5820, NX6820, NX6830, LX Systems, CS7101, CS7201

  If you transfer a job description file between systems of different classes, then the restart of each job fails with the error INCOMPATIBLE SYSTEM TYPE. Additionally transferring the job description file between incompatible systems can cause the system to halt/load again.
- The operating system option AUTORECOVERY is reset. The operator can reset this option using the OP (Options) system command. Resetting AUTORECOVERY causes the mix limit for each job queue to be set to zero after a halt/load. Any job that would have restarted will instead remain in a job queue until the operator uses the MQ (Make or Modify Queue) system command to assign a new mix limit to the job queue.

  Resetting AUTORECOVERY also prevents automatic halt/loads in some situations. For details, refer to the *System Commands Operations Reference Manual*.

- An operator changes the job queue definitions after the job is initiated, but before the halt/load. For example, the job attribute list of a job might set CLASS = 10 and MAXPROCTIME = 60. The definition of job queue 10 might include a PROCESSTIME limit of 120. The job is submitted through job queue 10 originally. While the job is executing, an operator might use the MQ (Make or Modify Queue) system command to lower the PROCESSTIME limit for that job queue to 30. Then a halt/load might occur. After the halt/load, the job cannot restart because its MAXPROCTIME value is greater than the PROCESSTIME limit that is now defined for job queue 10. The job terminates abnormally with a queue violation.
- A task of the job executed a checkpoint and then was terminated by the halt/load. In this case, the job is suspended after the halt/load and appears in the W (Waiting Entries) system command display. For information about operator responses to this situation, refer to "Restarting a Checkpointed Task" later in this section.

Understanding Disk Resource Control Effects

On systems using the Disk Resource Control (DRC) system, the system normally delays restarting WFL jobs until the DRC system becomes active. The WFL jobs remain in their job queues, and the system displays an RSVP message notifying the operator that DRC initialization is underway. You can use the FS (Force Schedule) system command to force a queued WFL job to restart before DRC is active. However, be aware that the following statements in a WFL job can have unexpected effects if they execute before DRC is active:

- CHANGE and REMOVE statements

  If these statements specify usercoded files, the statements are skipped without being executed.
- COPY statement

  If this statement creates a usercoded copy of a file, the copy proceeds normally, but DRC is not notified of the increased disk usage for that usercode. Therefore, it might become possible for the actual disk usage of that usercode to exceed the limit set in DRC.

For further information about DRC, refer to the *System Administration Guide* and the *System Operations Guide.*

Manually Restarting WFL Jobs

You restart a running WFL job with the RESTART (Restart Jobs) system command. This command first discontinues the current job and its tasks, and then restarts the new job as though a halt/load had occurred. For example, the job resumes execution from the last point at which no offspring were in use. For further information about the restart point, as well as about job and task side effects that you should plan for, refer to "Designing WFL Jobs for Automatic Restarts" earlier in this section.

If the DSED program dump option is set for any task of the job, the RESTART command causes the task to generate a program dump.

You can use the RESTART command to achieve some of the effects of a halt/load without interrupting the system. For example, if you need to perform maintenance on a disk unit, you must terminate any jobs that have files open on that disk unit. You can set the mix limit for the relevant job queue to zero, and then apply the RESTART command to such a job (rather than using the DS command). You can then hold the restarted job in the job queue until pack maintenance is completed. When you increase the mix limit, the job restarts from the last point where it had no tasks active.

You can also use the RESTART command to test ON RESTART statements in WFL jobs without having to halt/load the machine.

If the WFL job had no checkpointed task in progress at the time of the RESTART command, then the system automatically submits the job to a job queue. The job resumes execution whenever it is selected from that job queue.

If the WFL job was executing a checkpointed task at the time the RESTART command was entered, the job does not restart immediately after the command. Instead, the independent runner JOBRESTART appears in the W (Waiting Mix Entries) system command display. For information on how to respond to this waiting entry, refer to "Restarting Checkpointed Tasks Automatically" later in this section.

Checkpoint Facility

The checkpoint facility provides the ability to restart a terminated task from any selected point in its execution. Invoking a checkpoint causes the creation of a checkpoint file, which records the state of the task when the checkpoint was invoked. Either a statement in the task or a BR (Breakout) system command can invoke a checkpoint. Later, you can use the RERUN statement to restart the task from the point at which the checkpoint was invoked.

The main application of the checkpoint facility is the restarting of tasks that were terminated by a system halt/load. The unique advantage of the checkpoint facility is the ability to restart a task from a selected point during the task's execution. You can invoke repeated checkpoints for the same task and restart the task from any of these checkpoints.

ALGOL provides a CHECKPOINT statement that enables a program to invoke a checkpoint during its execution. Additionally, programs can invoke a checkpoint by calling the exported MCP procedure CALLCHECKPOINT. The CALLCHECKPOINT procedure can be invoked from any of the languages that support libraries, including C, COBOL74, COBOL85, FORTRAN77, NEWP, and Pascal. You can also use operator commands to initiate a checkpoint for ALGOL and COBOL74 tasks. However, checkpoints cannot be initiated for WFL or RPG tasks.

There are several restrictions on the circumstances in which a checkpoint can be invoked. One restriction is that the task must have been initiated from WFL, rather than from a session or a user program. Another is that the task must not have any offspring. These restrictions, and others, are discussed in detail in the following subsections.

Programmatically Invoked Checkpoints

Designing a program to be checkpointed and successfully restarted involves more than simply including a checkpoint invocation statement. You must verify that the program is not using features that are disallowed for checkpointing. You must also plan for recovery of data file contents and libraries.

Storing Information with a Checkpoint

Invoking a checkpoint causes the following types of information about the task to be stored:

- The structure of the process stack, including information about the procedures that have been entered, but have not yet exited, and the statement that is currently being executed
- The current values of all objects declared by the task
- The current values of the task attributes of the task

Planning for File Recovery

The checkpoint facility does not store a record of the contents of files used by a task. Instead, information is stored about the attributes of the files; that is, whether each file is open and the current position of the record pointer for each file. You must plan for the fact that file contents might have been modified, or files might have been removed or replaced, between the time the task was checkpointed and the time it is restarted.

When the task is restarted, each data file must be on the same type of medium as it was when the checkpoint was invoked. They do not have to be on the same physical units or at the same locations on disk. They must retain the same basic characteristics, such as blocking.

If a temporary disk file is open when the checkpoint is invoked, the file is locked and assigned a title that begins with the letters CP. However, the system does not assign this title to the TITLE attribute of the logical file; instead, the TITLE attribute retains whatever value it was assigned by the program. If this file is later locked by the program, the system enters the file in the disk directory under the title specified in the TITLE file attribute. At restart, the process looks for the file only under the CP directory, and the task is suspended with a NO FILE condition.

To prevent this situation, all files that will eventually be locked can be opened as permanent files. That is, the file attribute PROTECTION can be set to SAVE. You can design the task to remove this file later by closing the file with the PURGE option set. Another method of avoiding this problem is never to lock a temporary file.

Planning for Library Recovery

It is possible to checkpoint a user task that is linked to a library, but only if the task is not currently executing a library procedure. When a user task linked to a library is checkpointed, the checkpoint records the values of the library attributes. However, the checkpoint does not store any information about the state of the library or its contents.

When the user task restarts, the task is not immediately relinked to the library. The library link is reestablished the first time the user task calls a library procedure.

You must be aware that the values of global objects in the library might have changed since the user task was checkpointed. Global objects in the library might have changed for any of the following reasons:

- The user task might have invoked a library procedure after the checkpoint and before the user task terminated. This library procedure might have included statements that modified the values of global objects in the library.
- If the library was frozen with duration of TEMPORARY and a sharing option of PRIVATE, then the library thaws when the user task terminates, and the system removes the library process. The values of all global objects in the library are lost when the library terminates. When the user task restarts, its first attempt to use the library causes the creation of a new instance of the library.
- If the library has a sharing option of SHAREDBYALL, then other tasks have access to the library and might make changes to global objects in the library after the original user task is checkpointed.

Invoking the Checkpoint

The task invokes a checkpoint by executing a CHECKPOINT statement or by invoking the exported MCP procedure CALLCHECKPOINT. These methods are discussed separately in the following pages.

Using a CHECKPOINT Statement

ALGOL provides a CHECKPOINT statement. You can create multiple checkpoints by including a CHECKPOINT statement at several points in the program. Later, you can restart the task from any of these checkpoints.

Each CHECKPOINT statement can specify the following options:

- Device option

  Determines the family where the checkpoint-related files are to be created. A value of DISK causes checkpoint files to be created on the family named DISK. A value of DISKPACK causes checkpoint files to be created on the family named PACK. You can specify PACK as a synonym for DISKPACK.

- Disposition option

  Determines whether checkpoint files are saved. If the value is PURGE, then the checkpoint files are removed if the task terminates normally. If the value is LOCK, then checkpoint files are saved indefinitely. Later, you can use the checkpoint files to restart the task even if it terminated normally.

  The disposition option also determines if a checkpoint removes any previous checkpoint files created by the same task. If the disposition is PURGE, then any previous checkpoints that were invoked with a disposition of PURGE are removed. If the disposition is LOCK, then no previous checkpoints are removed.

The following is an example:

```
CHECKPOINT (DISK,PURGE);
```

Using the CALLCHECKPOINT Procedure

A program can invoke a checkpoint by calling the MCP exported procedure CALLCHECKPOINT. You can create multiple checkpoints by invoking CALLCHECKPOINT at several points in the program. Later, you can restart the task from any of these checkpoints.

CALLCHECKPOINT is an integer procedure that receives four integer parameters, in the following order: UTYP, CPTYP, CCODE, CPNUM, and RSFLAG. The following table explains these parameters.

Parameter Name	Type	Input/ Output	Meaning
UTYP	Integer	Input	Similar to the device option in a CHECKPOINT statement. This parameter determines the family where the checkpoint-related files are created. Checkpoint files with a value of 1 are created on the family named DISK; files with a value of 17 are created on the family named PACK. These values can also be represented by the VALUE function in ALGOL as VALUE(DISK) and VALUE(PACK).
CPTYP	Integer	Input	Similar to the disposition option in a CHECKPOINT statement. This parameter determines whether checkpoint files are saved. A value of 0 is the same as a disposition of PURGE: checkpoint files are removed if the task terminates normally. A value of 1 is the same as a disposition of LOCK: checkpoint files are always saved indefinitely. Later, you can use the checkpoint files to restart the task even if it terminated normally. The CPTYP parameter also determines if a checkpoint removes previous checkpoint files created by the same task. If the disposition is 0 (PURGE), any previous checkpoints invoked are removed. If the disposition is 1 (LOCK), no previous checkpoints are removed.
CCODE	Integer	Output	If the checkpoint is unsuccessful, the CCODE parameter stores one of the values listed in Table 11–1, “Checkpoint Completion Codes.”

Parameter Name	Type	Input/ Output	Meaning
CPNUM	Integer	Output	CPNUM returns the number the system assigned to this checkpoint. The numbering scheme is explained in "Creating Output Disk Files with a Checkpoint" in this section.
RSFLAG	Integer	Output	If the task is restarted from a checkpoint, RSFLAG returns a value of 1 the next time the task invokes CALLCHECKPOINT. In this case, CALLCHECKPOINT actually does not invoke a checkpoint for the task. If the task invokes CALLCHECKPOINT a second time, RSFLAG returns a value of 0 and the checkpoint is actually invoked.
Procedure result	Integer	Output	A value of 0 indicates a successful checkpoint. A value of 1 indicates the checkpoint was not taken, in which case either the CCODE parameter or the RSFLAG parameter should be nonzero.

The following are ALGOL statements that declare the CALLCHECKPOINT procedure and invoke it:

```
LIBRARY MCPSUPPORT(LIBACCESS=BYFUNCTION,FUNCTIONNAME="MCPSUPPORT.");

INTEGER PROCEDURE CALLCHECKPOINT(UTYP, CPTYP, CCODE, CPNUM, RSFLAG);
  INTEGER UTYP, CPTYP, CCODE, CPNUM, RSFLAG;
  LIBRARY MCPSUPPORT;

INTEGER CCODE_ACTUAL, CPNUM_ACTUAL, RSFLAG_ACTUAL, CPRESULT;

CPRESULT:= CALLCHECKPOINT(VALUE(DISK),1, CCODE_ACTUAL, CPNUM_ACTUAL,
  RSFLAG_ACTUAL);
```

The following COBOL85 program uses the explicit library interface to invoke the CALLCHECKPOINT procedure. The invocation specifies a device option of PACK and a disposition of PURGE.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CHECK-POINT.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT ATTR-FILE ASSIGN TO DISK.
DATA DIVISION.
FILE SECTION.
FD ATTR-FILE.
01 ATTR-REC PIC X(80).

WORKING-STORAGE SECTION.
77 CHECKPOINTDEVICE PIC S9(11) USAGE BINARY.
77 CHECKPOINTTYPE   PIC S9(11) USAGE BINARY.
77 COMPLETIONCODE   PIC S9(11) USAGE BINARY.
```

```
77 CHECKPOINTNUMBER PIC S9(11) USAGE BINARY.
77 RESTARTFLAG      PIC S9(11) USAGE BINARY.
77 RSLT             PIC S9(11) USAGE BINARY.
77 VALUE-OF-PACK    PIC S9(11) USAGE BINARY.
77 VALUE-OF-PURGE   PIC S9(11) USAGE BINARY VALUE 0.

LOCAL-STORAGE SECTION.
LD LD-CALLCHECKPOINT.
77 CHECKPOINTDEVICE PIC S9(11) USAGE BINARY.
77 CHECKPOINTTYPE   PIC S9(11) USAGE BINARY.
77 COMPLETIONCODE   PIC S9(11) USAGE BINARY.
77 CHECKPOINTNUMBER PIC S9(11) USAGE BINARY.
77 RESTARTFLAG      PIC S9(11) USAGE BINARY.
77 RSLT             PIC S9(11) USAGE BINARY.

PROGRAM-LIBRARY SECTION.
LB MCPSUPPORT IMPORT
   ATTRIBUTE
   FUNCTIONNAME IS "MCPSUPPORT"
   LIBACCESS    IS BYFUNCTION.
ENTRY PROCEDURE CALLCHECKPOINT
   WITH   LD-CALLCHECKPOINT
   USING
          CHECKPOINTDEVICE
          CHECKPOINTTYPE
          COMPLETIONCODE
          CHECKPOINTNUMBER
          RESTARTFLAG
   GIVING
          RSLT.

PROCEDURE DIVISION.
INIT-PARA.
        CHANGE ATTRIBUTE KIND         OF ATTR-FILE
                                      TO PACK.
        MOVE   ATTRIBUTE KIND         OF ATTR-FILE
                                      TO VALUE-OF-PACK.
        PERFORM CHECKPOINT-PARA.
        STOP RUN.

CHECKPOINT-PARA.
        MOVE VALUE-OF-PACK  TO CHECKPOINTDEVICE.
        MOVE VALUE-OF-PURGE TO CHECKPOINTTYPE.
        CALL CALLCHECKPOINT
        USING
               CHECKPOINTDEVICE
               CHECKPOINTTYPE
               COMPLETIONCODE
               CHECKPOINTNUMBER
               RESTARTFLAG
        GIVING
               RSLT.
```

Creating Output Disk Files with a Checkpoint

Invoking a checkpoint causes the creation of one or more of the following checkpoint-related disk files: a checkpoint file, a checkpoint job file, and checkpoint temporary files. The following paragraphs explain what these files are, and the factors that determine whether they are created.

A checkpoint file is always created if the checkpoint is successful. This file stores a complete description of the checkpointed task and is titled according to the following convention:

```
(<usercode>)CP/<job number>/<checkpoint number> ON <family name>
```

The usercode in the checkpoint file title is the usercode of the task. The job number is the 5-digit mix number of the job that initiated the task. The checkpoint number is a 3-digit number used to distinguish this checkpoint from any other checkpoints executed by the same task. The family name is taken from the value of the device option in the CHECKPOINT statement.

If the disposition option is set to PURGE, the checkpoint number is always 0 (zero) and each succeeding checkpoint with PURGE set removes the previous checkpoint file. If the disposition option is set to LOCK, the checkpoint number starts at a value of 1 for the first checkpoint, and is incremented by 1 for each succeeding checkpoint that is invoked with LOCK. If a task invokes some checkpoints with LOCK and some with PURGE, then the "locked" checkpoints use ascending checkpoint numbers and the "purged" checkpoints use a checkpoint number of 0.

A checkpoint job file is produced by the first checkpoint in the task that is invoked with LOCK. This checkpoint job file makes it possible to restart the checkpointed task after its original job has terminated. Later checkpoints with LOCK do not produce a job file, nor do any checkpoints invoked with PURGE. The checkpoint job file is titled according to the following convention:

```
(<usercode>)CP/<job number>/JOBFILE ON <family name>
```

Checkpoint temporary files store the contents of temporary files in use by the checkpointed task. Checkpoint temporary files are created only in certain circumstances, which are discussed under "Planning for File Recovery" earlier in this section. These files are titled according to the following convention:

```
(<usercode>)CP/<job number>/F<file number> ON <family name>
```

In the checkpoint temporary file title, the file number is a 3-digit file number that starts at 1 and is incremented by 1 for each temporary disk file.

The way the checkpointed task terminates can have an effect on the checkpoint-related files. If the checkpointed task terminates abnormally and the last checkpoint has a disposition of PURGE, the system retitles the checkpoint file to have the next sequential checkpoint number and creates a checkpoint job file if none exists. If the checkpointed task terminates normally and all checkpoints have a disposition of PURGE, then the system removes all checkpoint-related files that were created for the task.

For tasks that invoke a large number of checkpoints with the LOCK disposition, the checkpoint number is incremented up to 999 and then is recycled to 1 (leaving 0 undisturbed). When this happens, the checkpoint files previously numbered 1, 2, and so on are lost as new ones using those numbers are created.

When a task restarts at a checkpoint that was not the last, subsequent checkpoints invoked from the restarted task continue in numerical sequence from the one used for the restart. Old high-numbered checkpoints are thus lost.

If a rerun is initiated and the original job number is in use by another task, then a new job number is assigned to the job. The titles of all checkpoint-related files for the task are changed to reflect the new job number.

Restrictions on the Use of Checkpoints

There are certain restrictions on the features that can be in use by a task when it is checkpointed. These restrictions apply to both programmatically initiated checkpoints and operator initiated checkpoints. If any of these features are in use, they prevent a successful checkpoint. If a checkpoint fails, the task continues normally, but no checkpoint files are created.

The following restrictions apply to the tasking environment of the checkpointed task:

- The task must have been initiated by a RUN statement in a WFL job. The checkpointed task must be the only in-use offspring of the WFL job at the time the checkpoint is invoked.
- The task cannot be a remote task. That is, it must not be initiated on a BNA host system other than that on which the job is running.
- The task must not have any offspring at the time of the checkpoint. However, the task can have offspring at earlier, or later, points in its execution.

A checkpoint cannot be invoked from within the following types of procedures:

- An imported library procedure. The checkpoint cannot take place if an imported library procedure is anywhere in the process stack. However, the checkpoint can be invoked if the user process is merely linked to a library.
- A SORT input or output procedure. (SORT provides its own restart capability; refer to the *System Software Utilities Operations Reference Manual*.)

Several types of files cannot be open at the time of the checkpoint. However, the process can close these files, take a checkpoint, and then reopen the files and continue to use them. The following are the restricted types of files:

- Direct files
- Disk files whose DUPLICATED attribute value is TRUE
- Files whose FILESTRUCTURE attribute value is not ALIGNED180
- ISAM files
- Multireel unlabeled tape files

- ODT files
- Remote files
- Port files
- Reversed tape files

Some restrictions also apply to printer output. All open printer files must be backup files, and have a PRINTDISPOSITION file attribute value of EOJ or DONTPRINT. In addition, the BDBASE option of the OPTION task attribute cannot be set.

The process cannot have an Enterprise Database Server database open at the time of the checkpoint.

No direct arrays can be in the process stack at the time of the checkpoint. A direct array can be declared in a procedure in the program. However, the procedure must not have been entered, or must have been entered and exited, before the checkpoint.

The checkpoint file cannot be created if doing so would cause the user's file usage on a family to exceed the limits enforced for the user by the disk resource control (DRC) system. For information about the disk resource control system, refer to the *System Administration Guide* and the *System Operations Guide.*

Determining Eligibility for Checkpoints

A task can determine whether it is probably eligible for checkpoints by interrogating the CHECKPOINTABLE task attribute. This read-only Boolean attribute is assigned by the system. The system assigns a value of FALSE if the task does not meet certain basic requirements for a checkpointed task. However, a CHECKPOINTABLE value of TRUE does not guarantee that a checkpoint will succeed. For details, refer to the discussion of CHECKPOINTABLE in the *Task Attributes Programming Reference Manual*.

Determining Whether the Checkpoint Succeeded

The checkpoint facility returns a value indicating the result of the attempted checkpoint. This value is divided into the following fields:

[46:01]	If this bit is set, then the current task was restarted from this checkpoint.
[25:12]	If the checkpoint succeeded, this field stores the checkpoint number assigned to the checkpoint files.
[10:10]	If the checkpoint failed, this field stores the completion code that indicates why the checkpoint failed. For a list of the possible completion codes, refer to Table 11–1.
[00:01]	This is the exception bit. If this bit is set, then either the checkpoint did not succeed or the process was restarted from this checkpoint.

In ALGOL, you can store the result value in a Boolean variable by invoking the checkpoint facility as a function. This Boolean value can then be stored in a real variable, and the various fields of the real variable can be conveniently interrogated.

In the following ALGOL example, BOOL is a Boolean variable and the other variables are real.

```
BOOL:= CHECKPOINT(PACK,LOCK);
REALRSLT:= REAL(BOOL);
CPRESTART:= REALRSLT.[46:01];
CPNUMBER:= REALRSLT.[25:12];
CPCOMPLETION:= REALRSLT.[10:10];
CPEXCEPTION:= REALRSLT.[00.01];
```

You can tell whether a checkpoint was successful by observing the completion message that is displayed. The following is an example of a successful completion message:

```
#1082 CHECKPOINT #1080/001 TAKEN @ (029900)*
```

The following is an example of the completion message for a checkpoint that failed:

```
#1111 CHECKPOINT ABORTED: BAD IPC ENVIRONMENT @ (029900)*
```

Each completion message corresponds to one of the completion codes from field [10:10] of the checkpoint result. The completion messages are listed in Table 11–1.

Table 11–1. Checkpoint Completion Codes

Completion Code	Completion Message and Meaning
0	CHECKPOINT #<mix number>/<checkpoint number> TAKEN
	The checkpoint was executed successfully.
1	INVALID AREA IN STACK
2	SYSTEM ERROR
	Completion errors 1 and 2 both mean that a system error occurred.
3	BAD IPC ENVIRONMENT
	The process has offspring or was not initiated by a WFL *RUN* statement.
4	NO USER DISK FOR CP FILE
	The family requested by the device option in the checkpoint statement is not available.
5	IO ERROR DURING CHECKPOINT
	An I/O error occurred.
6	# ROWS IN CP FILE > 1024
	The process is too large to be successfully checkpointed.
7	DIRECT FILE NOT ALLOWED
	The process has a direct file that is open.

Table 11–1. Checkpoint Completion Codes

Completion Code	Completion Message and Meaning
8	TOO MANY TEMPORARY DISK FILES The process has more than 998 temporary files.
9	ILLEGAL FILEKIND The process is using a file for writing directly to a line printer or a card punch.
10	DUPLICATED FILE NOT ALLOWED The process is using a duplicated file.
11	ILLEGAL FILE ORGANIZATION The process is using an Index Sequential Access Method (ISAM) file.
12	INSUFFICIENT MEMORY TO CHECKPOINT Not enough memory is available to checkpoint the process.
13	OPEN REVERSED TAPE FILE NOT ALLOWED The process is using a reversed tape file.
14	ICM AREA IN STACK The process is using a BNA Version 2 port file.
15	DMS AREA IN STACK The process is using an Enterprise Database Server data set.
16	DIRECT ARRAY IN STACK The process has entered, and not yet exited, a block that includes a direct array declaration.
17	SECURITY ERROR SAVING TEMPORARY DISK FILE The process has a temporary file open under another usercode. This situation can occur, for example, if both the following are true: • The process opened a permanent disk file that resided under someone else's usercode. • While the process had the file open, another process attempted to remove the file, thus changing it to a temporary file.
19	STACKMARK A system error occurred.
20	SORT AREA IN STACK The process is using the SORT function.
21	IN USE ROUTINE NOT ALLOWED The process has entered a USE procedure.

Table 11–1. Checkpoint Completion Codes

Completion Code	Completion Message and Meaning
22	ILLEGAL CONSTRUCT
	Either the process has opened a port file or there is operating system code in the process stack. The latter can occur, for example, when a fault causes the execution of an ALGOL *ON* statement.
23	BDBASE ILLEGAL
	The BDBASE option of the OPTION task attribute has been set.
24	ILLEGAL FILESTRUCTURE
	The process has an open file with a FILESTRUCTURE value for which checkpointing is not implemented.
25	MULTI-REEL UNLABELED TAPE NOT ALLOWED
	The process has opened a multireel unlabeled tape file.
26	SURROGATE TASK NOT ALLOWED
	The task was initiated on a BNA host other than that on which the job is running.
27	NON-EVENT PO IN STACK
	The process is using the MCP post office box facility.
28	PROGRAM USES LIBRARIES
	The process is executing an imported library procedure.
30	ROW SIZE TOO SMALL FOR CP FILE
	The process stack is too large to fit in a row of the checkpoint file. The maximum size of a process stack that can be checkpointed is approximately 22700 words.
32	OPERATOR CHECKPOINT REQUEST CANCELED
	A checkpoint or restart was already underway.
34	BR REQUEST REJECTED
	The BRCLASS task attribute value is NOBR.
36	OPEN BACKUP FILE WITH PRINTDISPOSITION = EOT NOT ALLOWED
37	OPEN BACKUP FILE WITH PRINTDISPOSITION = CLOSE NOT ALLOWED
38	OPEN BACKUP FILE WITH PRINTDISPOSITION = DIRECT NOT ALLOWED
	Each of these three values means that the PRINTDISPOSITION file attribute has a value not allowed for checkpoints.
40	ATTEMPT TO EXCEED TEMPORARY FILE LIMIT ON CP FILE
41	ATTEMPT TO EXCEED FAMILY LIMIT ON CP FILE

Table 11–1. Checkpoint Completion Codes

Completion Code	Completion Message and Meaning
42	FAMILY INTEGRAL LIMIT EXCEEDED ON CP FILE These three values mean the checkpoint file cannot be created because doing so causes the user's file usage on a particular family to exceed the limits set by the disk resource control system. For information about the disk resource control (DRC) system, refer to the *System Administration Guide* and the *System Operations Guide.*
43	INVALID ENVIRONMENT IN STACK The process has invoked and not yet exited a library procedure. Either the process is currently executing code from that procedure, or it is executing code from some other procedure that was invoked from the library procedure.
44	DISK TYPE MUST BE DISK OR PACK A process invoked the exported MCP procedure CALLCHECKPOINT and passed a value of other than 1 (disk) or 17 (pack) to the UTYP parameter.
45	CHECKPOINT TYPE MUST BE ZERO OR ONE A process invoked the exported MCP procedure CALLCHECKPOINT and passed a value of other than 0 (purge) or 1 (lock) to the CPTYP parameter.
46	SHARED BUFFERS NOT ALLOWED The process has a file open for which the value of the BUFFERSHARING file attribute is not NONE.
47	STACK CONTAINS STRUCTURE TYPE VARIABLES The process includes structure block declarations.
48	LIBRARY IS LINKED TO A CONNECTION LIBRARY ENVIRONMENT The process is linked to a connection library.
49	PROGRAM USES CONNECTION LIBRARIES The process includes a connection library declaration.
50	PROGRAM USES SIGNALS The process uses the POSIX signals mechanism.
51	PROGRAM USES FILE DESCRIPTORS
52	FIFO NOT ALLOWED
53	POSIX SPECIAL FILES NOT ALLOWED
54	PRINTDISPOSITION OF OPEN BACKUP FILE MUST BE EOJ OR DONTPRINT The process has an open backup file with a PRINTDISPOSITION value other than EOJ, DONTPRINT, EOT, CLOSE, or DIRECT. PRINTDISPOSITION values of EOT, CLOSE, and DIRECT generate other completion codes.
55	PRINTER FILES MUST BE BACKUP An open printer file is not a printer backup file.

Operator-Invoked Checkpoints

You can initiate checkpoints for a task by using the BR (Breakout) system command. This feature is designed to allow you to checkpoint tasks when an external condition prevents execution from continuing. For example, checkpointing a task just before halt/loading the system preserves the work done up to that point.

The BR command, if it is completed successfully, has the same effect as a CHECKPOINT statement in a program. All restrictions that apply to a programmed checkpoint also apply to an operator-initiated checkpoint. For example, the task must have been initiated from a WFL job and must not have any offspring. The task must be written in ALGOL or COBOL74. For details about these restrictions, refer to "Restrictions on the Use of Checkpoints" earlier in this section.

The programmer is responsible for designing a task to recover data file contents and libraries after a restart. It can be difficult to design such recovery mechanisms without knowing exactly when the checkpoint will take place. You can overcome this difficulty through the use of the BRCLASS task attribute. This task attribute specifies whether the task currently allows an operator-invoked checkpoint.

Programmatically Preventing Operator Checkpoints

You can use the BRCLASS task attribute to specify how a task will respond to an operator-invoked checkpoint. Using repeated assignments to BRCLASS, you can specify that operator checkpoints be allowed at some points in the task and not at others.

You can disallow operator checkpoints by assigning BRCLASS a value of NOBR (the default). You can allow a single checkpoint, and cause the task to be discontinued automatically after the checkpoint, by assigning BRCLASS a value of ONCEONLY. You can allow multiple checkpoints, and allow the task to continue normally after each checkpoint, by assigning BRCLASS a value of MULTIPLE.

Note: *Multiple operator checkpoints are possible only if BRCLASS is set to MULTIPLE for both the job and the task. It is not sufficient to assign MULTIPLE to the task alone.*

The BRCLASS task attribute has effect only if the CHECKPOINTABLE task attribute is TRUE.

Displaying the Checkpoint Status

You can use the *<mix number> BR* system command to determine whether a task is eligible for an operator checkpoint and whether the task is currently being checkpointed or restarted. The response has the following form:

```
TASK <mix number> <checkpoint status>
```

The following are possible responses if the task is not currently being checkpointed or restarted. The phrase "CANNOT CONTINUE AFTER BR" indicates that the BRCLASS task attribute has a value of ONCEONLY. When BRCLASS = ONCEONLY, the system automatically discontinues the process after the checkpoint completes. However, the process can continue if the operator cancels the checkpoint with an OF (Optional File) system command, as discussed under "Operator Actions after the Checkpoint" later in this section.

```
TASK <mix number> IS NOT CHECKPOINTABLE BY THE OPERATOR
TASK <mix number> IS CHECKPOINTABLE
TASK <mix number> IS CHECKPOINTABLE (CANNOT CONTINUE AFTER BR)
```

The following are responses that indicate that a checkpoint has been requested or is underway:

```
TASK <mix number> CHECKPOINT REQUESTED
TASK <mix number> CHECKPOINT REQUESTED (CANNOT CONTINUE AFTER BR)
TASK <mix number> CHECKPOINT RUNNING
TASK <mix number> CHECKPOINT RUNNING (CANNOT CONTINUE AFTER BR)
```

The following responses indicate that a restart is underway. In these responses, the phrase "PROGRAM" means that the checkpoint was initiated by a CHECKPOINT statement in the task. The phrases "ONCEONLY" and "MULTIPLE" specify the value of the BRCLASS task attribute in cases where the checkpoint was operator initiated.

```
TASK <mix number>: RESTARTING
TASK <mix number>: RESTARTING (PROGRAM)
TASK <mix number>: RESTARTING (ONCEONLY)
TASK <mix number>: RESTARTING (MULTIPLE)
```

The Y (Status Interrogate) system command displays more limited information about the checkpoint status of a task. If a checkpoint or restart action has been requested for a task, a line of the following form appears in the Y display:

```
CHECKPOINT STATUS: <status>
```

The possible <status> values are REQUESTED, RUNNING, or RESTARTING. These values have the same meaning they do in the BR display.

Invoking a Checkpoint Interactively

You can invoke a checkpoint for a task by entering the *<mix number> BR +* form of the BR command. If the checkpoint is accepted, it is executed with DISK as the device option and PURGE as the disposition option. Checkpoint files are therefore created on DISK family, with a checkpoint number of 0.

If the BRCLASS task attribute value is ONCEONLY, the task is discontinued after the checkpoint.

The system might delay execution of the checkpoint request if it is not immediately able to save the task stack correctly. For example, a checkpoint request cannot be completed while the task is waiting on an event. If the checkpoint request is being delayed, a BR command shows a checkpoint status of REQUESTED.

Canceling a Checkpoint Interactively

If the checkpoint status is REQUESTED, you can cancel the checkpoint request by entering a BR command of the form *<mix number> BR –*. This command cancels the request immediately.

Operator Actions after the Checkpoint

As soon as the checkpoint has been taken successfully and the checkpoint file is entered into the directory, the checkpoint function waits for an operator action. The following is an example of the W (Waiting Mix Entries) system commands display for such a process:

```
-Job-Task-Pri-Elapsed--- 5 WAITING ENTRIES ---------------------
  6927\6928  50      1:44 (JASMITH) (JASMITH)OBJECT/ALGOL/CP ON SYSPK
       OPERATOR CHECKPOINT #6927/000 TAKEN  @ 112F:00EA:1 @ (00000500)
```

You can determine the possible responses to this waiting state by entering a *<mix number> Y* command. The REPLY line of the Y command display lists one or more of the following possible responses:

- <mix number> DS

  This command immediately discontinues the checkpointed task and its job. Any user protection, such as EPILOG procedures, will not be considered during the DS operation. This restriction ensures that neither the job nor the task will change the state of any of its files after the checkpoint has been taken. This response is always available after an operator-initiated checkpoint.

- <mix number> OF

  This command cancels the checkpoint, removes the files created by the checkpoint, and causes the checkpointed task and its job to continue their normal execution. This response can be used if you decide that the checkpoint was not needed.

- <mix number> OK

  This command causes the system to complete the checkpoint, and causes the checkpointed task and its job to continue execution normally after the checkpoint. Any files created by the checkpoint are saved. This response is allowed only if the BRCLASS task attribute value was MULTIPLE.

Additionally, if the task was checkpointed in preparation for a halt/load, the ??PHL (Programmatic Halt Load) system command can be used to initiate the halt/load. After the halt/load, you can enter a command to restart the task.

Restarting a Checkpointed Task

A checkpointed task can be restarted automatically after a halt/load or explicitly with a WFL *RERUN* statement.

Restarting Checkpointed Tasks Automatically

If a WFL job was executing a checkpointed task when a halt/load occurred, the job does not immediately restart after the halt/load. Instead, an independent runner called JOBRESTART appears in the W (Waiting Entries) system command display. The following is an example of the entry:

```
---Mix-Pri---Elapsed------------ 2 WAITING ENTRIES -----------------
  7082  50       :55  JOB  JOBRESTART
       RESTART PENDING 7119 DAILY/RUNNIT
```

In the preceding example, DAILY/RUNNIT is the job that is pending restart, and its mix number is 7119. The checkpointed task does not appear in the display. JOBRESTART is a job that was initiated by the system software to do the restarting, and its mix number is 7082. The following are the possible operator responses to this example and the effects of the responses:

- 7082 OK

  This command restarts the job and restarts the task at the last checkpoint.

- 7082 DS

  This command discontinues the job and the task. Any checkpoint files are saved, regardless of whether the disposition was LOCK or PURGE. The checkpoint number of PURGE files is left as 0 (zero).

- 7082 QT

  In this context, QT has the same effect as DS.

Initiating a Restart Explicitly

You can use a WFL *RERUN* statement to restart a checkpointed task. The checkpoint files of the task to be restarted must have been permanently saved. Checkpoint files are permanently saved if the checkpoint disposition is LOCK, if the job terminates abnormally, or if the checkpoint is initiated by an operator BR command.

The RERUN statement can be included in a WFL job. Also, you can enter the RERUN statement directly at the ODT, in which case the RERUN statement causes the creation of a WFL job that does the restart. The RERUN statement has the following form:

```
RERUN <job number> / <checkpoint number>
```

In the RERUN statement, the job number is the mix number of the job that initiated the checkpointed task. The checkpoint number identifies the checkpoint that is to be used.

If the checkpointed task had a usercode, the checkpoint files are stored under that usercode. To restart such a task, you must enter the RERUN command in a job that specifies the usercode. The following can then be entered at an ODT:

```
?BEGIN JOB;USERCODE = <usercode> / <password>;
  RERUN <job number> / <checkpoint number>
```

Following are some of the conditions that can prevent a successful restart:

- The usercode of the checkpointed task or its job is no longer valid.
- The program has been recompiled since the checkpoint was created.
- The system is now running on a different MCP release level than it was when the checkpoint was created. For example, the system is now running a 48.1 MCP, and the checkpoint was created on a system running a 47.1 MCP.
- The system is now using different intrinsics from when the checkpoint was taken.
- The checkpoint files are not present on DISK family or PACK family. The files must be on one of these two families, regardless of any FAMILY equations entered with the RERUN statement.
- The process was restarted on a different type of machine from the one where the checkpoint was taken. For example, the process was checkpointed on an LX5100 and restarted on an NX5820.

If a rerun is initiated and the job number is in use by another job, a new job number is assigned and the checkpoint files are automatically retitled to reflect the new job number.

The following messages can be displayed to show the result of the restart attempt:

- RESTART PENDING
- RESTART INITIATED
- RESTART ABORTED: MISSING CHECKPOINT FILE
- RESTART ABORTED: IO ERROR DURING RESTART
- RESTART ABORTED: USERCODE NO LONGER VALID
- RESTART ABORTED: OPERATOR DSED RESTART
- RESTART ABORTED: OPERATOR QTED RESTART
- RESTART ABORTED: MISSING CODE FILE
- RESTART ABORTED: NOT ABLE TO RESTART
- RESTART ABORTED: INVALID JOB FILE
- RESTART ABORTED: ERR COPYING JOB FILE
- RESTART ABORTED: MISSING JOB FILE
- RESTART ABORTED: FILE POSITIONING ERROR
- RESTART ABORTED: WRONG JOB FILE
- RESTART ABORTED: WRONG CODE FILE
- RESTART ABORTED: BAD CHECKPOINT FILE
- RESTART ABORTED: BAD STACK NUMBER
- RESTART ABORTED: WRONG MCP
- RESTART ABORTED: MISSING FAMILY MEMBER

- RESTART ABORTED: MACHINE TYPES DIFFER
- RESTART ABORTED: PAGED ARRAY PAGE SIZE HAS CHANGED
- RESTART ABORTED: FILE IS RESTRICTED
- RESTART ABORTED: FILE IS ON A RESTRICTED FAMILY
- RESTART ABORTED: TAPE LABELKIND CONFLICTS WITH FILEUSE

Using the CHECKPOINTARRAY Procedure

You can use the CHECKPOINTARRAY procedure to write the contents of arrays to disk files. CHECKPOINTARRAY can also restore arrays from the disk files where they were previously stored. The CHECKPOINTARRAY procedure is exported by the MCPSUPPORT library.

The CHECKPOINTARRAY procedure is not part of the checkpoint facility described earlier in this section, but serves a related purpose. The following table compares the features provided by CHECKPOINTARRAY procedure with those provided by the checkpoint facility.

Table 11–2. Comparison of CHECKPOINTARRAY and Checkpoint Facility

Features	CHECKPOINTARRAY	Checkpoint Facility
How invoked	CHECKPOINTARRAY procedure	CHECKPOINT statements, CALLCHECKPOINT procedure, or BR (Breakout) system command
Types of data stored by the checkpoint	Most types of arrays. Can be multidimensional. Must contain data. *Cannot* be any of the following: • Direct array • Double array • Event array • Interlock array • Queue array • Value array • Message • Array declared in a structure block or connection block • ALGOL string	All types of data objects, except for files.
Programming languages	ALGOL and NEWP (CHECKPOINTARRAY requires a parameter of type ANYTYPE, which is supported only by these languages)	ALGOL, C, COBOL74, COBOL85, FORTRAN77, NEWP, and Pascal
Restrictions on use	None, except that certain types of arrays are excluded	Numerous, including restrictions against using direct I/O, port files, Enterprise Database Server data sets, and many other features. For details, refer to "Restrictions on the Use of Checkpoints" earlier in this section.

The following paragraphs present a brief explanation and example of the CHECKPOINTARRAY procedure.

The CHECKPOINTARRAY procedure has the following parameters and result:

Parameter	Explanation
CPOPTIONS	A real variable that indicates whether the operation is a checkpoint or a restore, and whether the operation should abort or wait on an RSVP if certain conditions are encountered.
CPARRAY	The array to be checkpointed or restored. The array can be of any type, as long as it directly contains data. That is, the array cannot be a value array, a message array, a queue array, an event array, an interlock array, an ALGOL string, or an array in a structure block or connection block.
FTITLE	A real array that stores the title of the checkpoint file in display form.
INFO	A real array of which word 0 is stored during a checkpoint and retrieved during a restore. You can use this array to store version information of your choice.
Procedure Result	A Boolean value indicating whether the operation succeeded, and if not, why not.

The following ALGOL program shows how these parameters can be used in CHECKPOINTARRAY calls:

```
BEGIN

LIBRARY MCPSUPPORT (LIBACCESS = BYFUNCTION,
                    FUNCTIONNAME = "MCPSUPPORT.");

BOOLEAN PROCEDURE CHECKPOINTARRAY(CPARRAY, FTITLE, CPOPTIONS, INFO);
VALUE CPOPTIONS; REAL CPOPTIONS;
ANYTYPE CPARRAY;
ARRAY FTITLE, INFO[*];
   LIBRARY MCPSUPPORT;

DEFINE                  % CPOPTIONS parameter fields
    CHECKPOINTING = 1[0:1] #,
    RESTORING     = 0[0:1] #,
    NORSVP        = 1[1:1] #,
    RSVP          = 0[1:1] #;

DEFINE                  % Procedure result error field and numbers
    ERRNUMF = [15:8] #,
    NOCPAFILE     =   1 #,   % Restore: File not found or not accessible
    NOTACPAFILE   =   2 #,   % Restore: File not a checkpoint file
    BADFTITLE     =   3 #,   % CP or Restore: Invalid file title
    NOPACKFAMILY  =   4 #,   % CP or Restore: Family name not online
    NODATA        =   5 #,   % CP: Untouched / empty array
    DIFFERENTDIMS =   6 #,   % Restore: Array has different # of dims
    MISMATCH      =   7 #,   % Restore: Wrong array type / element-size
    BADARRAY      =   8 #,   % CP: Non-array or unsupported array type
    RESIZED       =   9 #,   % CP: User resized array during checkpoint
    DISKERROR     =  10 #,   % CP or Restore: Disk error occurred
    NOTENOUGHMEM  =  11 #,   % Restore: Not enough memory for array
    CALLERDSED    =  12 #,   % CP or Restore: Process was discontinued
                             %    (Dsed)
    WRONGVERSION  =  13 #,   % Restore: CPAFILE format incompatible with
                             %    current MCP
    WRONGPAGESIZE =  14 #,   % Restore: System has different page size
                             %    than when array was checkpointed
    NODISKSPACE   =  15 #,   % CP: Not enough disk space to checkpoint
    LOGICERROR    = 255 #;   % CP or Restore: MCP internal logic error

ARRAY TABLETHING[0:9999,0:9999];
ARRAY INFOW[0:1],
      CPTITLE[0:42];
BOOLEAN INITNEEDED,
        RSLT;

FILE CPFILE(KIND=DISK,TITLE="TABLETHING/CP.",NEWFILE=FALSE,
            DEPENDENTSPECS=TRUE);

% The following statements
%     - Check to see if the job was interrupted
%     - Check to see if a checkpoint array file is present
%     - Attempt to restore TABLETHING from the checkpoint file
%     - Check the mix number in the INFOW word to determine whether
%       the checkpoint was generated earlier within this job, or is an
%       obsolete remnant of a previous job run
REPLACE CPTITLE BY "TABLETHING/CP.";
INITNEEDED := TRUE;
IF MYJOB.RESTARTED AND CPFILE.RESIDENT THEN
   BEGIN
   IF NOT CHECKPOINTARRAY(TABLETHING,
                          CPTITLE,
                          0 & RESTORING & RSVP,
                          INFOW) THEN
      BEGIN
      IF INFOW[0] = MYJOB.MIXNUMBER THEN
         INITNEEDED := FALSE;
      END;
   END;

IF INITNEEDED THEN
   BEGIN
   % Initialize TABLETHING.
   % [Statements omitted from example here]

   % Then checkpoint the TABLETHING array to disk
   INFOW[0] := MYJOB.MIXNUMBER;
   RSLT := CHECKPOINTARRAY(TABLETHING,
870                             CPTITLE,
880                             0 & CHECKPOINTING & RSVP,
890                             INFOW);
900    END;
910
920 %%%% Begin Main Portion of Program %%%%
930
940 %  [ Statements omitted from example here]
950
960 END.
```

The preceding program includes only the code relevant to checkpointing and restoring an array.

The program initializes an array named TABLETHING at the start of each run. The array is assumed to be very large and costly to initialize. If the program is interrupted by a halt/load, then it is cheaper to restore the array from a checkpoint than to reinitialize the array.

The program is designed to run from a WFL job. If the program and its parent WFL job are interrupted by a halt/load, then when the job is restarted, the RESTARTED attribute of the job is TRUE. Therefore, at line 660 the program checks the RESTARTED attribute of the WFL job. If the job was restarted and a checkpoint file with the appropriate title is available, then the statements at 680 to 710 attempt to restore the TABLETHING array.

Line 730 checks the INFOW value returned by the CHECKPOINTARRAY procedure. This program always stores the mix number of the job in the INFOW array when performing a checkpoint. Note that if a WFL job is interrupted by a halt/load and restarted, the WFL job is restarted with its original mix number. Therefore, the program can use the INFOW value to determine whether the checkpoint was created earlier by this same job (before interruption by a halt/load).

If no checkpoint file was recovered, or if the recovered checkpoint file was not created by the current job, then the program initializes the array itself. The statements initializing the array would reside at line 820, but are omitted from this example.

Once the array is initialized, the statements at lines 850 to 890 checkpoint the array to a file.

The defines at lines 270 to 460 are not actually used in this example, but are included for your information. A program could use these defines to interrogate the procedure result and determine the cause of a CHECKPOINTARRAY failure.

Section 12
Tasking across Multihost Networks

The linking of systems into a multihost network provides the capability for a type of distributed processing. Each process executes on a single host system. However, the various members of a process family can run on different host systems and can communicate with each other in most of the same ways they could if they were all running on the same system.

This type of distributed processing is referred to as *remote tasking* and is provided by Host Services software. Remote tasking is supported across BNA Version 2, and Open Systems Interconnection (OSI) networks

This section uses some specialized terminology to discuss remote tasking. The term *remote process* is used to refer to a process that is initiated from one host system, but runs on another host system. The host from which the remote process is initiated is referred to as the *local host.* The host at which the remote process runs is referred to as the *remote host*.

In the same way, the *local operator* is an operator at the system from which the remote process is initiated. The *remote operator* is an operator at the system where the remote process runs.

A remote process can be initiated from programs or from interactive sources such as the operator display terminal (ODT), a Command and Edit (CANDE) session, or a Menu-Assisted Resource Control (MARC) session. Any messages generated by the process are routed back to the local ODT and originating terminal. You can monitor and control the remote process by transmitting ODT commands to the remote host system.

The following are reasons why you might want to initiate a remote process:

- To equalize the processor load on the various systems at an installation. If the local system is overloaded, a process may run more quickly at a remote host.
- To make use of a program stored at a remote host. A process must run on the same system where the object code file is stored. Therefore, initiating a program that is stored at another system implies the creation of a remote process.
- To efficiently access files stored on a remote host. A remote process running on the remote host can access these files more efficiently than a local process that accesses the files using Host Services logical I/O. The result can be savings in I/O time and elapsed time.

For further information about Host Services, other than remote tasking, refer to the *Distributed Systems Services (DSS) Operations Guide.* For additional information about Host Services logical I/O, refer to the *I/O Subsystem Programming Guide*.

Submitting Remote WFL Jobs

Any Work Flow Language (WFL) job can be designed to run on a remote host. Additionally, a local operator can initiate jobs that are stored on remote hosts.

Running a Local WFL Job on a Remote Host

In some cases, it might be convenient to store a WFL job source program on the local host, even though the job is to be run on a remote host. For these cases, you can include an *AT <hostname>* specification at the start of the job.

You can submit the WFL job for execution by entering a START command at the local host. If the *AT hostname* specification in the job requests a hostname that is not currently available, the system rejects the job and displays the message "SPECIFIED HOST NOT REACHABLE." If the requested hostname is available, the system transfers the job to the remote host. The entry "JOB/HANDLER/<local hostname>" appears in the mix at the remote host and indicates that a job has been transferred to the remote host. The job compilation, job queuing, and job execution all take place at the remote host.

If the *AT <hostname>* phrase is used, the job cannot include a job parameter list, any BINARY data specifications, or a null character within a quoted string. In addition, if the WFL source program is stored in a disk file, a question mark must be included before the END JOB statement. If the WFL source program is submitted in array form, it should not include any strings with embedded null characters; otherwise, the job receives a syntax error at the remote host.

The following is an example of the job heading for a job that is to run on a remote host named CHICAGO:

```
?AT CHICAGO BEGIN JOB REMOTE/RUNNER;
```

Submitting a WFL Job Stored on a Remote Host

If a WFL source program resides on a remote host, you can submit the WFL program for execution with the command *AT <hostname> START <file title>*. The following is an example of this command:

```
AT CHICAGO START (SMITH)REMOTE/RUNNER ON DPMAST
```

The WFL program is compiled and executed on the remote host where it resides.

A WFL job initiated in this way runs without a usercode in some circumstances. For a discussion of these circumstances, refer to "Usercode Identity" later in this section.

Meeting Remote Job Queue Requirements

You must be aware of the possibility that the job queue definitions on the remote host might be different from those on the local host. The job is enqueued on the remote host as if it were a local job submitted on that host. If the job does not qualify for any of the queues, it is discontinued.

The job queuing algorithm is outlined in “Selecting the Queue for a Job” in Section 4, “Tasking from Programming Languages.”

Initiating Non-WFL Remote Processes

You can initiate remote processes from a local session or a local process. Several restrictions apply to the features that can be used by the remote process.

Specifying the Remote Host

The HOSTNAME task attribute can be used to specify the remote host at which the process is to run. This task attribute can be assigned through a task equation or an assignment to the task variable before initiation.

The HOSTNAME task attribute can be accessed from ALGOL, COBOL, and WFL. Therefore, remote processes can be initiated from any of these languages. For example, the following ALGOL statements initiate a remote process:

```
PROCEDURE RUNNER;
  EXTERNAL;
REPLACE T.NAME BY "OBJECT/RUNNIT ON DPPACK.";
REPLACE T.HOSTNAME BY "SEATTLE.";
CALL RUNNER [T];
```

CANDE and MARC also enable HOSTNAME to be included as a task equation following a RUN statement. The following is an example of a CANDE command that initiates a remote process:

```
RUN RUNNIT;HOSTNAME=MIAMI
```

The equivalent statement in MARC is as follows:

```
RUN OBJECT/RUNNIT;HOSTNAME=MIAMI
```

Limitations on a Non-WFL Remote Process

The following restrictions apply to a remote process that is not a WFL job:

- The remote process must be an external process whose object code file is stored on the remote host.
- The remote process can be passed no more than one parameter. The parameter must be a real array of one dimension. The actual parameter must have a zero lower bound. The system automatically chooses a passing mode of call-by-value for the parameter.
- The WFL *COMPILE* statement cannot cause the resulting object code file to be executed as a remote process. For example, suppose the compiler equation COMPILER HOSTNAME = SFA15C is used. The compilation will run successfully on the foreign host with a disposition of LIBRARY or SYNTAX, but if the disposition is GO or LIBRARY GO, it is rejected.

If one of the preceding restrictions is violated, the initiating process is discontinued with HISTORYCAUSE = 2 (PROGRAMCAUSEV) and HISTORYREASON = 31 (ILLEGALTASKXFERV). The following error message is displayed:

```
ILLEGAL HOST-TO-HOST TRANSFER OF TASK
```

Another restriction is that a WFL job cannot use global file assignments for remote tasks initiated by the job. For example, the following sequence of statements is illegal:

```
FILE IN(KIND=DISK,TITLE=NEW/INPUT/DATA);
RUN OBJECT/UPDATE;
   HOSTNAME = ALBANY;
   FILE CARD:= IN;
```

Global file assignments have no effect when applied to remote tasks initiated from WFL. The remote task executes normally, but the file used by the task does not receive any of the file attributes specified for the global file in the WFL job. When the remote task opens the file, the following nonfatal attribute error message is displayed:

```
[<hostname>] <mixno> ATTRIBUTE ERROR:<file internal name>.GLOBALFILESIRW
```

A remote task initiated from a local WFL job cannot read from any data specifications in the WFL job. When the remote task attempts to read from a data specification, it is suspended with a "NO FILE" condition and waits for a card reader file with the requested title to appear. An RSVP message such as the following is routed back to the local host:

```
[ALBANY] 2079 RSVP (JASMITH)OBJECT/UPDATE ON USERPK. NO FILE CARD (CR)
```

A coroutine cannot use a continue statement to transfer control to a coroutine on a remote host. By default, the PARTNER task attribute of a remote task is treated as MYSELF and the PARTNEREXISTS task attribute of a remote task returns a value of FALSE. In this case, any continue statement executed by the remote task has no effect. Execution simply proceeds to the next statement in the remote task.

The MYJOB task variable of a remote task is treated as a reference to the DSSSUPPORT library on the remote host. Any references to MYSELF.EXCEPTIONTASK in the remote task are treated as references to TASKING/MESSAGE/HANDLER, a task initiated by DSSSUPPORT on the remote host. TASKING/MESSAGE/HANDLER is discussed under "Displaying TASKING/MESSAGE/HANDLER and TASKING/STATE/CONTROLLER" later in this section. Any references to MYJOB.EXCEPTIONTASK in the remote task are treated as references to the remote task itself.

Any files accessed by a remote process are searched for on the remote host by default. If the remote process uses a file on the local host, the HOSTNAME file attribute must be assigned. For the remote process to open a remote file on the local host, the process must also set its STATION task attribute to zero and assign the desired station name to the FILENAME task attribute. Note that the STATIONNAME task attribute is ignored when the HOSTNAME file attribute is non-null.

Host Availability

If a process attempts to initiate a task at a remote host that is nonexistent or currently unavailable, the initiation fails, but the initiating process continues normally. The task variable of the task stores a STATUS value of -2 (BADINITIATE), a HISTORYTYPE of 4 (DSEDV), and a HISTORYCAUSE of 13 (NETWORKCAUSEV). The HISTORYREASON value varies depending on the exact reason the host is unreachable.

A pair of messages such as the following is displayed when this error occurs:

```
6749 TASK NOT INITIATED AT TESTSYS: ERROR - HOST NOT REACHABLE
6749 FOREIGN TASK INITIATION FAILED @ 103A:0001:4 @ (00000234)*
```

Initiating Processes from a Remote Session

An alternate method of initiating a process on a remote host is to initiate it from a remote CANDE or MARC session. You can establish a remote session by using the Station Transfer feature provided by Host Services. A process initiated from such a session is considered a local process because the session is under the direct control of the remote host. The process is therefore not limited by any of the restrictions previously discussed under "Limitations on a Non-WFL Remote Process" in this section.

For a detailed discussion of Station Transfer, refer to the *Distributed Systems Services (DSS) Operations Guide*.

Interrogating the Remote Ancestry of a Process

A process can find out which host system it is running on by interrogating its own HOSTNAME task attribute. This feature makes it possible to write a single program that will take different actions when it is run on different systems.

A process can interrogate its remote ancestry by inspecting the ITINERARY task attribute. This task attribute stores the hostnames of the host systems where each of the ancestors of the process is running. This task attribute can be useful for cases where the process needs to transmit information back to the user and thus needs to know where the user is located.

Preventing User Identity Problems

The user identity of a process consists of several related task attributes, including USERCODE, ACCESSCODE, CHARGE, FAMILY, GROUPCODE, and SUPPLEMENTARYGRPS. Each system in a multihost network has its own USERDATAFILE, which stores definitions of the users that are allowed on the system. These definitions can be different on different host systems. For a remote process to run successfully, it must be assigned an identity that is recognized on the remote host.

Usercode Identity

The most basic user identity requirement is that a remote process must run with a usercode that is allowed at the remote host, or it must run without a usercode.

If the remote process has a usercode, then the usercode must be one that is permitted as a remote user at the remote host. Remote users are defined by REMOTEUSER entries in the USERDATAFILE of the remote host. REMOTEUSER entries can specify in detail the hosts that can submit a process with a particular usercode. The following is an example of a REMOTEUSER entry at a remote host that allows processes with usercode JASMITH to be initiated from the host named CHICAGO:

```
RU JASMITH OF CHICAGO
```

In addition to the REMOTEUSER entry, there must be a USER entry for the usercode of the process in the USERDATAFILE at the remote host. A USER entry defines a usercode and assigns usercode attributes to the usercode.

A system administrator at the remote host can cause remote processes that request a particular usercode to be run under a different usercode instead. The substitute usercode is referred to as a *local alias usercode.* A remote process assumes a local alias usercode if the REMOTEUSER entry at the remote host specifies a local alias for the requested usercode. The local alias usercode must also be defined by a USER entry in the USERDATAFILE at the remote host. The following is an example of a REMOTEUSER entry that specifies a local alias usercode:

```
RU JASMITH OF CHICAGO LOCALALIAS=JOHNSMITH
```

The following is an example of a USER entry for the local alias:

```
USER = JOHNSMITH
       MAXPW = 1
       PASSWORD = ?
       FAMILY DISK = SYSPK OTHERWISE DISK
       IDENTITY = "ALIAS FOR JASMITH FROM CHICAGO"
```

Local alias usercodes are intended for use in cases where two different users on two different systems happen to have the same usercode. Establishing local alias usercodes allows these users to run processes on each other's systems, but prevents them from accessing each other's files.

Alternatively, the system administrator can use a local alias usercode to cause many usercodes from remote systems to be mapped to a single usercode at the local system. This mechanism can be useful if the users need to have access to the same set of files.

If no local alias usercode is defined for the requested usercode, then the requested usercode must itself be defined by a USER entry in the USERDATAFILE at the remote host. Note that one or more of the usercode attributes can have different values on the remote host than they have on the local host. These differences do not prevent remote process initiation.

In addition, the usercode can have passwords on the remote host that are different from those defined for that usercode on the local host. If the remote process inherits the usercode of the local process, the password is implicitly changed to a password that is accepted at the remote host. However, if the remote process is explicitly assigned a usercode at initiation time, the password specified should be one of the passwords defined for the usercode at the local host. If the remote process changes its usercode after it is initiated, the process must specify a password that is allowed on the remote host.

A WFL job is the only type of remote process that can run without a usercode. The following are sources from which a remote WFL job can receive a usercode, listed in order of precedence:

1. Assignments to the USERCODE task attribute in the job attribute list.
2. The usercode of a session, if the job is submitted from a CANDE or MARC session.
3. The terminal usercode, if the job is submitted from an ODT. An operator can assign a terminal usercode to an ODT by using the TERM (Terminal) system command.
4. The host usercode of the system, if the job is submitted from an ODT or a nonusercoded MARC session. The host usercode is assigned by the HU (Host Usercode) system command. You can create a nonusercoded MARC session by logging on with an asterisk (*) at a SUPERUSER-capable station.

If a process does not receive a usercode from any of the first three sources listed, then the host usercode is evaluated for SYSTEMUSER status. If the USER entry for the usercode at the remote host assigns SYSTEMUSER status, then the job runs without a usercode. Otherwise, the host usercode is used as the usercode for the job.

A local process cannot initiate a nonusercoded remote process. In the first place, any null usercode explicitly assigned to a process is overridden at initiation time by inheritance from the parent. For example, if the local process assigns a null USERCODE value to a task variable, and then initiates a remote process with the task variable, then the null USERCODE assignment is ignored. The task initiates successfully, but inherits the usercode of its parent. In the second place, if the local process is nonusercoded, and it does not explicitly assign a usercode to the remote process, then the remote process inherits a null usercode. However, the system cannot initiate a remote process that has a null usercode, so the system displays error messages such as the following:

```
DISPLAY: 1807000 HOST SERVICES ERROR 17: USER ERROR - NO USERCODE
TASK OBJECT/ALGOL/TASK ON PACK NOT INITIATED AT PARIS: USER ERROR
   - NO USERCODE
FOREIGN TASK INITIATION FAILED @ 109E:0001:4 @ (00012500)*
```

For more information about usercode definitions and the REMOTEUSER command, refer to the *Security Administration Guide*.

Accesscode and Charge Validation

A remote process does not inherit the CHARGE task attribute value of its parent. If no CHARGE value is explicitly assigned to the remote process, it runs without a charge code. However, the remote process runs with a CHARGE value if one is explicitly assigned by a statement in the parent process. Note that usercode definitions in the USERDATAFILE can specify the range of CHARGE values that are valid for processes with a given usercode. If the CHARGEREQ attribute is set in the usercode definition at the remote host, then the remote task must have one of the CHARGE values defined by the CHARGECODE usercode attribute at the remote host.

The ACCESSCODE task attribute also is not inherited by remote processes, but it can be assigned. However, if it is assigned, it must be assigned a value that is allowed for the remote process usercode on the remote host.

Family and Group Code Identity

A remote task does not inherit the FAMILY, GROUPCODE, or SUPPLEMENTARYGRPS task attribute values of its parent. Instead, the remote task inherits the values of the corresponding usercode attributes in the usercode definition at the remote host. If any of these attributes is not specified in the usercode definition at the remote host, then by default that task attribute receives a null value.

The parent process can override the family default at the remote host by explicitly assigning a FAMILY value to the remote task. The parent process can override the FAMILY specification at the remote host by explicitly assigning a FAMILY value to the remote task, either with a task equation or with an assignment to the task variable of the remote task. However, the parent process cannot override the GROUPCODE or SUPPLEMENTARYGRPS values at the remote host because these attributes are read-only.

Logging of Remote Processes

The system logging and job logging responsibilities for remote process families are divided between the local host and the remote host.

System Log Entries

When a local process or session initiates a remote process, the local system log does not contain a log entry to record the event.

All system log entries for the remote process are made in the system log at the remote host. The remote process receives BOJ and EOJ log entries, even if it is actually a task. The BOJ log entry shows the originating unit as zero and includes a line called ITINERARY that specifies the host that initiated the process. The following is an example of this log entry:

```
BOJ  23586  *SYSTEM/LOGANALYZER ON DISK.
            CODE COMPILED: 07/02/2001 11:20:45 BY DCALGOL 48.140
            RELEASE ID: SSR 48.1 [48.138.000] (48.138.0019)
            QUEUE: 0, ORIGINATING UNIT: 0
            STACK NUMBER: 02DC, PRIORITY: 50
            USERCODE: JASMITH.  REALUSERCODE: JASMITH.
            ITINERARY: SANTAFE
```

Job Summaries for Remote Processes

The job summary for a process family is printed on the system where the job runs. Thus, for WFL jobs that include an *AT <hostname>* specification, the job summary is printed at the remote host. The same is true for WFL jobs started by an *AT <hostname> START* command. Any other independent remote processes, such as programs initiated by an ALGOL *RUN* statement, also print job summaries on the remote host.

When a local job initiates a remote task, a single entry is made in the job summary indicating that the remote task was initiated. No other job summary entries are made for the remote task. The following is an example of this entry:

```
17:55:18      1708  DISPLAY: [MIAMI] 3382 BOT (JAS)OBJECT/UPDATE/FILES.
```

Resource Limits for Remote Processes

Remote processes do not inherit resource limits from their local parents. For example, if a local job has a MAXPROCTIME limit, remote tasks do not inherit that limit. Furthermore, the local job cannot be terminated because of excessive resource usage by its remote tasks.

The only way resource limits are propagated across networks is by explicit assignment. Thus, a local process can initiate a remote task and assign it a MAXPROCTIME value. If the remote task uses more processor time on the remote host than MAXPROCTIME allows, the remote task is discontinued.

For information about how resource limits are propagated in a local process family, refer to Section 2, "Understanding Interprocess Relationships."

Interacting with Remote Processes

An operator at the local host system can use system commands to monitor or interact with processes running on remote host systems. A user at a MARC or CANDE session can also use MARC or CANDE commands to monitor or interact with processes running on remote host systems.

Viewing Remote Process Messages

In general, any messages generated by a remote process are routed back to the local host. These include "BOT" and "EOT" messages, display messages, accept messages, and RSVP messages. The following are the only exceptions to this rule:

- WFL jobs initiated by an *AT <hostname> START* command. No messages are returned to the local host for such a job. (On the other hand, messages are returned for WFL jobs that use an *AT <hostname>* specification in the job header.)
- Non-WFL independent processes. These include any remote processes initiated by an ALGOL or COBOL *RUN* statement.

Remote process messages appear in the MSG (Display Messages) system command display at the local host, prefixed by the hostname of the remote host, as in the following example. If the controller option SEPARATEMSGS is set, use the MSG NW system command to see these entries.

```
---Mix-Time--------------------- MESSAGES -------------------------
*  **  19:33 [PARIS] 1057 EOT (JASMITH)OBJECT/REPORTER ON DPMAST.
*  **  19:25 [PARIS] 1057 BOT (JASMITH)OBJECT/REPORTER ON DPMAST.
```

If the remote process was initiated from a CANDE or MARC session, the process messages are also routed back to the CANDE or MARC session. The following is an example of a CANDE command that initiates a remote task and the messages that are returned:

```
RUN REPORTER ON DPMAST;HOSTNAME=PORTLAND
  #RUNNING 6881 AT PORTLAND.
  #[PORTLAND] 6881 BOT (JASMITH)OBJECT/REPORTER ON DPMAST.
  #ET=1:00.2 PT=0.0 IO=0.2
  #[PORTLAND] 6881 EOT (JASMITH)OBJECT/REPORTER ON DPMAST.
```

The message "#ET=1:00.2 PT=0.0 IO=0.2" is the termination message displayed for the process by CANDE. The elapsed time, processor time, and I/O time displayed in this message summarize the resource usage accumulated by the process on the remote host.

Note that the local termination message for the process appears before the EOT message from the remote host. This occurs because there is a slight delay in the forwarding of messages from the remote host.

Local Operator Control of Remote Processes

At the local host, you can control and interrogate remote processes by using system commands prefixed with the phrase *AT <hostname>*. You can direct any system command to a remote host in this way. However, security restrictions can be implemented at the remote host to limit or prevent the execution of such commands.

The system provides a usercode for each system command that is directed to a remote host. If the ODT has a terminal usercode, the terminal usercode is used. You can assign a terminal usercode with the TERM (Terminal) system command. If there is no terminal usercode, the host usercode is used. You can assign a host usercode with the HU (Host Usercode) system command.

If a process uses the DCKEYIN statement to submit a system command with an *AT <hostname>* prefix, the system command is submitted under the usercode of the process that executed the DCKEYIN statement.

For the command to be accepted at the remote host, the associated usercode must have a USER entry and a REMOTEUSER entry in the USERDATAFILE at the remote host. Otherwise, an error occurs at the remote host and the command is not executed. The command is also rejected if no usercode is associated with it. (The command might not have a usercode if neither a terminal usercode nor a host usercode is defined.)

If the REMOTEUSER entry defines a local alias usercode, the system command becomes associated with the local alias usercode. In this case, the USERDATAFILE must include a USER entry for the local alias usercode.

The remote host inspects the USER entry of the associated usercode to find out whether SYSTEMUSER status is set for the usercode. If SYSTEMUSER status is set for the usercode, then the system command is always allowed. If the usercode is not a SYSTEMUSER, then only a limited subset of the system commands can be used.

If the command usercode does not have SYSTEMUSER status, then the output of mix display commands is filtered so that only processes running under the command usercode are displayed. Likewise, commands that specify a particular process can only be applied to processes running under the command usercode. The following are the tasking-related commands that are available:

- AX (Accept)
- C (Completed Mix Entries)
- CU (Core Usage)
- DBS (Database Stack Entries)
- DS (Discontinue)
- DUMP (Dump Memory)

- FA (File Attribute)
- FI (File Information)
- FR (Final Reel)
- HI (Cause EXCEPTIONEVENT)
- J (Job and Task Structure Display)
- LIBS (Library Task Entries)
- MSG (Display Messages)
- MX (Mix Entries)
- OF (Optional File)
- OK (Reactivate)
- OT (Inspect Stack Cell)
- RM (Remove)
- SL (Support Library)
- SQ (Show Queue)
- ST (Stop)
- THAW (Thaw Frozen Library)
- TI (Times)
- Y (Status Interrogate)

MARC Control of Remote Processes

You can enter system commands in MARC and direct them to a remote host by including the *AT <hostname>* prefix. The security checking that is done is the same as that done for commands entered at the ODT. However, the usercode of the MARC session is used as the command usercode. If the MARC session has no usercode, then the system uses the host usercode.

CANDE Control of Remote Processes

You can direct system commands to a remote host from a CANDE session by prefixing the command with *?AT <hostname>*. These system commands are subject to the same restrictions as commands entered using *AT <hostname>* at an ODT. The usercode of the CANDE session, or its local alias, is inspected for SYSTEMUSER status at the remote host and the commands are handled according to the results of this test.

Visibility of Remote Processes to Remote Operators

A remote process is visible to an operator at the remote host in the same way as it would if it were a local process. However, the remote process appears to be a job, even if it is actually a task. The following is an example of the Y (Status Interrogate) system command display for a remote task:

```
Status of Job 2022/2022 at 18:20:31
Program name:*UNIDATS/MCS ON DISK
Priority: 50
Origination: Unit 0
Stack State: Waiting on an event
Display: DISPLAY:WARNING: PREVIOUS ALIAS REPLACED: SPARE/UNIDATS/1.
```

The "ORIGINATION" displayed is always "UNIT 0" if the process was initiated from a remote host. (However, other circumstances can also cause an origination of "UNIT 0" to be displayed.)

The following is an example of how such a task might appear in the J (Job and Task Structure) system command display. No job is displayed for the task.

```
1450  50..(CYNTHIA) (CYNTHIA)OBJECT/ALGOL/TASK ON SYS37
```

TASKING/MESSAGE/HANDLER and TASKING/STATE/CONTROLLER

The networking software creates two special processes that handle initiation of remote processes and communication between the remote processes and their local parents. These processes are tasks initiated by the DSSSUPPORT library on the local host and the remote host. Their names are TASKING/MESSAGE/HANDLER and TASKING/STATE/CONTROLLER. One instance of each of these tasks appears in the mix at a host as long as any remote processes or parents of remote processes are running at the host. These tasks also continue to appear in the mix for a few minutes after all remote processes have terminated.

Using Host Services-Supported Task Attributes

Host Services supports task attributes in four ways.

- Supported attributes set in the task variable before the remote task is initiated are passed to the remote task for its initiation.
- Supported attributes set in the initiating program while the remote program is running are copied to the remote task immediately.
- Supported attributes that are inquired on in the initiating program while the remote program is running are fetched from the remote task immediately.
- When the remote task terminates, the following attributes are copied back to the task variable in the local program:

ACCUMIOTIME	ACCUMPROCTIME	CHARGE	ELAPSEDTIME
FAMILY	HISTORYCAUSE	HISTORYREASON	HISTORYTYPE
MIXNUMBER	OPTION	PDUMPTITLE	STACKHISTORY
STATION	STATUS	STOPPOINT	SW1
SW2	SW3	SW4	SW5
SW6	SW7	SW8	TASKVALUE

Supported task attributes are only copied back to the initiating program either when they are interrogated by the initiating program or when they are in the preceding list and the remote program terminates.

Host Services supports the majority of the task attributes discussed in this guide. However, some task attributes cannot be used across multihost networks. If Host Services does not support a task attribute, then that attribute cannot be accessed by a process running on a different host system. For example, if a parent process is running on one system and its task is running on another system, the parent cannot access some of the task attributes of the task.

To determine whether a particular task attribute is supported by Host Services, refer to the description of that attribute in the *Task Attributes Programming Reference Manual*.

Generally, if Host Services does not support a task attribute, then any attempt to access the attribute through Host Services is ignored. No error results, but the task attribute remains unchanged. A warning message is displayed if the DSSSUPPORT library on the system was compiled with the DIAGNOSTICS option set. However, if a process attempts to access the ACCEPTEVENT, EXCEPTIONEVENT, or TASKFILE task attribute of another process across a multihost network, the accessing process is discontinued.

Hosts running different software release levels can be linked in the same network. When accessing the task attributes of a remote task, the remote host may be running an old version of Host Services that does not support all the task attributes that the current version of Host Services does.

Section 13 Understanding Interprocess Communication

Interprocess communication, or IPC, is a voluntary exchange of information between two or more processes. Interprocess communication is sometimes referred to as *interprogram communication*. The latter term is avoided in this guide for two reasons.

First, a program is an artifact stored in a file that doesn't do anything. When the program is initiated, a process is created, and the process can communicate with other processes.

Second, the processes involved in interprocess communication are not necessarily instances of different programs. They might be two different instances of the same program, or they might be internal processes created by initiating procedures within the same program.

If the distinction between programs and processes seems unclear to you, read Section 1, "Understanding Basic Tasking Concepts," before proceeding any further in this section.

Information in a computer system is always stored in a particular form. For example, to store information about whether a given condition is true or false, a process might declare a Boolean variable. To store numeric data, the process might declare an integer variable or real variable. To record a set of instructions that can be invoked repeatedly, the process might declare a procedure. All of the things that can be declared in processes can be thought of, in a general way, as "objects."

With this point in mind, you can see that IPC consists of processes making use of objects declared by other processes. For example, if one process assigns a value of 3 to an integer variable declared in another process, this assignment is an example of interprocess communication. If a process invokes a procedure declared by another process, this procedure invocation is another example of interprocess communication.

Why should two processes need to have access to the same objects? The following are some examples:

- In an electronic mail system. One way for such a system to work would be for each user to initiate his or her own mail process. The mail processes could then use IPC techniques to send messages back and forth.
- For transaction processing. For example, you might have a file that is updated by many different online users. You can use IPC techniques to ensure that different users' updates do not overwrite each other.

- To promote reuse of code. You might write a procedure that is useful in many different applications. You can place the procedure in a library where it can be used by many different applications.

The system provides a variety of IPC techniques. This section introduces

- The methods available for sharing various types of objects
- The means of synchronizing access to the shared object

Methods of Sharing Objects

The system provides several methods of sharing objects among processes. Your choice of which method to use is affected by several factors that are listed in Table 13–1, "Methods of Sharing Objects." The following are the meanings of the various columns in this table:

- IPC Method. The name of the method used for sharing objects among processes.
- Language Support. The programming languages that support this IPC method.
- Objects Shared. The types of objects that can be shared by using this method.
- Between Process Families. Indicates whether this method permits communication between processes in different process families.
- Multihost Networks. Indicates whether this method permits communication among processes running on different hosts. If such communication is permitted, this column also indicates whether the hosts must all be A Series hosts, or whether they can be from a variety of vendors.

Table 13–1. Methods of Sharing Objects

IPC Method	Language Support	Objects Shared	Between Process Families	Multihost Networks	More Info
Task Attributes	WFL ALGOL COBOL NEWP	Boolean Event Integer Real String	No	No	Section 14
Global Objects	ALGOL NEWP WFL	Any	No	No	Section 15

Table 13–1. Methods of Sharing Objects

IPC Method	Language Support	Objects Shared	Between Process Families	Multihost Networks	More Info
Tasking Parameters (by reference or by name)	ALGOL C COBOL NEWP Pascal WFL	Most types of objects	No	No	Section 17
Tasking Parameters (by value)	ALGOL C COBOL NEWP Pascal WFL	Most types of objects	Yes	Yes	Section 17
Server Libraries	ALGOL C COBOL FORTRAN7 7 NEWP Pascal	Procedures	Yes	No	Section 18
Connection Libraries	ALGOL NEWP	Procedures	Yes	No	Section 18
Port Files	All but WFL	Multiple text records	Yes	Yes; A Series only	Section 19
HC Files	ALGOL NEWP	Multiple text records	Yes	Yes; A Series only	Section 19
HY Files	ALGOL NEWP	Multiple text records	Yes	Yes; multi-vendor	Section 19
Shared Logical Files	All	Multiple text records	Yes	No	Section 19

Table 13–1. Methods of Sharing Objects

IPC Method	Language Support	Objects Shared	Between Process Families	Multihost Networks	More Info
Disk Files with Buffer Sharing and Record Locking	COBOL85 native; library support for others	Multiple text records	Yes	No	Section 19
CRCR	COBOL85 native; library support for others	Single text string	Yes	No	Section 20
STOQ	COBOL85 native; library support for others	Multiple text strings	Yes	No	Section 20
ONC+ RPC	C	Procedures	Yes	Yes; multi-vendor	Section 24
Shared Memory	C native; library support for others	Boolean Character Double Integer Real	Yes	No	*POSIX User's Guide*

Methods of Synchronizing Access

When two or more processes are able to update the value of a common data item, the possibility arises that the updates can interfere with and overwrite each other. An example is that of a variable that records the current balance of a customer account. Suppose the account has a current balance of $100. One process might have responsibility for subtracting $10 from the account. Another process, running simultaneously, might have responsibility for adding $15 to the account. The net result should be a balance of $105. However, the actual results can be quite different.

The problem arises because this type of update involves building on the value that is already present. If more than one process updates the account, a sequence like the following can occur:

1. Process A reads the account balance ($100) into variable A1.
2. Process B reads the account balance ($100) into variable B1.
3. Process A subtracts $10 from variable A1, leaving $90.
4. Process B adds $15 to variable B1, leaving $115.
5. Process A assigns the value from A1 to the account balance, leaving a balance of $90.
6. Process B assigns the value of B1 to the account balance, leaving a balance of $115.

In other words, process B can unintentionally delete the effect of the update performed by process A. The result is that the customer balance is left at $115 instead of the correct $105.

To prevent such situations from occurring, it is sometimes necessary that a process be able to secure exclusive access to an object for the duration of the transaction. The system provides a special type of variable called an *event* for handling these and other types of timing problems. You must use some means, such as global declarations, parameters, or SHAREDBYALL libraries, to provide the communicating processes with access to the event. You can then design the processes to use the event as a sort of flag to signal the availability of another object, such as a variable or file.

Events can be declared and used in ALGOL and COBOL. Certain implicitly declared events can also be accessed by WFL. For further information about events, refer to Section 16, “Using Events and Interlocks.”

In addition to events, the system supports a synchronizing mechanism known as *interlocks*. You can use interlocks for some of the same purposes as events, but interlocks have the advantage of being executed more quickly. For further information about interlocks, refer to “Using Interlocks” in Section 16, “Using Events and Interlocks.”

The system also supports two synchronization mechanisms based on the POSIX standards: signals and semaphores. These mechanisms serve a purpose similar to events, and are primarily useful for POSIX applications that are ported to the A Series. For further information, refer to the *POSIX User's Guide*.

Section 14 **Using Task Attributes**

Certain task attributes exist only for the purpose of transmitting information between different members of a process family. These attributes have no meaning to the system, and thus can be used only for storing values to be read later. The following are the task attributes that fall into this category:

- AX

  This attribute stores a string that the process can read with an ACCEPT statement.

- LOCKED

  This Boolean-valued task attribute accesses the availability state of an event. For further information, refer to "Using Implicitly Declared Events" in Section 16, "Using Events and Interlocks." For information about using this attribute to control a task from a job, refer to "Controlling a Task from a Job" in Section 4, "Tasking from Programming Languages."

- SW1 through SW8

  Each of these attributes stores a Boolean value.

- TARGET

  This attribute stores an integer value.

- TASKSTRING

  This attribute stores a string value.

- TASKVALUE

  This attribute stores a real value.

In a more general way, all task attributes are instruments for interprocess communication (IPC). After all, each task attribute stores information about the process it applies to, and this information is visible to any other process that can access the task variable. What distinguishes the task attributes in the preceding list is that they have no meaning at all, except what is established by convention between two processes.

These task attributes provide the simplest means of IPC. There is no need to create and define complex data structures, as all task attributes are predeclared.

Each of the attributes involved stores only a single Boolean, arithmetic, or string value, which can be changed and read repeatedly during process execution.

A disadvantage to using these task attributes is that the task attribute names are fixed and thus do not convey any information about what is being stored in the attribute. Someone reading the program might have trouble understanding why the attribute is being used. By contrast, a variable can always be assigned a meaningful name.

Another disadvantage is that it generally takes more processor time to read or write a task attribute than to read or write variables declared by the process.

For two processes to communicate using task attributes, one or both must have access to a common task variable. If two processes belong to the same process family, they can always communicate by way of the MYJOB task variable. If two processes have a common parent, they can communicate by way of their own EXCEPTIONTASK task attribute. For further information about the task variables a process can access, refer to Section 2, "Understanding Interprocess Relationships."

The task attribute most commonly used for IPC is TASKVALUE, and its most common use is in task equations. For example, you could use TASKVALUE to instruct a program whether to produce a printout. The program could contain the following statement:

```
IF MYSELF.TASKVALUE = 1 THEN F.KIND:= VALUE(PRINTER)
   ELSE F.KIND:= VALUE(REMOTE);
```

If TASKVALUE has a value of 1, the program produces a printout; otherwise the program displays its output at the user's terminal. You might use a statement like the following to initiate the program and cause the program to produce a printout:

```
RUN REPORT/GENERATOR;TASKVALUE = 1
```

Section 15
Using Global Objects

In Section 1, "Understanding Basic Tasking Concepts," the concept of an *internal task* was introduced. An internal task is created by a statement that initiates a single procedure within a program. The capability of initiating internal tasks exists only in WFL and ALGOL.

WFL and ALGOL share a similar type of program structure. Both languages allow you to create blocks that can include declarations of objects for use within the block. Both languages allow you to nest blocks within other blocks. Both languages allow nested blocks to use objects declared in the blocks they are nested within. These objects are referred to as *global objects*.

Globally declared objects can be used to allow an internal task to communicate with its parent or with other internal tasks of the same parent. Even widely separated members of a process family can communicate with each other by way of global objects. For example, sibling or cousin tasks could communicate, or a task could communicate with an ancestor. For an introduction to the possible relationships in a process family, refer to Section 2, "Understanding Interprocess Relationships."

Processes can communicate through a particular global object only if the processes meet both the following rules:

- Each process is one of the following: the process that executed the declaration of the global object, or an internal task of that process, or an internal task of one of these internal tasks, and so on.
- Each process must have been created by initiating a procedure that falls within the scope of the declaration of the global object.

The *scope* of a declaration consists of all the blocks that have access to the object declared. Conversely, the *addressing environment* of a block consists of all the objects that can be used by statements in the block. The following subsections discuss the scope of declarations in WFL and ALGOL, and give examples of related processes that communicate through global objects.

Global objects can also be used in SHAREDBYALL libraries to provide communication between unrelated processes. The use of global objects in libraries is discussed in Section 18, "Using Libraries."

Communication through Global Objects in WFL

The types of blocks that can occur in a WFL job are the outer block and any SUBROUTINE declarations in the job. The scope of a declaration in WFL is limited to the following blocks:

- The block in which the declaration occurs
- Any blocks that are nested in the declaration block and that occur after the declaration

The following WFL example illustrates the effects of these scope rules:

```
100 ?BEGIN JOB;
110 INTEGER OUTERINT1;
120 SUBROUTINE FIRSTSUB;
130   BEGIN
140     INTEGER FIRSTINT;
150     SUBROUTINE NESTEDSUB;
160       BEGIN
170         INTEGER NESTEDINT;
180         OUTERINT1:= 3;
190         FIRSTINT:= 3;
200         NESTEDINT:= 3;
210       END NESTEDSUB;
220     OUTERINT1:= 2;
230     FIRSTINT:= 2;
240   END FIRSTSUB;
250 INTEGER OUTERINT2;
260 OUTERINT1:= 1;
270 OUTERINT2:= 1;
280 ?END JOB
```

This example includes three procedures: the outer block of the job and two subroutines, of which NESTEDSUB is nested within FIRSTSUB. Each procedure includes integer variable declarations. Additionally, each procedure that is within the scope of an integer variable declaration includes a statement making an assignment to the integer variable.

Thus, the integer variable OUTERINT1 can be used by statements in the outer block, the FIRSTSUB subroutine, and the NESTEDSUB subroutine. This is because the scope of a declaration includes the procedure it is declared in and all nested procedures. By contrast, the integer variable OUTERINT2 cannot be used by statements in FIRSTSUB or NESTEDSUB, because these subroutines are declared prior to OUTERINT2.

The integer variable FIRSTINT can be used by statements in FIRSTSUB, because FIRSTINT is declared in FIRSTSUB; and by statements in NESTEDSUB, because it is nested in FIRSTSUB. However, FIRSTINT cannot be used by statements in the outer block, because the outer block is not nested inside FIRSTINT.

The integer variable NESTEDINT can be used only by statements in NESTEDSUB, because no other procedures are nested in NESTEDSUB.

The next example shows the use of global objects in WFL to provide an elementary type of IPC.

```
100 ?BEGIN JOB GLOBAL/DISPLAY;
110   CLASS = 0;
120 STRING MSG;
130 TASK S1, S2;
140 SUBROUTINE SUBONE;
150   BEGIN
160     WHILE S2(STATUS) ISNT SUSPENDED DO
170        WAIT(1);
180     MSG:= ACCEPT("ENTER A MESSAGE PLEASE");
190     S2(STATUS = ACTIVE);
200   END SUBONE;
210 SUBROUTINE SUBTWO;
220   BEGIN
230     MYSELF(STATUS = SUSPENDED);
240     DISPLAY(MSG);
250   END SUBTWO;
260
270 PROCESS SUBONE [S1];
280 PROCESS SUBTWO [S2];
290
300 ?END JOB
```

In this example, two subroutines, SUBONE and SUBTWO, are initiated as asynchronous tasks. Both subroutines fall within the scope of the string MSG, which is declared in the outer block. SUBONE waits for SUBTWO to become suspended. SUBTWO executes a statement that suspends itself. At this point, SUBONE resumes execution and assigns an operator ACCEPT message to the MSG string. SUBONE then changes the status of SUBTWO to ACTIVE. When SUBTWO resumes execution, it displays the value of the MSG string.

This is a simple example, but even in this example, it was necessary to take measures to regulate the timing of the asynchronous tasks. For example, the statement at line 180 should execute before the statement at line 240; otherwise, the DISPLAY statement at line 240 displays an empty value. This example uses assignments to the STATUS task attribute to suspend and restart execution of the asynchronous tasks. Other timing methods available in WFL include the LOCKED task attribute and various forms of the WAIT statement. These timing methods are discussed under “Using Implicitly Declared Events” in Section 16, “Using Events and Interlocks.”

Communication through Global Objects in ALGOL

ALGOL programs can include the following types of blocks: the outer block, PROCEDURE declarations, and simple blocks (BEGIN...END groups that include at least one declaration). The scope of most types of objects in ALGOL is subject to the scope limits previously described for WFL. That is, the scope of an object is limited to the following blocks:

- The block in which the object is declared
- Any blocks that are nested in the declaration block and that occur after the declaration of the object

However, ALGOL provides special constructs that can be used to increase the scope of procedure, interrupt, and switch label declarations. These constructs, which are called *forward reference declarations*, allow two objects to contain references to each other, without violating the rule that objects must be declared before they can be used. The following example illustrates this and other aspects of ALGOL scope rules.

```
100 BEGIN
110 INTEGER OUTER_INT1;
120
130 PROCEDURE SECONDPROC; % Forward reference declaration
140   FORWARD;
150
160 PROCEDURE FIRSTPROC;
170   BEGIN
180     INTEGER FIRST_INT;
190     OUTER_INT1:= * + 1;
200     FIRST_INT:= * + 1;
210     % Begin simple block
220     BEGIN
230       INTEGER SIMPLE_INT;
240       SIMPLE_INT:= OUTER_INT1 + FIRST_INT;
250     END;
260     % End simple block
270     IF OUTER_INT1 + FIRST_INT < 100 THEN
280        SECONDPROC;
290   END FIRSTPROC;
300
310 PROCEDURE SECONDPROC;
320   BEGIN
330     FIRSTPROC;
340   END;
350
360 INTEGER OUTER_INT2;
370 % Beginning of outer block statements
380 OUTER_INT1:= 1;
390 OUTER_INT2:= 1;
400 SECONDPROC;
410 END.
```

Two integers are declared in the outer block of this example: OUTER_INT1 and OUTER_INT2. Of these, OUTER_INT1 is visible to the outer block, FIRSTPROC, SECONDPROC, and the simple block inside FIRSTPROC. However, OUTER_INT2 is visible only to the outer block, because it follows all the nested procedure declarations.

Procedure FIRSTPROC is visible to SECONDPROC, because both are declared in the outer block, and the declaration of FIRSTPROC precedes that of SECONDPROC. Procedure SECONDPROC is visible to FIRSTPROC because of the forward reference declaration that precedes FIRSTPROC.

Integer SIMPLE_INT is visible only in the simple block in which it is declared, because there are no other blocks nested within the simple block.

The next example shows the use of global objects in ALGOL to provide an elementary type of IPC.

```
100 BEGIN
110 EBCDIC ARRAY MSG[0:71];
120 EVENT E;
130 TASK S1, S2;
140 PROCEDURE SUBONE;
150   BEGIN
160     REPLACE MSG BY "ENTER A MESSAGE PLEASE";
170     ACCEPT(MSG);
180     CAUSE(E);
190   END SUBONE;
200 PROCEDURE SUBTWO;
210   BEGIN
220     WAIT(E);
230     DISPLAY(MSG);
240   END SUBTWO;
250
260 PROCESS SUBONE [S1];
270 PROCESS SUBTWO [S2];
280
290 WHILE S1.STATUS GTR VALUE(TERMINATED) OR
300       S2.STATUS GTR VALUE(TERMINATED) DO
310    WAITANDRESET(MYSELF.EXCEPTIONEVENT);
320 END.
```

In this example, the two procedures SUBONE and SUBTWO are able to communicate through the EBCDIC array MSG, which is declared globally to the two procedures. This example is similar to the example shown earlier at the end of the “Communication through Global Objects in WFL” subsection. The major difference is that this ALGOL example regulates the timing of the tasks with a special type of object called an *event*. Thus, the WAIT(E) statement at line 220 causes SUBTWO to wait until SUBONE executes the CAUSE(E) statement at line 180. For further information about events, refer to Section 16, “Using Events and Interlocks.”

Section 16
Using Events and Interlocks

Shared objects and task attributes provide a relatively simple means of communicating information if all the tasks involved are synchronous tasks. If the tasks in a process family are all synchronous, then only one process is executing at a time. The order in which processes access shared objects is therefore fixed.

However, in cases where asynchronous processes access the same object, the order in which they access the shared objects is not fixed. This fact can create many unexpected side effects in communication. For example, suppose two processes communicate a vital bit of information by way of a shared integer variable. How is one process to know that the other process has updated the variable, so that it is now ready to be read?

The answer is that a programmer must implement flags to indicate whether a particular variable is to be accessed at this time. You can implement many types of flags. For example, a process could reset the value of a variable to zero after reading it. Another process could be designed to write a new value to the variable whenever the variable contains a zero. In this example, the zero value is being used as a flag to show that the variable has been read and is ready to have a new value written into it.

One problem with these types of flags is that the processes involved have to keep checking the flag periodically to see if it has been set. These repeated checks waste processor time. Another problem is that, between the time that one process reads the flag and the time it sets the flag, another process might have written to or read the flag. The flag is, therefore, not completely reliable.

You can avoid both of these problems by using *events.* An event is a special type of object that is used only for regulating the timing of asynchronous processes. A process can wait for an event to assume a certain state, without using any processor time while it waits. When the event assumes the desired state, the process resumes execution automatically.

Aside from events, the system supports objects called *interlocks,* which can also be used for timing purposes. Interlocks provide faster execution than events in some situations. Interlocks also make it easy for processes to detect and recover from cases where the process owning an interlock is discontinued.

Using Events

Events can be declared in ALGOL and COBOL programs, but not in other languages. Work Flow Language (WFL) jobs can wait on certain predeclared and implicitly declared events.

Events can be made available to tasks in the same way as other objects can. That is, internal tasks can access events declared globally in their parents. An internal or external task can be passed an event as a parameter.

An event consists of an identifier that has two states associated with it: the available state and the happened state. The available state can be AVAILABLE or NOT AVAILABLE. The happened state can be HAPPENED or NOT HAPPENED. These values can be inspected or changed by any of several event-related statements that are described in the following pages.

The available state of an event is typically used to temporarily restrict access to a particular object, so that only one process can access the object during a given period of time. The happened state is used to allow one or more processes to wait without using any processor time while waiting.

The initial available state of an event is AVAILABLE. The initial happened state of an event is NOT HAPPENED.

Declaring Events

In ALGOL, an event declaration is similar to a simple variable declaration. The following statement declares two events:

```
EVENT EDATA, EACCESS;
```

Events can be grouped in ALGOL as a one-dimensional event array. The following example declares an event array:

```
EVENT ARRAY EVNT[1:12];
```

The elements of this array can be used wherever an event is allowed. For example, EVNT[3] accesses the third event in the previous array declaration.

Events can be declared in COBOL as elementary or group items. The following example declares an event as an elementary item:

```
77 E1 USAGE IS EVENT.
```

The following example declares a group item that contains two events and a two-dimensional event array:

```
01 EGROUP      USAGE IS EVENT.
   03 E-1.
   03 E-2.
   03 E-3 OCCURS 5.
      05 E-4 OCCURS 10.
```

Accessing the Available State

You can use the available state of an event to ensure that only one process has access to an object, or set of objects, at a time. This one-at-a-time access is sometimes referred to as *mutual exclusion*. You might need to ensure mutual exclusion for either of the following reasons:

- To prevent updates made by one process from overwriting updates made by other processes. In this case, you need mutual exclusion only among processes that update an object.
- To ensure that a set of related objects are updated, and consistency restored, before the next time a process reads any of those objects. This consistency can be important, for example, when table entries include cross-references to other table entries. In this case, you need mutual exclusion among all the processes that read or update the objects.

The available state of an event records whether the event is currently assigned to a process. An event can only be assigned to one process at a time. If the event is currently assigned to a process, the available state is NOT AVAILABLE. If the event is not assigned to a process, the available state is AVAILABLE. A *procure* statement is one that changes the available state from AVAILABLE to NOT AVAILABLE. A *liberate* statement is one that changes the available state to AVAILABLE.

To prevent two processes in a process family from accessing the same object at the same time, you declare an event that can be used by all the processes that access the object. The processes should be designed according to a common convention so that each attempts to procure the declared event before accessing the shared object. If the event cannot be procured immediately, the process should either wait for the event to become AVAILABLE or proceed with other business until the event becomes AVAILABLE. When a process is finished using the shared object, it should liberate the event and thus make the shared object available for use by other processes.

This mechanism of protecting a shared object depends on the cooperation of all the processes that access the object. The system is not aware of any link between the event and the object it protects.

Furthermore, procuring an event does not prevent other processes from accessing the event. It simply prevents other processes from directly procuring the event. These other processes could execute statements to liberate the event and then procure it, or execute statements that access the happened state. This fact allows considerable flexibility in the use of events.

Procuring an Event Unconditionally

An unconditional procure statement is one that stops execution of the process until the requested event becomes AVAILABLE. When the event becomes available, the unconditional procure statement immediately changes the event back to NOT AVAILABLE and allows the process to resume executing. If the requested event is already AVAILABLE, then the unconditional procure does not stop execution of the process; instead, the unconditional procure immediately changes the event to NOT AVAILABLE and allows the process to continue executing.

The following ALGOL statement unconditionally procures the event E1:

```
PROCURE (E1);
```

The following COBOL statement has the same effect:

```
LOCK (E1).
```

There is one situation that can cause an unconditional procure to continue waiting even after an event becomes AVAILABLE. For details, refer to "Partially Liberating an Event" later in this section.

Procuring an Event Conditionally

A conditional procure statement allows the process to continue execution if the requested event cannot be immediately procured. The process makes one attempt to procure the event and, if the event is AVAILABLE, changes the available state to NOT AVAILABLE. The conditional procure statement returns information that enables the process to tell whether the conditional procure action was successful.

The following ALGOL statement conditionally procures the event E1 and stores the result in the Boolean variable BOOL. If the conditional procure succeeds, BOOL receives a value of FALSE. If the conditional procure fails, BOOL receives a value of TRUE.

```
BOOL:= FIX (E1);
```

The following COBOL statement conditionally procures the event E1. The AT LOCKED clause specifies an action to be taken if the procure fails.

```
LOCK (E1) AT LOCKED GO P2.
```

Liberating an Event

A liberate statement sets the available state of the event to AVAILABLE and sets the happened state to HAPPENED. (For information about the happened state, refer to "Accessing the Happened State" later in this section.) The process then continues normally.

If another process was waiting to procure the event, that process procures the event and continues execution. The available state returns to NOT AVAILABLE. If more than one process was waiting to procure the event, then only the highest-priority process succeeds, and the other processes continue to wait until the event is liberated again. Note, however, that process priority is affected by a number of factors aside from the PRIORITY task attribute. Therefore, the programmer cannot use the PRIORITY task attribute to determine which of the contending processes will procure the event.

The following ALGOL statement liberates event E1:

```
LIBERATE (E1);
```

The following COBOL statement has the same effect:

```
UNLOCK (E1).
```

Partially Liberating an Event

A partial liberate statement sets the available state of an event to AVAILABLE, but leaves the happened state unchanged. The process performing the partial liberate statement continues execution normally. The partial liberate statement differs from a liberate statement in that it does not cause waiting processes to resume execution. Any processes that previously had executed a wait statement or an unconditional procure statement will continue to wait indefinitely. However, because the partial liberate statement changes the event to AVAILABLE, the event can be procured by procure statements executed after the partial liberate statement.

You should be very careful when using the partial liberate statement. You need to either cause or liberate the event eventually so that the processes that are waiting on the event can resume. (The cause statement is discussed under "Causing an Event" later in this section.) However, because the partial liberate statement changes the state to AVAILABLE, another process could procure the event before the first process fully liberates it. Unless you design the code carefully, two different processes might accidentally use the resource flagged by the event at the same time.

In ALGOL, the partial liberate statement is called FREE. The following is an example of this statement:

```
FREE (E1);
```

The partial liberate statement can also be used as a function that returns a Boolean value. If the event is already AVAILABLE, a value of FALSE is returned. If the event is NOT AVAILABLE, a value of TRUE is returned, and the event is set to AVAILABLE. The following ALGOL statement partially liberates event E1 and stores the result in BOOL:

```
BOOL:= FREE (E1);
```

The partial liberate statement is not available in COBOL.

Testing the Availability of an Event

An availability test returns a Boolean value that indicates whether the event is AVAILABLE. If the available state is AVAILABLE, the test returns TRUE. If the available state is NOT AVAILABLE, the test returns FALSE. The process continues normal execution in either case; it does not wait for the event to become AVAILABLE. The availability test does not make any change to the event and does not affect processes waiting on the event.

The following ALGOL statement tests the available state of the event E1:

```
WHILE AVAILABLE (E1) DO...
```

The availability test is not available in COBOL.

Note that the availability test is not an adequate substitute for the conditional procure statement. Thus, the effects of the following two statements are quite different:

```
FIX (E1);
IF AVAILABLE (E1) THEN PROCURE (E1);
```

Suppose these statements are executed by a process called *A*. The first statement, FIX, causes a conditional procure. This statement procures event E1 if it is AVAILABLE, but abandons the procure and allows process A to continue running if E1 is NOT AVAILABLE. The second statement attempts an unconditional procure if E1 is AVAILABLE. However, there might be another process, hereafter referred to as *B*. Process B might procure E1 after process A executes the availability test, but before process A executes the unconditional procure. In that case, process A would cease execution until process B eventually liberated the event.

The lesson to be learned from this example is that the availability test should be used only in cases where the process does not need to procure the event, but only needs to determine whether the event is currently in use by another process. However, even this use can cause efficiency problems if done with excessive frequency. Refer to “Buzz Loops” later in this section for details.

Determining the Ownership of an Event

A process becomes the owner of an event when the process successfully procures that event, and remains the owner until the event is liberated. A process can use the MCP procedure EVENT_STATUS to determine whether that process is the current owner of the event.

The EVENT_STATUS procedure is primarily useful in fault-handling code, EPILOG procedures, and EXCEPTION procedures. In these contexts, the EVENT_STATUS result enables the process to determine whether it should liberate the event before exiting a procedure, to make the event available to other processes. For further information about EPILOG and EXCEPTION procedures, refer to “Using EPILOG and EXCEPTION Procedures” later in this section.

Note: *The EVENT_STATUS procedure is the only safe method of determining the owner of an event. Unsafe NEWP programs that manipulate events directly should be avoided, because the format of events differs among system models and is subject to change without notice. Use of the EVENT_STATUS procedure makes it unnecessary to modify application programs when the event format changes.*

The following declarations can be included in an ALGOL program to enable the EVENT_STATUS procedure to be used:

```
LIBRARY MCPSUPPORT (LIBACCESS = BYFUNCTION);

REAL PROCEDURE EVENT_STATUS(EV);
     EVENT EV;
     LIBRARY MCPSUPPORT;

DEFINE LOCKOWNERF = [18:15] #; % Lock owner field in EVENT_STATUS result
```

The program passes the event in question to the EV parameter of the EVENT_STATUS procedure. This procedure returns information about the event in the procedure result. The format of this result is as follows:

Field	**Meaning**
[18:15]	Stack number of the process that owns this event
[3:2]	Event usage
	0 = Normal event
	2 = Interrupt attached
[1:1]	Availability state
	0 = AVAILABLE
	1 = NOT AVAILABLE
[00:01]	Happened state
	0 = NOT HAPPENED
	1 = HAPPENED

Note: *The information in the EVENT_STATUS result reflects the state of the event at a single moment in time. If other processes have access to the event, ownership of the event could change between the time a process calls EVENT_STATUS and the time the process reads the result.*

Assuming that EVENT_STATUS, the MCPSUPPORT library, and the LOCKOWNERF field have been declared as shown previously, the following ALGOL statement can be used to determine whether the current process owns an event:

```
IF PROCESSID = EVENT_STATUS(EVENT1).LOCKOWNERF THEN
BEGIN
   %  Take various appropriate actions
END;
```

Accessing the Happened State

A process can use the happened state of an event to inform another process that some expected condition has been fulfilled. A statement that sets the happened state to HAPPENED is said to *cause* the event. A statement that sets the happened state to NOT HAPPENED is said to *reset* the event. Every process that has visibility to the event also has the right to cause or reset the event.

A process can also *wait on* an event, in which case execution of the process is suspended until another process causes the event. A process that is waiting on an event does not use any processor time. The waiting process cannot resume execution until the event is caused by some other process. Any number of processes can wait on the same event.

Processes that wait on the happened state and processes that wait on an unconditional procure statement are in a similar situation. In both cases, the process can take no further action until another process modifies the event. However, the following differences might make it more convenient to use the happened state in some cases and the availability state in others:

- Causing the happened state reactivates all the processes that are waiting on the happened state. However, liberating the available state reactivates, at most, one process. Other processes attempting unconditional procures will continue to wait.
- A single process can wait on the happened state of more than one event simultaneously. If any of the events are caused, the process resumes execution. By contrast, a single process can attempt to procure only one event at a time.
- The functions that wait on the happened state and functions that reset the happened state can be used separately or together. By contrast, a function that waits on the availability state of an event always resets the availability state at the same time that it reactivates the process.

Causing an Event

The cause statement sets the happened state to HAPPENED and reactivates all processes that are waiting on the happened state of an event. Causing an event has no effect on the available state. The process that performs the cause continues without interruption.

Reactivating a process simply makes that process eligible for processor time. The priority of the process, compared to other processes in the mix, determines how soon the process resumes execution.

The happened state can be reset as soon as it is caused, if another process is waiting on the event with a wait and reset statement. Refer to “Waiting On and Resetting an Event” later in this section.

An ALGOL statement can cause only one event at a time. The following ALGOL statement causes the event EVNT:

```
CAUSE (EVNT);
```

A single COBOL statement can cause one or more events, as in the following example:

```
CAUSE EVNT1, EVNT2, EVNT3.
```

Implicitly Causing an Event

A secondary effect of liberating an event is that the happened state is set to HAPPENED, and processes waiting on the happened state are reactivated. For more information about the liberate statement, refer to “Liberating an Event” earlier in this section.

Causing and Resetting an Event

The cause and reset statement reactivates waiting processes and then returns the event to NOT HAPPENED. If the event is already HAPPENED when this function is applied, the effect is to reset the event to NOT HAPPENED.

The following ALGOL statement causes and resets the event EVNT1:

```
CAUSEANDRESET (EVNT1);
```

The following COBOL statement causes and resets three events:

```
CAUSE AND RESET EVNT1, EVNT2, EVNT3.
```

Partially Causing an Event

The partial cause statement sets the happened state of an event without reactivating any processes that are waiting on the event. The waiting processes cannot reactivate until a later statement causes the event. In addition, any new processes that attempt to wait on the event will immediately continue because the event is already HAPPENED.

The following ALGOL statement partially causes the event EVNT:

```
SET (EVNT);
```

The partial cause statement is not available in COBOL.

Resetting an Event

The reset statement changes the happened state of an event to NOT HAPPENED. This statement makes it possible to reuse an event after it has been caused. If the event is not reset after it is caused, then any processes that try to wait on the event will continue immediately instead of waiting.

The following ALGOL statement resets the event EDATA:

```
RESET (EDATA);
```

The following COBOL statement resets two events:

```
RESET EDATA, ECONTROL.
```

You can also reset the happened state with the statements discussed under “Causing and Resetting an Event” earlier in this section or “Waiting on and Resetting an Event” later in this section.

Waiting on an Event

The wait statement suppresses execution of the process until another process causes the event. If the happened state is already HAPPENED, then the wait statement has no effect and the process proceeds immediately. The wait statement does not change the happened or available states of the event.

The waiting process does not use any processor time. Nevertheless, the waiting process is considered active, rather than suspended, and does not appear in the W (Waiting Entries) system command display.

A waiting process is discontinued if it exceeds the time limit specified by the WAITLIMIT task attribute.

The following is an ALGOL statement that waits on the event EVNT1.

```
WAIT (EVNT1);
```

In COBOL, the statement appears as follows:

```
WAIT UNTIL EVNT1.
```

Waiting on Time

A wait statement can also cause the process to wait for a specified number of seconds. The wait statement implicitly causes the system to create an event. The system causes the event after the specified time period. The actual time can be somewhat longer than the requested time, depending on the priority of the process and how busy the processor is.

The maximum time delay that a process can request is 164926 seconds (about 46 hours). If a wait statement specifies a longer period of time, the system reduces it to this maximum value.

The following ALGOL statement waits for 123 seconds:

```
WAIT ((123));
```

The WFL syntax is the same, except that only one set of parentheses is used.

In COBOL, the statement appears as follows:

```
WAIT UNTIL 123.
```

Waiting on and Resetting an Event

The wait and reset statement has the same effect as the wait statement, except that the happened state of the event is reset to NOT HAPPENED after the process reactivates.

The following ALGOL statement waits and resets the event EWAIT:

```
WAITANDRESET (EWAIT);
```

The following COBOL statement has the same effect:

```
WAIT AND RESET UNTIL EWAIT.
```

Waiting on Multiple Events

The wait statement can specify a list of events. The process waits until any one of the events is caused. If any one of the events is already HAPPENED when the wait statement is executed, the process does not wait at all. The wait statement can return a value that specifies which one of the events was caused. If more than one of the events was caused, the value returned indicates the leftmost of the caused events in the list.

A wait and reset statement can also wait on multiple events. This statement resets to NOT HAPPENED the single event that reactivates the process.

ALGOL and COBOL85 programs can optionally specify which of the events in a WAIT statement should be tested first. COBOL74 does not offer this feature. This feature can help prevent starvation problems of the type described under “Preventing Starvation Problems” later in this section.

The following ALGOL statement waits for 10 seconds, or until event E1 or E2 is caused, whichever comes first. The variable EVCOUNT specifies which of the three events listed should be tested first. The relative position of the event that reactivates the process is stored in T. For example, if E1 reactivates the process, T receives a value of 2.

```
T:= WAIT ((10), E1, E2) [EVCOUNT];
```

The following COBOL85 statement has the same effect:

```
WAIT UNTIL 10, E1, E2 USING EVCOUNT GIVING T.
```

The USING EVCOUNT clause would be omitted in COBOL74, as this language does not support this feature.

ALGOL also supports an INTERRUPTIBLE option in WAIT or WAITANDRESET statements. This option causes the statement to return if a signal is received before any of the other events are caused. For information about the use of this option, refer to the *POSIX User's Guide.*

Waiting for POSIX Signals

The POSIX environment provides a notification mechanism that is similar to events in some ways. This is the *signals* mechanism. For an overview of signals, refer to the *POSIX User's Guide*.

A POSIX application can use any of the following functions to wait for a signal to arrive: pause(), sigsuspend(), sigpause(), or sleep(). Unfortunately, these functions do not provide a method of waiting for an event and a signal at the same time. For example, if you attach an interrupt to an event, and the event is caused while the process is waiting for a signal, the interrupt is not executed until a signal is received. Even then, the signal is handled before the interrupt is executed.

If you want a process to wait for signals and other events at the same time, you must use a WAIT or WAITANDRESET statement with the INTERRUPTIBLE option. The INTERRUPTIBLE option is available only in ALGOL and NEWP.

For example, suppose a program includes the following statement:

```
RSLT := WAIT [INTERRUPTIBLE] ((NAPTIME), E1, E2);
```

This statement causes the process to wait until the first of the following events occur: a signal is received, event E1 or E2 is caused, or NAPTIME number of seconds elapses. If the WAIT statement returns because a signal is received, then RSLT stores a zero.

Testing the Happened State

The happened test inspects the happened state of an event. This test returns a value of TRUE if the event is HAPPENED and FALSE if the event is NOT HAPPENED.

Note that repeated happened tests are not the most efficient method of waiting on an event. Refer to “Efficiency Considerations” later in this section.

The following ALGOL statement invokes the procedure PFILE if the event E1 is HAPPENED:

```
IF HAPPENED (E1) THEN PFILE;
```

The following COBOL statement has the same effect:

```
IF E1 THEN GO PFILE.
```

Duration of the Happened State

You can use an event to flag either a momentary condition or an elapsed condition. A momentary condition is one that is relevant only to the particular process or processes that are already waiting for the condition. An elapsed condition is one that continues to be relevant to other processes in the future.

You can flag a momentary condition by using statements that cause the event and then immediately reset it. An event is immediately reset after being caused if either of the following conditions is true:

- The event was caused by a cause and reset statement.
- At least one of the processes waiting on the event used a wait and reset statement.

The program will be easier to understand and maintain if these methods of resetting the event are not mixed. If you use a wait and reset statement, you should use a simple cause statement. If you use a simple wait statement, you should use a cause and reset statement.

You can flag an elapsed condition by using simple cause and wait statements. After the event is caused, it remains in the HAPPENED state. When the elapsed condition ends, a reset statement returns the event to the NOT HAPPENED state.

Note that the use of separate reset statements automatically implies an elapsed condition. Even if the reset statement is the first action executed after a cause or wait statement, a significant interval of time can elapse before the reset statement is executed. Only through the use of wait and reset or cause and reset statements can you flag a truly momentary condition.

Using Implicitly Declared Events

A process can access a number of types of events that are never explicitly declared. Some of these are predeclared and always available. Others are created by the system in response to certain forms of the wait statement.

Two predeclared events that are associated with every process are the exception event and the accept event. You can access these events by using the EXCEPTIONEVENT and ACCEPTEVENT task attributes. A process can wait on, cause, or reset these events by way of their associated task attributes. The following ALGOL statement waits on the exception event of the process:

```
WAIT (MYSELF.EXCEPTIONEVENT);
```

The following COBOL statement has the same effect:

```
WAIT UNTIL EXCEPTIONEVENT OF MYSELF.
```

The following WFL statement has the same effect:

```
WAIT;
```

The accept event cannot be accessed in WFL. The ALGOL and COBOL syntax for accessing the accept event parallels that used for the exception event. In addition, COBOL allows the following special syntax for waiting on the accept event:

```
WAIT UNTIL ODT-INPUT-PRESENT.
```

You can access another predeclared event by using the LOCKED task attribute. This attribute translates Boolean assignments into procure and liberate statements. Thus, a statement that sets the LOCKED attribute of a process to TRUE has the effect of unconditionally procuring the predeclared event. Setting LOCKED to FALSE liberates the predeclared event. If LOCKED is already TRUE, then any processes that attempt to set LOCKED to TRUE are queued until another process sets LOCKED to FALSE. The main virtue of this task attribute is that it provides WFL jobs with an easy way of protecting a resource, even though WFL jobs cannot access events directly.

Certain types of objects have event-valued attributes associated with them. These objects include DCALGOL queues, Direct I/O buffers, port files, and remote files. Processes can wait on these event-valued attributes just as if they were explicitly declared events. For information about DCALGOL queues, refer to the *DCALGOL Programming Reference Manual*. For information about Direct I/O buffers, port files, and remote files, refer to the *I/O Subsystem Programming Guide*.

The WAIT statement in WFL can also include clauses that cause the job to wait until specified task attribute values or file attribute values are attained. Refer to the *Work Flow Language (WFL) Programming Reference Manual* for full details.

Using Interrupts

An interrupt is a procedure that is associated with an event. Specifying an interrupt allows a process to continue executing other statements at the same time that it waits on an event. When the event is caused, control passes directly to the interrupt procedure. When the interrupt procedure finishes, the process resumes execution where it left off.

An interrupt cannot be invoked using any of the standard procedure invocation statements. An interrupt is entered only when the associated event is caused. Causing an event invokes the interrupt even if the event is already in a HAPPENED state. Therefore, there is no effective difference between using a cause statement or a cause and reset statement to invoke the interrupt.

You can use attach and detach statements to specify with which event an interrupt is associated. Execution of the interrupt can be selectively allowed or suppressed through the use of enable and disable statements. The statements that attach or detach and enable or disable an interrupt can occur in any order, and do not affect each other. For example, detaching an interrupt does not also cause it to be disabled. The initial state of an interrupt is detached and enabled.

An interrupt might not always execute immediately when its event is caused. Any of the following three circumstances can delay execution of an interrupt:

- The interrupt is disabled.
- The process is waiting on an event. Any interrupts that are caused are queued and executed when the process resumes.
- The processor is engaged in executing a higher-priority process.

Declaring Interrupts

The purpose of an interrupt declaration is to assign an identifier to the interrupt and specify the statements that are to be executed when the associated event is caused.

An interrupt cannot be passed any parameters. Otherwise, it has the same addressing environment as a procedure would have if it were declared at the same point in the program. That is, in ALGOL the interrupt can access objects declared within the interrupt and within any procedures that are declared globally to the interrupt.

In rare instances, you might want to restart the process at a point other than the point at which the process was interrupted. You can achieve this effect in ALGOL with a bad GO TO statement (that is, a GO TO statement that transfers control to a statement outside the interrupt). However, COBOL does not allow a GO TO statement to transfer control outside of the interrupt.

You should be aware of a side effect that arises from using a bad GO TO to exit an interrupt. During execution of an interrupt, the system automatically executes a general disable on all other interrupts used by the process. A bad GO TO out of an interrupt leaves the process with all interrupts disabled. You should include a general enable statement to correct this situation. (Refer to “Using General Disable and Enable Statements” later in this section.)

The following is an ALGOL example of an interrupt declaration:

```
INTERRUPT BLOCK1;
   BEGIN
     DISPLAY("ERROR");
     DISPLAY("INTERRUPT BLOCK1 OCCURRED");
   END;
```

In COBOL, an interrupt declaration can occur only in the DECLARATIVES section of the procedure division. The following is an example of a DECLARATIVES section that includes an interrupt called INT-1:

```
DECLARATIVES.
INT SECTION.
    USE AS INTERRUPT PROCEDURE.
INT-1.
    DISPLAY "ERROR".
    DISPLAY "INTERRUPT 1 OCCURRED".
END DECLARATIVES.
```

Attaching or Detaching an Interrupt

The attach statement associates an interrupt with an event. If the interrupt is already attached to another event, it is automatically detached from the old event and then attached to the new event.

You can attach each interrupt to only one event. However, you can attach more than one interrupt to the same event. When the event is caused, the associated interrupts are queued for execution in the reverse of the order that they were attached to the event.

It is possible to attach an interrupt to an event that is declared in a different process. The interrupt executes as part of the process that declared it, even if it is associated with an event in a different process. The interrupt declaration cannot be more global than the event declaration, or an "UP LEVEL ATTACH" error results. This error occurs at compile time if the compiler detects the problem. Otherwise, it occurs at run time.

The detach statement removes the association of an interrupt with an event. If the interrupt is not currently associated with an event, the detach statement has no effect and execution continues normally.

Note that if the interrupt is disabled, queued instances of the interrupt might have accumulated. Detaching the interrupt, or attaching the interrupt to a different event, causes these queued instances to be deleted. You can prevent this problem by enabling the interrupt before detaching it from an event or attaching it to a different event.

The following are ALGOL statements that attach and detach an interrupt. The first statement attaches the interrupt INT1 to the event E1. The second statement implicitly detaches the interrupt and then attaches it to the event E2. The third statement then detaches the interrupt and leaves it detached.

```
ATTACH INT1 TO E1;
ATTACH INT1 TO E2;
DETACH INT1;
```

The following COBOL statements attach two interrupts to the same event and then detach them:

```
ATTACH INT-1 TO E1.
ATTACH INT-2 TO E1.
DETACH INT-1, INT-2.
```

Enabling or Disabling an Interrupt

There might be periods during process execution when it would be undesirable for the interrupt to occur. These are generally periods when the process is accessing objects that are also modified by the interrupt. A programmer can selectively suppress execution of interrupts by using enable and disable statements.

If an interrupt's event is caused while the interrupt is disabled, the interrupt is queued for later execution. If the event is caused more than once, then multiple instances of the interrupt are queued for execution. When a later statement enables the interrupt, the queued interrupts are executed one at a time in reverse chronological order.

All interrupts are implicitly disabled while any interrupt is executing. That is, any interrupts that are caused while an interrupt is executing are queued for later execution. When the interrupt completes, the queued interrupts are executed one at a time in reverse chronological order.

Because the queuing of interrupts creates substantial overhead for a process, you should leave the interrupt in the enabled state whenever possible.

The following are examples of ALGOL statements that enable and disable an interrupt. Each statement can specify only one interrupt:

```
ENABLE INT1;
DISABLE INT1;
```

The following are examples of COBOL statements that enable and disable multiple interrupts:

```
ALLOW INT1, INT2.
DISALLOW INT1, INT2.
```

Using General Disable and Enable Statements

You can use a general disable statement to disable all the interrupts declared by the process. Interrupts declared in other related processes, such as a parent or offspring, are not affected. While a general disable is in effect, any interrupts whose events are caused are queued for later execution.

To again enable the interrupts that were disabled by the general disable statement, use a general enable statement. For the most part, the general enable statement does not enable interrupts that were already disabled when the general disable statement was entered. However, if a statement enables a specific interrupt while a general disable statement is in effect, then the general enable statement also enables that interrupt.

The following ALGOL statements illustrate the interaction of specific and general enables and disables for three interrupts, INT1, INT2, and INT3:

```
ENABLE INT1;   % Enables INT1.
DISABLE INT2;  % Disables INT2.
DISABLE INT3;  % Disables INT3.
DISABLE;       % Disables INT1.  INT2 and INT3 remain disabled.
ENABLE INT2;   % All three events remain disabled.
ENABLE;        % Enables INT1 and INT2.  INT3 remains disabled.
```

The following are the general disable and enable statements in COBOL:

```
DISALLOW INTERRUPT.
ALLOW INTERRUPT.
```

Waiting for Interrupts

You can use a special form of the wait statement to make the process wait for interrupts. While the process is waiting for interrupts, any interrupt can execute; as soon as the interrupt completes, the process returns to its waiting state. The only way the process can continue further is if an interrupt executes a bad GO TO statement that transfers control to a different statement outside the interrupt.

Waiting for interrupts can be useful for processes, such as message control systems (MCSs), that are driven by input received over time from a variety of sources. However, waiting on multiple events might be more efficient in these cases; refer to "Efficiency Considerations" later in this section.

In ALGOL, the following wait statement causes the process to wait for interrupts:

```
WAIT;
```

The COBOL equivalent is the following statement:

```
WAIT UNTIL INTERRUPT.
```

Efficiency Considerations

The event and interrupt features provide a very efficient method of synchronizing processes, if they are used as intended. However, some misuses of these features can cause performance problems. The following subsections describe some possible problems and ways to avoid them.

Buzz Loops

Several of the event-related statements allow a process to test the state of an event without causing the process to wait. These are the happened test, the availability test, the conditional procure statement, and the partial liberate statement.

These statements are designed for occasional, rather than frequent, use because each execution of the statement uses processor time. In particular, looping continuously on these statements is a very inefficient way of making a process wait. Such a loop is called a buzz loop. The following is an ALGOL example of such a loop:

```
WHILE NOT HAPPENED (E1) DO;
```

This loop repeats the happened test over and over until the event E1 attains a state of HAPPENED. This loop causes two problems:

- It wastes processor time that could be devoted to executing other processes, including the process that will eventually cause the event.
- On a single-processor system, it could become an infinite loop. Assume that another process is supposed to cause event E1. If the looping process has higher priority, it will completely monopolize the processor. The second process never executes and thus never causes event E1.

You should replace the buzz loop with some form of the wait statement, which does not use any processor time. The ALGOL statement *WAIT (E1)* could replace the loop shown in the preceding example.

Preventing Excessive Interrupt Overhead

Use of interrupts increases the processor usage of a process. The processor overhead is small if only one interrupt is used and the interrupt is not often caused. However, the overhead is much greater when multiple interrupts are used and greater still when interrupts are queued because an interrupt was disabled.

By contrast, a wait statement does not cause any continuing drain on processor resources. A process that executes a wait statement is simply ignored until the associated event is caused.

Because of these facts, wait statements should be used in preference to interrupts where possible. This is particularly true where the process needs to wait on several events simultaneously. In these cases, a statement that waits on multiple events is more efficient than a statement that waits on multiple interrupts.

Preventing Starvation Problems

A process that waits on multiple events must be carefully designed or there is a possibility that some events might be overlooked. This possibility arises because the value returned by the wait statement always indicates the leftmost of the events in the event list that has been caused. For example, consider the following ALGOL statement:

```
ENUM:= WAIT (E1, E2, E3);
```

If E1 is caused, ENUM receives a value of 1. If E1 and E2 are caused, ENUM still receives a value of 1. Now, suppose that E1 is an event that happens very frequently. Each time the wait statement is executed and E1 has already happened, the wait statement returns 1 as a value; thus, the process might never be notified that event E2 or E3 has happened. This situation is referred to as a *starvation problem.*

Strictly speaking, a starvation problem exists only if the repeated wait statement is not fulfilling the needs of the particular application. The effect of the wait statement is to give preference to the leftmost events in the event list. But if the leftmost events occur infrequently, there will be no starvation. If you order the list so that the most important events are on the left, then the starvation condition might even be desirable.

If the events in a wait list are equally important, you might wish to consider using a start index in the wait statement. A start index indicates which of the events in a wait list should be tested first. You can design the program to change the value of the start index before each time the wait statement is executed. Start indexes are supported only in ALGOL and COBOL85. The following is an ALGOL example using start indexes:

```
EVORDER:= 1;
WHILE NOT DONE DO
BEGIN
```

```
      EVT:= WAIT (EV1, EV2, EV3, EV4) [EVORDER];
      EVORDER:= * + 1;
      IF EVORDER > 4 THEN
         EVORDER:= 1;
      .
      .
      .
      END;
```

Another way to prevent events from being overlooked is to use happened tests after each execution of the wait statement. You could apply a happened test to each event that is to the right of the event that was returned by the wait statement. The following is an ALGOL example of a procedure that uses this technique:

```
   PROCEDURE EVENTWAIT;
   BEGIN
      BOOLEAN BOOL;
      INTEGER ENUM;
      DO BEGIN
         ENUM:= WAIT (E1, E2, E3);
         CASE ENUM OF
           BEGIN
             1: BOOL:= INPUTHANDLER (TRUE, HAPPENED(E2), HAPPENED(E3));
             2: BOOL:= INPUTHANDLER (FALSE, TRUE, HAPPENED(E3));
             3: BOOL:= INPUTHANDLER (FALSE, FALSE, TRUE);
           END;
      END
      UNTIL BOOL;
   END EVENTWAIT;
```

The procedure EVENTWAIT is responsible for waiting on three events, E1, E2, and E3, which were declared globally. When at least one of these events is caused, EVENTWAIT invokes another procedure called INPUTHANDLER and passes it Boolean values indicating whether each of the three events has been caused. The ENUM value indicates the leftmost event that has happened. The happened test is used for each of the events to the right of that event. You increase efficiency by minimizing the number of happened tests.

INPUTHANDLER is expected to make whatever response is appropriate for each event. INPUTHANDLER returns a Boolean value of TRUE if there is no need to wait on any more events. INPUTHANDLER is also expected to reset the events that were caused, so that it will be meaningful to wait on them again.

Note that the INPUTHANDLER invocation is used in this example for the sake of simplicity. From an efficiency standpoint, such repeated procedure invocations are rather expensive. It would be better to include the code that handles each event in the EVENTWAIT procedure.

Discontinued Processes and Events

When a number of processes are being synchronized by using events, unexpected problems can occur if one of the processes is discontinued. A process might be discontinued by the system because of an error, or by an operator using a DS (Discontinue) system command.

If the process has procured an event, but has not yet liberated it, then the event remains procured when the process is discontinued. Any other processes attempting to unconditionally procure the event will wait indefinitely.

Similarly, if the process was supposed to execute a cause statement, but was discontinued first, then the event is never caused. Other processes waiting on the event wait indefinitely, unless the processes use a wait statement with a time limit or multiple events. Refer to "Waiting on Time" and "Waiting on Multiple Events" earlier in this section.

The programmer can ignore these problems if none of the processes using an event is ever likely to be discontinued. However, in environments such as a SHAREDBYALL library, where a large number of user processes from various sources can access the same event, the programmer might want to take special precautions. The following subsections describe methods of dealing with these problems.

An additional alternative is to replace an event with an interlock, as discussed under "Discontinued Processes and Interlocks" later in this section.

Using EPILOG and EXCEPTION Procedures

You can use EPILOG or EXCEPTION procedures to help ensure that a process liberates or causes a certain event, even if the process terminates abnormally. These procedures are available only in selected languages, including ALGOL, DCALGOL, and NEWP. For further information about EPILOG and EXCEPTION procedures, refer to "Performing Cleanup during an Abnormal Termination" in Section 10, "Determining Process History."

Using Timed Wait Statements

By including a time limit on a wait statement, you can make it possible for a process to recover if a particular important event is not caused. For example, the following statement could be used in ALGOL:

```
ENUM:= WAIT ((120),E1);
```

This statement waits for 120 seconds or until event E1 is caused, whichever comes first. For example, you might know that if E1 is not caused within 120 seconds, then something has gone wrong. The process could check the value of ENUM to determine if the wait timed out. If so, the process could check the STATUS task attribute of the process that was supposed to cause the event and find out whether that process was discontinued. (This type of checking is possible only if the process has access to the task variable of the process that was supposed to cause the event.)

Using Conditional Procure Statements

There is no direct way to set a time limit on an unconditional procure statement. One alternative is to use a conditional procure statement, such as the FIX statement in ALGOL or a LOCK statement with an AT LOCKED clause in COBOL. If the conditional procure operation fails, the process could attempt it again after a specified time period. (Note that the process should not execute conditional procure operations in rapid succession, as this causes the problem discussed under "Buzz Loops" earlier in this section.) If several conditional procure operations fail, the process could check the status of other processes that might have procured the event.

Determining Whether to Liberate an Event

If the state of an event is NOT AVAILABLE, then the process that most recently procured the event can be referred to as the *owner* of that event. A process can use the MCP procedure EVENT_STATUS to determine whether that process is the current owner of an event. The EVENT_STATUS procedure is especially useful in fault-handling code and in EPILOG and EXCEPTION procedures. Refer to "Determining the Ownership of an Event" earlier in this section.

Example of Event Usage

The following is a simplified example of an online application that has one driver process and three servers. The driver process reads input from users and passes it on to whichever server is not currently busy. The underlying assumption is that the user is capable of submitting input faster than any single server can process it; this could be the case if the server has to perform many time-consuming actions, such as disk I/Os, to process the input. However, this example concentrates on the timing and resource control aspects of this situation, and so the servers in the example do not really do any useful work.

```
100 BEGIN
110 FILE TERM(KIND=REMOTE);
120 BOOLEAN FINISHED;
130 EBCDIC ARRAY MSG[0:71];
140 EVENT INMSG_EVENT, MSG_READ;
150 INTEGER I, READNUM;
160 TASK T1, T2, T3;
170
180 PROCEDURE SERVER;
190 BEGIN
200   BOOLEAN DONE;
210   EBCDIC ARRAY MSGCOPY[0:71];
220   WHILE NOT DONE DO
230   BEGIN
240     PROCURE(INMSG_EVENT);
250     REPLACE MSGCOPY BY MSG FOR 72;
260     CAUSE(MSG_READ);
270     IF MSGCOPY = "QUIT" THEN
280        DONE:= TRUE
```

```
    ELSE BEGIN
           REPLACE MSGCOPY[66] BY MYSELF.MIXNUMBER FOR * DIGITS;
           WRITE(TERM,72,MSGCOPY);
         END;
  END;
END;

PROCURE(INMSG_EVENT);
PROCESS SERVER [T1];
PROCESS SERVER [T2];
PROCESS SERVER [T3];

OPEN(TERM);
WHILE NOT FINISHED DO
BEGIN
  WAIT(TERM.INPUTEVENT);
  READ(TERM,72,MSG);
  IF MSG = "QUIT" THEN
     BEGIN
       FINISHED:= TRUE;
       READNUM:= 3;
     END
  ELSE READNUM:= 1;
  I:= 1;
  WHILE I LEQ READNUM DO
    BEGIN
      LIBERATE(INMSG_EVENT);
      WAITANDRESET(MSG_READ);
      I:= * + 1;
    END;
END;

WHILE T1.STATUS GTR VALUE(TERMINATED) OR
      T2.STATUS GTR VALUE(TERMINATED) OR
      T3.STATUS GTR VALUE(TERMINATED) DO
      WAITANDRESET(MYSELF.EXCEPTIONEVENT);

END.
```

The communication in this example takes place between the parent process and three asynchronous tasks that are instances of procedure SERVER. The communication takes place by way of the array MSG and the events INMSG_EVENT and MSG_READ. Of these, MSG is used to convey messages from the parent process to the servers. The parent process uses INMSG_EVENT to inform the servers that there is a message waiting to be read. A server uses MSG_READ to inform the parent that it has successfully read the message, so the parent can now reuse the MSG array.

When this program is initiated, the driver process procures INMSG_EVENT and initiates three instances of the SERVER procedure. Each of these servers begins by attempting to procure INMSG_EVENT; since the driver has already procured this event, all the servers wait.

The driver process then enters the loop on lines 420-590. Within this loop, the driver waits for input from a user to appear in the remote file, and then reads the input into MSG. In most cases, the driver then liberates INMSG_EVENT and waits on the MSG_READ event. When the driver liberates INMSG_EVENT, one of the servers succeeds in procuring the event and copies the contents of MSG to the local array MSGCOPY. The server then causes MSG_READ, informing the driver that MSG is again available for use as a buffer. The server then performs some processing on the input in MSGCOPY and notifies the user of the result by writing a message to the remote file.

If the input received from the user is the command QUIT, then the driver takes some special actions. It liberates INMSG_EVENT and waits on MSG_READ three times without performing any more read operations. This allows the contents of the MSG array to be read by each of the three servers. Each server recognizes the QUIT command and terminates gracefully. Then the driver terminates as well.

Note that this program uses the available state of INMSG_EVENT, but uses the happened state of MSG_READ. This difference reflects the different purposes for which these events are used. The program alternates between two phases: a phase in which the driver uses the MSG array, and a phase in which any single one of the servers can use the MSG array. Causing MSG_READ initiates the phase in which the driver uses MSG; liberating INMSG_EVENT initiates the phase in which one of the waiting servers is allowed to use MSG.

Using Interlocks

You can use interlocks to protect a resource that is shared among several participating processes. Interlocks are used in much the same way as the availability state of an event. However, for interlocks, the LOCK and UNLOCK functions are used instead of PROCURE and LIBERATE statements. Interlocks provide the following advantages when compared to the use of events:

- Interlock functions are executed more quickly than PROCURE and LIBERATE statements.
- Unlike the PROCURE statement, the LOCK statement can include options causing the action to be abandoned if a specified event occurs or if a given time limit elapses.

Interlocks are available only in ALGOL and NEWP. The syntax for declaring and using interlocks is the same in both languages.

Interlocks can be in any of three different states: FREE, LOCKED_CONTENDED, or LOCKED_UNCONTENDED. Only one process can succeed in locking the interlock at a time. If multiple processes are waiting to lock the interlock, then when the current owner unlocks that interlock, the system allows one of the waiting processes to complete its lock; the other processes continue to wait.

Declaring Interlocks and Interlock Arrays

You can declare individual interlocks or arrays of type INTERLOCK, as shown in the following examples:

```
INTERLOCK LK1;

INTERLOCK ARRAY[0:10];
```

The initial state of an interlock is FREE.

Locking an Interlock

The LOCK function attempts to secure ownership of an interlock. If the LOCK succeeds, the following effects occur:

- The state of the interlock changes to LOCKED_CONTENDED or LOCKED_UNCONTENDED (depending on whether other processes are also waiting to secure this lock).
- The system updates the interlock to record the stack number of the process that now owns the interlock.

The LOCK function can specify any of several actions to take if the requested interlock is not immediately available. The following are examples:

RSLT := LOCK (LK1);

Wait indefinitely.

RSLT := LOCK (LK1, 60);

Wait with 60-second time limit.

RSLT := LOCK (LK1, 0);

Abandon lock attempt if interlock is unavailable.

RSLT := LOCK(LK1, EVENT2);

Wait with option of being interrupted by event EVENT2.

RSLT := LOCK [INTERRUPTIBLE] (LK1);

Wait with option of being interrupted by receipt of a signal. Signals are a feature of the POSIX environment. For an overview of signals, refer to the *POSIX User's Guide*.

RSLT := LOCK [DSABLE] (LK1);

Wait with the following conditions:

- If the process has already been discontinued, the wait is abandoned. This situation can occur if the LOCK function is executed in an EPILOG or EXCEPTION procedure.
- Once the process is waiting, if a DS (Discontinue) system command is received, the wait is abandoned.

The DSABLE option is implemented in DMALGOL and NEWP because programs in these languages are capable of entering a state in which they would otherwise be immune to DS commands. The DSABLE option is not available in ALGOL because ALGOL interlocks can be discontinued anyway.

If the DSABLE option is used, then the INTERRUPTIBLE option cannot be used. However, the INTERRUPTIBLE option includes the effects of the DSABLE option.

The LOCK function is of type INTEGER and returns one of the following values:

Value	Meaning
0	A signal was received or a DS command was received before the interlock could be acquired.
1	The interlock was successfully acquired.
2	The timeout elapsed or the event HAPPENED before the interlock could be acquired.

LOCK can be used as a statement rather than a function. However, if the LOCK statement does not finish successfully, the system discontinues the process executing the LOCK statement.

Unlocking Interlocks

The UNLOCK function gives up ownership of an interlock. If the UNLOCK succeeds, the following effects occur:

- If no process is waiting to acquire the interlock, then the state of the interlock changes to FREE and the system changes the stack number stored in the interlock to 0.
- If one process is waiting to acquire the interlock, then the state of the interlock changes to LOCKED_UNCONTENDED, and the system changes the stack number in the interlock to record the new owner.

- If multiple processes are waiting to acquire the interlock, then the state of the interlock is LOCKED_CONTENDED. The highest-priority process acquires the interlock. Note, however, that process priority is affected by a number of factors aside from the PRIORITY task attribute. Therefore, the programmer cannot use the PRIORITY task attribute to determine which of the contending processes will acquire the interlock.

  Additionally, the system changes the stack number in the interlock to record the new owner. The state of the interlock remains LOCKED_CONTENDED.

The UNLOCK function is typically used by the current owner of the interlock. However, it is also possible for a process to UNLOCK an interlock owned by another process.

The UNLOCK function is of type INTEGER and returns one of the following values:

Value	**Meaning**
1	The interlock was successfully unlocked.
2	The interlock has a state of FREE and therefore cannot be unlocked.

UNLOCK can be used as a statement rather than a function. However, if the UNLOCK statement does not finish successfully, the system discontinues the process executing the UNLOCK statement.

Following are examples of UNLOCK functions:

```
I := UNLOCK (MYLOCK)

I := UNLOCK (YOURLOCKS [2])
```

Interrogating the Interlock Status

The LOCKSTATUS function returns the state and ownership of the specified interlock. The LOCKSTATUS function is of type REAL, and the result has the following subfields:

Field	**Value and Meaning**	
[47:24]	Stack number of the owner process. If the current state is FREE, this field stores a 0.	
[23:22]	Undefined	
[01:02]	Current state:	
	0	FREE
	1	LOCKED_UNCONTENDED
	2	LOCKED_CONTENDED

Following are examples of the LOCKSTATUS function:

```
R := LOCKSTATUS (MYLOCK);

OWNERID := LOCKSTATUS (YOURLOCKS [3]).[47:24];

IF I := LOCKSTATUS (YOURLOCKS [3]).[47:24] = PROCESSID THEN
          GOFORIT;
```

Discontinued Processes and Interlocks

When multiple processes have access to the same interlock, you need to plan for the possibility that a process might be discontinued while it has ownership of the interlock. This is the same problem that occurs for shared events, as previously discussed under “Discontinued Processes and Events” in this section. Interlocks incorporate the following features that help you to deal with this problem:

- The time limit and event options in the LOCK function.
- The stack number value that is returned by the LOCKSTATUS function.

For example, if you believe that the process is normally unlikely to have to wait more than one minute, you could include a 60-second time limit in the LOCK function. If the time limit expires, the process could use LOCKSTATUS to determine the stack number of the current owner of the interlock. The process could then use GETSTATUS call type 6 (Mix Entries), Subtype 4, to determine whether the owner process is still running. If the owner process has terminated, this process could use an UNLOCK statement to make the interlock available for locking again.

Replacing Readlocks with Interlocks

In the past, many programmers have avoided the processor overhead of PROCURE and LIBERATE statements through techniques that make use of the READLOCK intrinsic instead. The READLOCK intrinsic is available in ALGOL and NEWP. The use of READLOCK should be eliminated. Other alternatives are available. A swap statement replaces the use of the READLOCK intrinsic for swapping values where locking is not required. If locking is required, the READLOCK intrinsic is replaced by an interlock function. The interlock functions are a suitable replacement with low processor overhead. The following example shows how interlocks can replace readlocks in one locking situation.

Using readlocks:

```
REAL
    THELOCK,
    THEOWNER;

EVENT
    THEEVENT;

DEFINE
```

```
        ACQUIRETHELOCK =
        BEGIN
            IF READLOCK(PROCESSID,THELOCK) NEQ 0 THEN
                DO
                    PROCURE(THEEVENT) % or WAITANDRESET(THEEVENT)
                UNTIL READLOCK(-1,THELOCK) EQL 0;
            THEOWNER := PROCESSID;
        END#,

        RELINQUISHTHELOCK =
        BEGIN
            THEOWNER := 0;
            IF READLOCK(0,THELOCK) NEQ PROCESSID THEN
                LIBERATE(THEEVENT); % or CAUSE(THEEVENT);
        END#;
```

Using interlocks:

```
  INTERLOCK
     THELOCK;

  DEFINE
     ACQUIRETHELOCK = LOCK(THELOCK)#,

     RELINQUISHTHELOCK = UNLOCK(THELOCK)#;
```

Section 17
Using Parameters

A *parameter* is an object passed to a procedure by the procedure invocation statement. Note that the term "procedure" is used here, as it is throughout this guide, to refer to complete programs as well as to subroutines within a program. Most programming languages can pass parameters to procedures. Parameters can be of many types, and in each language, most or all of the types of available variables can be passed as parameters.

Each parameter has two aspects: an *actual parameter* and a *formal parameter*. The actual parameter is the parameter specified in the procedure invocation statement. The formal parameter is the parameter as it is declared in the procedure that is being invoked.

Parameters that are used in a process initiation statement provide an avenue of communication between the initiating process and the new process. Such parameters are referred to hereafter as *tasking parameters*.

Parameters that are used in a library procedure invocation statement provide another type of interprocess communication. Such parameters are hereafter referred to as *library parameters*.

The "Determining the Scope of Parameters" and "Parameter Passing Modes" subsections of this section provide information that is relevant to both tasking parameters and library parameters. The remainder of this section addresses only tasking parameters. For further information about library parameters, refer to Section 18, "Using Libraries."

Determining the Scope of Parameters

Section 15, "Using Global Objects," defined the *scope* of a declaration as all the blocks in a program with access to an object declared in the program. That section explained how the scope of a declaration extends through nested blocks in a program.

You can use parameters to pass an object to a procedure that does not fall within the scope of the declaration of that object. This fact makes parameters a more general tool for IPC than global objects. A parameter can increase the scope of an object in the following ways:

- An object can be passed as a parameter to a procedure that is not nested within the block that declares the object.
- A parameter can be passed to an external procedure, whether the procedure is a passed external procedure, a library procedure, or a separate program.

The objects a procedure can access, because the objects are declared in the procedure or are declared globally to the procedure, are referred to as the *direct addressing environment* of the procedure. The objects in the direct addressing environment, together with any objects passed as parameters to the procedure, comprise the *extended addressing environment* of the procedure.

If a procedure is passed as a parameter to another procedure, the invoked procedure gains access to the passed procedure. However, the scope is extended only one way. The passed procedure does not automatically gain access to objects declared in the invoked procedure.

The following ALGOL example includes two cases where the scope of a declaration has been increased by the effects of parameter passing:

Example

```
100 BEGIN
110
120 PROCEDURE P(Q);
130   PROCEDURE Q (R);
140     REAL R;
150     FORMAL;
160 BEGIN
170   REAL A;
180   A:= 2;
190   Q(A);
200   DISPLAY (STRING(A,*));
210 END;
220
230 PROCEDURE Y;
240 BEGIN
250   PROCEDURE X(Z);
260   REAL Z;
270   BEGIN
280     Z:= Z * 2;
290   END;
300
310   P(X);
320 END;
330
340 Y;
350
360 END.
```

Case 1

Procedure X cannot be directly invoked by a statement in procedure P, because the declaration of procedure X occurs within procedure Y. However, the statement at line 310 that invokes procedure P passes procedure X as an actual parameter to the formal parameter Q. Thus, the statement at line 190, which invokes the formal parameter Q, actually results in an invocation of procedure X. In this way, a statement in procedure P is able to invoke a procedure outside the direct addressing environment of procedure P.

Case 2

Real variable A cannot be directly accessed by a statement in procedure X because A is declared within procedure P. However, the statement at line 310 passes procedure X as an actual parameter to formal parameter Q of procedure P. The statement in procedure P at line 190 then passes A as a parameter to procedure Q, thus making it possible for procedure X to access A.

Even after being passed to P, X does not automatically have access to objects declared in P. Thus, X could not have accessed A if A had not been passed as a parameter to X.

Parameter Passing Modes

There are several different *passing modes* that govern the relationship between the actual parameter and the formal parameter. The passing mode determines, for example, whether assignments made to the formal parameter are reflected by the actual parameter. The passing mode can also make a large difference in program performance in cases where the actual parameter is an expression. The three types of passing modes available are call-by-value, call-by-name, and call-by-reference.

The following subsections describe the three types of passing modes and explain how you can specify which passing mode is used.

Call-by-Value Parameters

If a parameter is passed by value, the system evaluates the actual parameter when the procedure is invoked and assigns the value to the formal parameter. Changes made to the value of the formal parameter do not affect the value of the actual parameter. Similarly, any changes made to the value of the actual parameter after procedure invocation do not affect the value of the formal parameter.

An advantage to using call-by-value parameters is that they never result in the accidental creation of a *thunk*. (Thunks are defined in the discussion of call-by-name parameters that follows.) Another advantage is that they simplify program structure. Because the actual parameter and the formal parameter do not affect each other, new values can be assigned to either without creating unexpected side effects.

Call-by-Name Parameters

When a parameter is passed by name, the system never creates the formal parameter. Instead, the system substitutes the actual parameter for the formal parameter wherever the formal parameter is mentioned in the procedure.

The effect of passing by name is simplest in cases where the actual parameter is a simple variable. When the procedure accesses the formal parameter, the effect is as if the procedure were using a global variable. Any changes made to the value of the formal parameter immediately affect the value of the actual parameter and vice versa. This feature makes call-by-name parameters a useful means of communicating information between an asynchronous process and its initiator.

When an actual parameter that is a constant or an expression is passed by name, the compiler generates a *thunk.* A thunk (also known as an *accidental entry*) is a piece of code that evaluates the actual parameter and assigns the resulting value to the formal parameter. The system substitutes the thunk for the formal parameter wherever the formal parameter is mentioned in the procedure.

Thunks can be undesirable because they slow execution of the program and affect the definition of the critical block. (Critical blocks are discussed in Section 2, "Understanding Interprocess Relationships.") The programmer can prevent the creation of a thunk by passing each element of the expression as a separate parameter.

If a constant is passed by name, then whenever the value of the formal parameter is read, the formal parameter returns the value of the constant. The value of the formal parameter cannot change. An attempt to assign a value to the formal parameter results in a run-time error.

The effect of passing an expression by name varies, depending on whether the expression evaluates as a reference to a single object. For example, A[I] evaluates into a reference to a single element of array A. In this guide, such an expression is referred to as a *simple expression.* Other examples of simple expressions are the POINTER function in ALGOL and references to character-based record fields. On the other hand, an expression such as A + B does not evaluate as a reference to a single element. Such an expression is referred to as a *complex expression.*

For a simple expression, the system passes a thunk that reevaluates the expression each time the parameter is used in the procedure. For example, suppose the actual parameter A[I] is passed to the formal parameter F. At the time of the procedure invocation, I has a value of 5. The formal parameter F becomes a reference to element 5 of array A. When F is read, it reflects the most recent value of A[5]. When F is assigned, it changes the value of A[5]. If I is then assigned a value of 10, F becomes a reference to A[10]. Thereafter, reading or assigning F really accesses the value stored in A[10].

For a complex expression, the system passes a thunk that reevaluates the expression each time the formal parameter is read in the procedure. However, it is impossible to assign a value to the formal parameter; any attempt to do so results in a run-time error.

Call-by-Reference Parameters

When a parameter is passed by reference, the system passes the formal parameter a reference to the place where the actual parameter is stored in memory. Passing a parameter by reference is essentially the same as passing it by name, except that the compiler does not create a thunk for a call-by-reference parameter. Any expressions passed by reference are, therefore, evaluated immediately and changed into simple values or pointers to simple values.

The effects of passing a parameter by reference are somewhat different in FORTRAN77 than in other languages. In non-FORTRAN languages, the effects of passing by reference are as follows:

- When a simple variable is passed by reference, the effect is the same as if it had been passed by name. Changes made to the value of the formal parameter immediately affect the value of the actual parameter and vice versa.
- In most languages, constants and complex expressions cannot be passed by reference; a syntax error results from an attempt to do so. However, simple expressions can be passed by reference. For a simple expression, the system passes a reference to the location of the element. This location never changes, even if the value of the subscript later changes. For example, suppose the actual parameter A[I] is passed to the formal parameter F and I has a value of 5. Formal parameter F becomes a reference to array element A[5]. Even if I is later assigned a different value, F remains a reference to A[5]. When F is read, it reflects the most recent value of A[5]. When F is assigned, it changes the value of A[5].

In FORTRAN77 the effects of passing by reference are as follows:

- For simple variables of type integer, real, double precision, complex, or logical, two different kinds of call-by-reference passing are available. The default method is known as *call-by-value-result*. With this method, the value of the actual parameter is assigned to the formal parameter. Thereafter, assignments to the actual parameter have no effect on the formal parameter. Assignments to the formal parameter have no immediate effect on the actual parameter; however, when the procedure is exited, the value of the formal parameter is assigned to the actual parameter. The alternate method is true *call-by-reference* passing, in which the formal parameter receives a reference to the actual parameter itself; changes to the actual parameter are immediately visible to the formal parameter and vice versa. The programmer can request true call-by-reference passing by enclosing the formal parameter in slashes (/).
- For parameters that are character variables, arrays, or subprograms, the parameter is always treated as a true call-by-reference parameter. Any changes to the actual parameter are immediately visible to the formal parameter and vice versa.
- Constants of type integer, real, double precision, complex, or logical can be passed by reference, but character or array constants cannot. The receiving procedure can make assignments that change the value of the formal parameter, but the value of the actual parameter is never updated to reflect the change.
- For an actual parameter that is a simple expression, the parameter is treated as either call-by-value-result, or true call-by-reference, depending on the way the formal parameter is declared. If the formal parameter is a character variable or array, then the parameter is treated as a true call-by-reference parameter. If the formal parameter is an integer, real, double precision, complex, or logical variable, then by default the parameter is treated as call-by-value-result; however, if the formal parameter is enclosed in slashes, the parameter is treated as true call-by-reference.
- For an actual parameter that is a complex expression, the system evaluates the expression and passes the value to the formal parameter. The receiving procedure can make assignments that change the value of the formal parameter, but the value of the actual parameter remains unchanged.

Read-Only Parameters

A concept related to parameter passing modes is that of *read-only parameters*. The term “read-only” refers, not to a passing mode, but to a restriction on the ways a parameter can be used.

Formal parameter declarations in a Pascal program can include a CONST clause, which causes a parameter to be treated as a *read-only parameter*. The CONST clause prevents the receiving Pascal program or procedure from making any changes to the value of the formal parameter. However, the CONST clause does not guarantee that the formal parameter has a constant value. The formal parameter value can change because the CONST clause does not affect the passing mode. If the actual parameter is passed by name or by reference, then any changes made by the initiator to the value of the actual parameter are immediately reflected in the value of the formal parameter.

Specifying the Passing Mode

You will seldom have the opportunity to choose among all three of these passing modes for a particular parameter. The choice of passing modes is restricted on the basis of several different considerations, including parameter type, language, and process type.

Though there are many different parameter types, these types fall into two basic categories: *word* and *descriptor*. Boolean variables, integer variables, and real variables are examples of word types. Strings, arrays, files, and other complex data structures are descriptor types.

Word-type parameters can be passed by value, by name, or by reference.

In most languages, descriptor-type parameters must be passed by name or by reference. Exceptions are Pascal, which allows descriptor type parameters to be passed by value, and WFL, which can pass strings by value. Also, message control systems (MCSs) and Host Services tasking can pass descriptor type parameters by value. Host Services tasking makes it possible to write a program that passes an array to a remote process by value. (Remote processes are discussed in Section 12, “Tasking across Multihost Networks.”)

Each language imposes a different set of restrictions on the passing mode. For example, ALGOL passes descriptor types by name or by reference and word types by name, by reference, or by value. COBOL passes all library parameters by reference, and parameters to tasks or bound-in procedures by reference or by value. WFL passes parameters either by reference or by value. For details about these language restrictions, refer to the programming language reference manuals.

One additional restriction is based on the process type. A statement that initiates an independent process can pass parameters only by value, not by name or by reference.

Using Tasking Parameters

The system provides the application programmer with the ability to design a program in one language that initiates a program written in a different language. The initiating program can even pass parameters to the initiated program. However, because each language provides a different set of parameter types, the programmer needs to understand which types of parameters are compatible.

The languages that can initiate a process and pass it parameters are ALGOL, COBOL74, COBOL85, and Work Flow Language (WFL).

The languages that can receive tasking parameters from another program are ALGOL, C, COBOL74, COBOL85, AND Pascal.

WFL jobs can also receive parameters. However, strictly speaking, these are compile-time rather than tasking parameters because a WFL job is recompiled each time it is submitted. ALGOL, COBOL74, COBOL85, and RPG can all submit WFL jobs, but none of them can pass a parameter to the WFL job. Parameters can be passed to a WFL job only by a START statement. START statements can be submitted in Command and Edit (CANDE) or Menu-Assisted Resource Control (MARC) sessions or at an operator display terminal (ODT). START statements can also be submitted by DCALGOL programs using the DCKEYIN function or by WFL jobs.

The remainder of this section discusses only tasking parameters and not WFL compile-time parameters.

Whenever a process passes a tasking parameter, the system software checks that the number of actual parameters the calling program passes matches the number of formal parameters declared in the receiving program.

The system also compares each actual parameter with the matching formal parameter to determine if they are of compatible types. The matching is done based on parameter order rather than parameter names. It is permissible for the actual and formal parameters to have different names.

In many cases, the system allows matches between similar, though not identical, parameter types. For instance, an integer actual parameter can generally be passed to a real formal parameter. Also, types that are, in effect, identical might have different names in different languages. Details about which parameter matches are allowed by the system software are given under “Matching Each Parameter Type” later in this section.

Information about how the passing mode is determined for tasking parameters is given under “Resolving Passing Mode Conflicts” later in this section.

Special considerations for arrays passed as tasking parameters are discussed under “Passing Arrays” later in this section.

Matching Each Parameter Type

By using Tables 17-1 and 17-2 at the end of this subsection, you can find out what parameter types in a given language match particular parameter types in any other given language. In the following discussion, the term *original parameter* refers to the parameter you want to find a match for. The original parameter might be either an actual parameter or a formal parameter. The term *matching parameter* refers to the parameter about which you are uncertain. The tables can help you decide what type the matching parameter should be.

Note: *The tables in this section document the tasking parameter-type-matching rules enforced by the system at process initiation time. If you are initiating an imported library procedure, you should know about the library parameter matching rules discussed in Section 18, "Using Libraries." These rules are enforced by the operating system at library linkage time, and in general, are much stricter than the tasking parameter matching rules.*

Further, if you are initiating a bound-in procedure, you should know about the binding parameter matching rules discussed in the Binder Programming Reference Manual*. These rules are enforced by the Binder during its run, and in general, are more limiting than the library parameter matching rules.*

To use the parameter matching tables, you must know the following characteristics of the original parameter: the language, the name of the parameter type, and whether it is a formal or an actual parameter. For the matching parameter, you must know the language in which it will be specified.

Begin by looking at Table 17–1. Table 17–1 is separated into three columns labeled Language, Parameter Type, and General Type. Search the Language column until you find the language of your original parameter. Next, scan the Parameter Type column until you find the type of your original parameter. Next, in the General Type column to the right, note the general type listed there.

In some cases, the general type shown is "(Unique)" instead of a word or a phrase. This means your original parameter is a unique type that does not match any other parameter type. For example, an ALGOL Boolean direct array can pass only to another ALGOL Boolean direct array. In this case, you can skip the rest of these directions, because there are no other matching parameter types.

Next, look at Table 17–2, which extends over several pages with each page including one or more boxes. Each box is a separate entry with a General Type heading appearing at the upper left of each box. The boxes appear in alphabetical order based on the General Type headings. Look for the box whose General Type heading corresponds to the general type you noted earlier.

Within the box you selected, scan down the Language and Parameter Type columns. Make a note of the parameter types that are in the language you want to find out about.

At this stage, you can consider yourself finished. Take the parameter types you noted and look in the appropriate programming language reference manual for the detailed syntax of the parameter types. However, if this initial search did not uncover any parameters in the desired language, or if you want a complete list of the possible parameter types for the matching parameter, then the information in the Special Matches column can help you extend your search.

The Special Matches column of each box can include up to three subentries that list general types that match your original parameter, but only in some limited circumstances. Examine each subentry that appears in the Special Matches column of the box. The following are the possible subentries and their meanings:

- Matching Actuals

  This list of general types can match your original parameter, if your original parameter is a formal one and you are looking for an actual parameter to match it. If you are looking for an actual parameter, note each of these general types, and for each do the following:

  – Go to the box labeled with the name of the specific general type.

  – Look at the main parameter group in the box and note any parameter types shown that are in the language you want for your matching parameter.

  – Ignore any Matching Actuals, Matching Formals, or COBOL74 Matches subentries that appear in the box.

- Matching Formals

  This list of general types can match your original parameter, if your original parameter is an actual one and you are looking for a formal parameter to match it. To translate these general types into specific parameter types, follow the same steps that you did for the Matching Actuals subentry.

- COBOL74 Matches

  These general-type matches are allowed if the calling program or the receiving program is written in COBOL74. If true, then note the general types shown. For each general type, do the following:

  – Go to the box that is labeled with the name of the specific general type.

  – If the original parameter is in COBOL74, note any parameters shown in the Parameter Types column that are in the language you want for your matching parameter. If the original parameter is not in COBOL74, note only the COBOL74 types that appear in the Parameter Types column.

  – Ignore any Matching Actuals, Matching Formals, or COBOL Matches subentries that appear in the box.

You now have a complete list of the possible parameter types for your matching parameter. Refer to the various programming language reference manuals for the syntax used to declare the parameter types you have listed.

Note that the programming languages restrict some parameter types so that they can be used only as formal parameters or only as actual parameters. The syntax given in the programming language reference manuals should explain any such restrictions.

The following examples illustrate the method for finding matching parameter types.

Example 1

Suppose you want to pass a string value from a WFL job to an ALGOL program. Look at the last line of Table 17–1. The parameter type shown is WFL STRING. The general type shown is Real Array.

Now look through Table 17–2 until you find the box labeled Real Array. The main parameter group in the box includes two ALGOL types: REAL ARRAY and REAL VALUE ARRAY.

In addition, the Matching Formals list specifies the general type integer array. Look at the box labeled Integer Array. The main parameter group in the box includes the following ALGOL types: INTEGER ARRAY and INTEGER VALUE ARRAY.

You now have a list of four different ALGOL parameter types. However, if you refer to the ALGOL manual, you will find that value arrays are not allowed as formal parameters (although they can be actual parameters). Therefore, a WFL STRING parameter can be passed to two ALGOL types: REAL ARRAY or INTEGER ARRAY.

Example 2

Suppose you want to pass a 01 DISPLAY Group Item from a COBOL74 program to an ALGOL program. Look through Table 17–1 until you find the line that says COBOL74 01 DISPLAY Group Item. The general type shown is EBCDIC Array.

Now look through Table 17–2 until you find the box labeled EBCDIC Array. The ALGOL parameter types shown in the box are EBCDIC ARRAY and EBCDIC VALUE ARRAY. Note these.

The Special Matches column of the box includes three general types as COBOL74 Matches: Hex Array, Integer Array, and Real Array. Go to the box for Hex Array. The Parameter Types column includes two ALGOL types: HEX ARRAY and HEX VALUE ARRAY. Note these. Repeat this process for each of the general types that you noted.

When you have finished this process, you have the following list of ALGOL formal parameter types:

```
EBCDIC ARRAY
EBCDIC VALUE ARRAY
HEX ARRAY
HEX VALUE ARRAY
INTEGER ARRAY
INTEGER VALUE ARRAY
REAL ARRAY
REAL VALUE ARRAY
```

Of these, you should discard the value arrays because they cannot be used as formal parameters. The matching parameter could be any of the remaining ALGOL types from this list.

Example 3

Suppose you want to pass a real array from an ALGOL program to a COBOL74 program. Scan through the ALGOL parameters in Table 17–1 until you find REAL ARRAY. The General Type shown is also Real Array.

Now look through Table 17–2 until you find the box labeled Real Array. One COBOL74 parameter is shown in the Parameter Types column: 01 BINARY Group Item. Note this. Additionally, the box contains entries in the Special Matches column for Matching Actuals, Matching Formals, and COBOL74 Matches. You can interpret these entries as follows:

- Matching Actuals

  Ignore this entry, as your original parameter is the actual parameter. The matching parameter you are looking for is a formal parameter.

- Matching Formals

  The entry shown under this heading is Integer Array. Go to the box for Integer Array. In the Parameter Type column, you find one COBOL74 parameter: 77 BINARY Elementary Item. Make a note of this.

- COBOL74 Matches

  The entries shown under this heading are EBCDIC Array and Hex Array.

  – Go to the box for EBCDIC Array. The COBOL74 parameters shown in the Parameter Type column are 01 DISPLAY Group Item and 01 KANJI Group Item. Make a note of these.

  – Go to the box for Hex Array. The COBOL74 parameters shown in the Parameter Type column are 01 COMP Group Item and 01 INDEX Group Item. Make a note of these.

When you finish this process, you find that the COBOL74 formal parameter can be of any of the following types:

```
01 BINARY Group Item
01 COMP Group Item
01 DISPLAY Group Item
01 INDEX Group Item
01 KANJI Group Item
77 BINARY Elementary Item
```

Example 4

Suppose you want to pass a HEX DIRECT ARRAY from an ALGOL program to a COBOL74 program. Look through Table 17–1 until you find the line that lists ALGOL HEX DIRECT ARRAY. The General Type column lists “(Unique)” instead of the general type. This means that HEX DIRECT ARRAY is a unique parameter type that can match only a parameter of exactly the same type. In this case, there is no need for you to look at Table 17–2.

Table 17–1. Programming Language Parameter Types

Language	Parameter Type	General Type
ALGOL	ASCII PROCEDURE REFERENCE ARRAY	(Unique)
ALGOL	ASCII ARRAY	ASCII Array
ALGOL	ASCII DIRECT ARRAY	(Unique)
ALGOL	ASCII PROCEDURE	(Unique)
ALGOL	ASCII STRING	(Unique)
ALGOL	ASCII STRING ARRAY	(Unique)
ALGOL	ASCII VALUE ARRAY	ASCII Array
ALGOL	BOOLEAN	Boolean
ALGOL	BOOLEAN ARRAY	Boolean Array
ALGOL	BOOLEAN DIRECT ARRAY	(Unique)
ALGOL	BOOLEAN PROCEDURE	Boolean Procedure
ALGOL	BOOLEAN PROCEDURE REFERENCE ARRAY	(Unique)
ALGOL	BOOLEAN VALUE ARRAY	Boolean Array
ALGOL	COMPLEX	(Unique)
ALGOL	COMPLEX ARRAY	Complex Array
ALGOL	COMPLEX PROCEDURE	(Unique)
ALGOL	COMPLEX PROCEDURE REFERENCE ARRAY	(Unique)
ALGOL	COMPLEX VALUE ARRAY	Complex Array
ALGOL	DIRECT FILE	(Unique)
ALGOL	DIRECT SWITCH FILE	(Unique)
ALGOL	DOUBLE	Double
ALGOL	DOUBLE ARRAY	Double Array
ALGOL	DOUBLE DIRECT ARRAY	(Unique)
ALGOL	DOUBLE PROCEDURE	Double Procedure
ALGOL	DOUBLE PROCEDURE REFERENCE ARRAY	(Unique)
ALGOL	DOUBLE VALUE ARRAY	Double Array
ALGOL	EBCDIC ARRAY	EBCDIC Array
ALGOL	EBCDIC DIRECT ARRAY	(Unique)
ALGOL	EBCDIC PROCEDURE ARRAY	(Unique)
ALGOL	EBCDIC STRING	(Unique)
ALGOL	EBCDIC STRING ARRAY	(Unique)
ALGOL	EBCDIC VALUE ARRAY	(Unique)
ALGOL	EBCDIC PROCEDURE REFERENCE ARRAY	(Unique)

Table 17–1. Programming Language Parameter Types

Language	Parameter Type	General Type
ALGOL	ENTITY REFERENCE	(Unique)
ALGOL	ENTITY REFERENCE ARRAY	(Unique)
ALGOL	EPILOG PROCEDURE	(Unique)
ALGOL	EVENT	Event
ALGOL	EVENT ARRAY	Event Array
ALGOL	FILE	File
ALGOL	FORMAT	(Unique)
ALGOL	HEX ARRAY	Hex Array
ALGOL	HEX DIRECT ARRAY	(Unique)
ALGOL	HEX PROCEDURE	(Unique)
ALGOL	HEX STRING	(Unique)
ALGOL	HEX STRING ARRAY	(Unique)
ALGOL	HEX VALUE ARRAY	Hex Array
ALGOL	HEX PROCEDURE REFERENCE ARRAY	(Unique)
ALGOL	INTEGER	Integer
ALGOL	INTEGER ARRAY	Integer Array
ALGOL	INTEGER DIRECT ARRAY	Integer Direct Array
ALGOL	INTEGER PROCEDURE	Integer Procedure
ALGOL	INTEGER PROCEDURE REFERENCE ARRAY	(Unique)
ALGOL	INTEGER VALUE ARRAY	Integer Array
ALGOL	INTERLOCK	Interlock
ALGOL	INTERLOCK ARRAY	Interlock
ALGOL	LABEL	(Unique)
ALGOL	LIST	(Unique)
ALGOL	PICTURE	(Unique)
ALGOL	PICTURE ARRAY	(Unique)
ALGOL	POINTER	Pointer
ALGOL	PROCEDURE (SUBROUTINE)	Procedure
ALGOL	QUERY VARIABLE	(Unique)
ALGOL	QUEUE	(Unique)
ALGOL	QUEUE ARRAY	(Unique)
ALGOL	REAL	Real
ALGOL	REAL ARRAY	Real Array

Table 17–1. Programming Language Parameter Types

Language	Parameter Type	General Type
ALGOL	REAL DIRECT ARRAY	Real Direct Array
ALGOL	REAL PROCEDURE	Real Procedure
ALGOL	REAL PROCEDURE REFERENCE ARRAY	(Unique)
ALGOL	REAL VALUE ARRAY	Real Array
ALGOL	SWITCH	(Unique)
ALGOL	SWITCH FILE	(Unique)
ALGOL	SWITCH FORMAT	(Unique)
ALGOL	SWITCH LIST	(Unique)
ALGOL	TASK	Task
ALGOL	TASK ARRAY	Task Array
ALGOL	TRANSACTION RECORD	ALGOL Transaction Record
ALGOL	TRANSACTION RECORD ARRAY	ALGOL Transaction Record
ALGOL	UNTYPED PROCEDURE REFERENCE ARRAY	(Unique)
C	int argc, char* argv[]	Real Array (unbounded)
COBOL74	01 BINARY Group Item	Real Array
COBOL74	01 COMP Group Item	Hex Array
COBOL74	01 CONTROL-POINT Elementary Item	Task
COBOL74	01 CONTROL-POINT Group Item	Task Array
COBOL74	01 DISPLAY Group Item	EBCDIC Array
COBOL74	01 EVENT Group Item	Event Array
COBOL74	01 INDEX Group Item	Hex Array
COBOL74	01 KANJI Group Item	EBCDIC Array
COBOL74	01 LOCK Group Item	Event Array
COBOL74	77 BINARY Elementary Item	Integer
COBOL74	77 CONTROL-POINT Elementary Item	Task
COBOL74	77 DOUBLE Elementary Item	Double
COBOL74	77 EVENT Elementary Item	Event
COBOL74	77 LOCK Elementary Item	Event
COBOL74	77 REAL Elementary Item	Real

Table 17–1. Programming Language Parameter Types

Language	Parameter Type	General Type
COBOL74	File	File
COBOL74	Transaction Record	Transaction Record
COBOL85	01 BINARY Group Item	Integer Array
COBOL85	01 COMP Group Item	Hex Array
COBOL85	01 REAL Group Item	Real Array
COBOL85	01 DOUBLE Group Item	Double Array
COBOL85	01 DISPLAY Group Item	EBCDIC Array
COBOL85	77 REAL Elementary Item	Real
COBOL85	77 DOUBLE Elementary Item	Double
COBOL85	77 BINARY PIC 9(1-11) Elementary Item	Integer
COBOL85	77 BINARY PIC 9(11-23) Elementary Item	Double
COBOL85	77 File	File
Pascal	Array of Boolean	Boolean Array
Pascal	Array of Char	Integer Array
Pascal	Array of Char Subrange	Integer Array
Pascal	Array of Enumeration	Integer Array
Pascal	Array of Enumeration Subrange	Integer Array
Pascal	Array of Explicit Data Type	Real Array
Pascal	Array of Fixed (n < 12)	Integer Array
Pascal	Array of Fixed (n > 11)	Double Array
Pascal	Array of Integer	Integer Array
Pascal	Array of Integer Subrange	Integer Array
Pascal	Array of Packed Array	Real Array
Pascal	Array of Real	Real Array
Pascal	Array of Record	Real Array
Pascal	Array of Set	Real Array
Pascal	Array of Sfixed (n < 12)	Integer Array
Pascal	Array of Sfixed (n > 11)	Double Array
Pascal	Array of Vlstring	Real Array
Pascal	Binary (n)	EBCDIC Array
Pascal	Bits (n)	EBCDIC Array

Table 17–1. Programming Language Parameter Types

Language	Parameter Type	General Type
Pascal	Boolean	Boolean
Pascal	Boolean Subrange	Boolean
Pascal	Boolean1	Hex Array
Pascal	Boolean4	Hex Array
Pascal	Char	Integer
Pascal	Char Subrange	Integer
Pascal	Digits (n)	Hex Array
Pascal	Digits_s (n)	Hex Array
Pascal	Display_s (n)	EBCDIC Array
Pascal	Display_z (n)	EBCDIC Array
Pascal	Enumeration	Integer
Pascal	Enumeration Subrange	Integer
Pascal	Explicit Record (call-by-value)	Real Array
Pascal	Explicit Record (var)	EBCDIC Array
Pascal	Fixed (n < 12)	Integer
Pascal	Fixed (n > 11)	Double
Pascal	Function: Boolean	Boolean Procedure
Pascal	Function: Boolean Subrange	Boolean Procedure
Pascal	Function: Char	Integer Procedure
Pascal	Function: Char Subrange	Integer Procedure
Pascal	Function: Enumeration	Integer Procedure
Pascal	Function: Enumeration Subrange	Integer Procedure
Pascal	Function: Fixed (n < 12)	Integer Procedure
Pascal	Function: Fixed (n > 11)	Double Procedure
Pascal	Function: Integer	Integer Procedure
Pascal	Function: Integer Subrange	Integer Procedure
Pascal	Function: Real	Real Procedure
Pascal	Function: Sfixed (n < 12)	Integer Procedure
Pascal	Function: Sfixed (n > 11)	Double Procedure
Pascal	Hex (n)	Hex Array
Pascal	Integer	Integer
Pascal	Integer Subrange	Integer
Pascal	Integer48 (n)	EBCDIC Array

Table 17–1. Programming Language Parameter Types

Language	Parameter Type	General Type
Pascal	Integer96 (n)	EBCDIC Array
Pascal	Long Set (> 48 Elements in Set)	Real Array
Pascal	Packed Array of Boolean	Hex Array
Pascal	Packed Array of Char	EBCDIC Array
Pascal	Packed Array of Enumeration	
	(0-16 Elements)	Hex Array
	(17-256 Elements)	EBCDIC Array
	(> 256 Elements)	Integer Array
Pascal	Packed Array of Fixed (n < 12)	Integer Array
Pascal	Packed Array of Fixed (n > 11)	Double Array
Pascal	Packed Array of Integer	Integer Array
Pascal	Packed Array of Real	Real Array
Pascal	Packed Array of Record	Real Array
Pascal	Packed Array of Set	Real Array
Pascal	Packed Array of Sfixed (n < 12)	Integer Array
Pascal	Packed Array of Sfixed (n > 11)	Double Array
Pascal	Packed Array of Subrange	
	(0-16 Elements)	Hex Array
	(17-256 Elements)	EBCDIC Array
	(> 256 Elements)	Integer Array
Pascal	Packed Array Of Vlstring	Real Array
Pascal	Procedure	Procedure
Pascal	Real	Real
Pascal	Real48 (n)	EBCDIC Array
Pascal	Record	Real Array
Pascal	Schema	Refer to "Passing Parameters to Pascal Schemata" later in this section.
Pascal	Sfixed (n < 12)	Integer
Pascal	Sfixed (n > 11)	Double
Pascal	Short Set (1-48 Elements In Set)	Real
Pascal	S_digits (n)	Hex Array
Pascal	S_display (n)	EBCDIC Array

Table 17–1. Programming Language Parameter Types

Language	Parameter Type	General Type
Pascal	U_display (n)	EBCDIC Array
Pascal	Vlstring	Real Array
Pascal	Word48 (n)	EBCDIC Array
Pascal	Word96 (n)	EBCDIC Array
Pascal	Z_display (n)	EBCDIC Array
WFL	BOOLEAN	Boolean
WFL	INTEGER	Integer
WFL	REAL	Real
WFL	STRING	Real Array

Table 17–2. Matching Parameter Types

General Type	Language	Parameter Type	Special Matches
ASCII Array	ALGOL	ASCII ARRAY	
	ALGOL	ASCII VALUE ARRAY	
Boolean	ALGOL	BOOLEAN	
	Pascal	Boolean	
	Pascal	Boolean Subrange	
	WFL	BOOLEAN	
			Matching Actuals: • Boolean Procedure (with no parameters) • Integer • Real Matching Formals: • Integer • Real
Boolean Array	ALGOL	BOOLEAN ARRAY	
	ALGOL	BOOLEAN VALUE ARRAY	
	Pascal	Array of Boolean	

Table 17–2. Matching Parameter Types

General Type	Language	Parameter Type	Special Matches
Boolean Procedure	ALGOL	BOOLEAN PROCEDURE	
	ALGOL	BOOLEAN VALUE ARRAY	
	Pascal	Array of Boolean	
	Pascal	Function: Boolean	
Complex Array	ALGOL	COMPLEX ARRAY	
	ALGOL	COMPLEX VALUE ARRAY	
Direct File	ALGOL	DIRECT FILE	
Double	ALGOL	DOUBLE	
	COBOL74	77 DOUBLE Elementary Item	
	COBOL85	77 BINARY PIC 9(11-23) Elementary Item	
	COBOL85	77 DOUBLE Elementary Item	
	Pascal	Fixed (n > 11)	
	Pascal	Sfixed (n > 11)	
Double Array	ALGOL	DOUBLE ARRAY	
	ALGOL	DOUBLE VALUE ARRAY	
	COBOL85	01 DOUBLE Group Item	
	Pascal	Array of Fixed (n > 11)	
	Pascal	Array of Sfixed (n > 11)	
	Pascal	Packed Array of Fixed (n > 11)	
	Pascal	Packed Array of Sfixed (n > 11)	
Double Procedure	Pascal	Function: Fixed (n>11)	
	Pascal	Function: Sfixed (n>11)	
EBCDIC Array	ALGOL	EBCDIC ARRAY	
	ALGOL	EBCDIC VALUE ARRAY	
	COBOL74	01 DISPLAY Group Item	
	COBOL74	01 KANJI Group Item	

Table 17–2. Matching Parameter Types

General Type	Language	Parameter Type	Special Matches
EBCDIC Array (cont.)	COBOL85	01 DISPLAY Group Item	
	Pascal	Binary (n)	
	Pascal	Bits (n)	
	Pascal	Display s (n)	
	Pascal	Display z (n)	
	Pascal	Explicit Record (var)	
	Pascal	Integer48 (n)	
	Pascal	Integer96 (n)	
	Pascal	Packed Array of Enumeration (17-256 Elements in Enumeration)	
	Pascal	Packed Array of Subrange (17-256 Elements in Subrange)	
	Pascal	Packed Array of Char	
	Pascal	Real48 (n)	
	Pascal	S display (n)	
	Pascal	U display (n)	
	Pascal	Word48 (n)	
	Pascal	Word96 (n)	
	Pascal	Z display (n)	
			Matching Actuals: • Integer Array Matching Formals: • Integer Array COBOL74 Matches: • Hex Array • Integer Array • Real Array
Event	ALGOL	EVENT	
	COBOL74	77 EVENT Elementary Item	
	COBOL74	77 LOCK Elementary Item	

Table 17–2. Matching Parameter Types

General Type	Language	Parameter Type	Special Matches
Event Array	ALGOL	EVENT ARRAY	
	COBOL74	01 EVENT Group Item	
	COBOL74	01 LOCK Group Item	
File	ALGOL	FILE	
	COBOL74	File	
	COBOL85	77 File	
Hex Array	ALGOL	HEX ARRAY	
	ALGOL	HEX VALUE ARRAY	
	COBOL74	01 COMP Group Item	
	COBOL74	01 INDEX Group Item	
	COBOL85	01 COMP Group Item	
	Pascal	Boolean1	
	Pascal	Boolean4	
	Pascal	Digits s (n)	
	Pascal	Digits (n)	
	Pascal	Hex (n)	
	Pascal	Packed Array of Enumeration (0-16 Elements)	
	Pascal	Packed Array of Subrange (0-16 Elements)	
	Pascal	Packed Array of Boolean	
	Pascal	S digits (n)	
			Matching Actuals: • Integer Array Matching Formals: • Integer Array COBOL74 Matches: • EBCDIC Array • Integer Array • Real Array

Table 17–2. Matching Parameter Types

General Type	Language	Parameter Type	Special Matches
Integer	ALGOL	INTEGER	
	COBOL74	77 BINARY Elementary Item	
	COBOL85	77 BINARY PIC 9(1-11) Elementary Item	
	Pascal	Char	
	Pascal	Char Subrange	
	Pascal	Enumeration	
	Pascal	Enumeration Subrange	
	Pascal	Fixed (n < 12)	
	Pascal	Integer	
	Pascal	Integer Subrange	
	Pascal	Sfixed (n < 12)	
	WFL	INTEGER	
			Matching Actuals: • Boolean • Integer Procedure (with no parameters) • Real • Real Procedure (with no parameters) Matching Formals: • Boolean • Real
Integer Array	ALGOL	INTEGER ARRAY	
	ALGOL	INTEGER VALUE ARRAY	
	COBOL85	01 BINARY Group Item	
	Pascal	Array of Char	
	Pascal	Array of Char Subrange	
	Pascal	Array of Enumeration	
	Pascal	Array of Enumeration Subrange	
	Pascal	Array of Fixed (n < 12)	
	Pascal	Array of Integer	
	Pascal	Array of Integer Subrange	

Table 17–2. Matching Parameter Types

General Type	Language	Parameter Type	Special Matches
Integer Array (cont.)	Pascal	Array of Sfixed (n < 12)	
	Pascal	Packed Array of Enumeration (> 256 elements)	
	Pascal	Packed Array of Fixed (n < 12)	
	Pascal	Packed Array of Integer	
	Pascal	Packed Array of Subrange (> 256 Elements)	
	Pascal	Packed Array of Sfixed (n < 12)	
			Matching Actuals: • EBCDIC Array • Hex Array • Real Array Matching Formals: • EBCDIC Array • Hex Array • Real Array
Integer Direct Array	ALGOL	INTEGER DIRECT ARRAY	
			Matching Actuals: • Real Direct Array Matching Formals: • Real Direct Array
Integer Procedure	ALGOL	INTEGER PROCEDURE	
	Pascal	Function: Char	
	Pascal	Function: Char Subrange	
	Pascal	Function: Enumeration	
	Pascal	Function: Enumeration Subrange	
	Pascal	Function: Fixed (n < 12)	
	Pascal	Function: Integer	

Table 17–2. Matching Parameter Types

General Type	Language	Parameter Type	Special Matches
Integer Procedure (cont.)	Pascal	Function: Integer Subrange	
	Pascal	Function: Sfixed (n < 12)	
			Matching Actuals: • Real Procedure Matching Formals: • Integer • Real • Real Procedure
Interlock	ALGOL	INTERLOCK	
Interlock Array	ALGOL	INTERLOCK ARRAY	
Pointer	ALGOL	POINTER	
Procedure	ALGOL	PROCEDURE (SUBROUTINE)	
	Pascal	Procedure	
Real	ALGOL	REAL	
	COBOL74	77 REAL Elementary Item	
	COBOL85	77 REAL Elementary Item	
	Pascal	Real	
	Pascal	Short Set (1-48 Elements in Set)	
	WFL	REAL	
			Matching Actuals: • Boolean • Integer • Integer Procedure • Real Procedure (with no parameters) Matching Formals: • Boolean • Integer

Table 17–2. Matching Parameter Types

General Type	Language	Parameter Type	Special Matches
Real Array	ALGOL	REAL ARRAY	
	ALGOL	REAL VALUE ARRAY	
	C	Int argc, Char *argv	
	COBOL74	01 BINARY Group Item	
	COBOL85	01 REAL Group Item	
	Pascal	Array of Explicit Data Type	
	Pascal	Array of Packed Array	
	Pascal	Array of Real	
	Pascal	Array of Record	
	Pascal	Array of Set	
	Pascal	Array of Vlstring	
	Pascal	Explicit Record (call-by-value)	
	Pascal	Long Set (> 48 Elements in Set)	
	Pascal	Packed Array of Real	
	Pascal	Packed Array of Record	
	Pascal	Packed Array of Set	
	Pascal	Packed Array of Vlstring	
	Pascal	Record	
	Pascal	Vlstring	
	WFL	STRING	
			Matching Actuals: • Integer Array Matching Formals: • Integer Array COBOL74 Matches: • EBCDIC Array • Hex Array

Table 17–2. Matching Parameter Types

General Type	Language	Parameter Type	Special Matches
Real Direct Array	ALGOL	REAL DIRECT ARRAY	
			Matching Actuals: • Integer Direct Array Matching Formals: • Integer Direct Array
Real Procedure	ALGOL	REAL PROCEDURE	
	Pascal	Function: Real	
			Matching Actuals: • Integer Procedure Matching Formals: • Integer • Integer Procedure • Real
Task	ALGOL	TASK	
	COBOL74	01 CONTROL-POINT Elementary Item	
	COBOL74	77 CONTROL-POINT Elementary Item	
Task Array	ALGOL	TASK ARRAY	
	COBOL74	01 CONTROL-POINT Group Item	
Transaction Record	ALGOL	TRANSACTION RECORD	
	ALGOL	TRANSACTION RECORD ARRAY	
	COBOL74	Transaction Record	

Resolving Passing Mode Conflicts

Some programming languages, such as WFL and ALGOL, allow the initiating program to specify the passing mode for a tasking parameter. In addition, programming languages typically allow the receiving program to specify a passing mode for the formal parameter. Thus, it is possible for the calling program and the receiving program to request different passing modes for the same parameter.

The system is very forgiving of these types of mismatches and generally allows any combination of actual and formal passing modes without issuing an error. However, when the calling program and the receiving program request different passing modes, the system uses the passing mode requested by the calling program. For example, if a call-by-value actual parameter is passed to a call-by-reference formal parameter, the system passes the parameter by value.

Note that the system is less forgiving of passing mode mismatches for parameters passed to library procedures. For a discussion of the allowable passing mode combination for library procedures, refer to Section 18, "Using Libraries."

Be very careful when writing a program that is to be initiated, and passed a parameter, by calling programs written by other people. The calling program might use a different passing mode for the parameter than you expected. For example, you might design the receiving program to receive a parameter by value, and make assignments to the parameter. However, if the calling program actually passes an expression by name, then the receiving program terminates with an error when it attempts to assign a value to the formal parameter. This is true because the calling program implicitly passed a thunk, and it is not possible to store values into thunks. You can avoid these types of problems by not making assignments to the formal parameter.

There is one type of passing mode problem that can make it impossible even for the receiving program to read the value of the formal parameter. If the calling program specifies a constant or an expression as a call-by-name actual parameter, then the compiler creates a thunk. If the receiving program specifies the formal parameter as call-by-reference, then the formal parameter cannot receive a thunk. The calling program can initiate the receiving program successfully. However, when the receiving program attempts to interrogate or modify the value of the formal parameter, the system issues an "INVALID OPERATOR" error and discontinues the receiving program.

Note that this error does not occur if the call-by-name actual parameter is a variable, rather than a constant or an expression. If a variable is used, then the compiler does not create a thunk. The receiving program can use the formal parameter without any problems.

Examples

Suppose that the following COBOL74 program is the receiving program. Note that the formal parameter specification indicates the parameter REAL-PARAM is to be received by reference:

```
100 IDENTIFICATION DIVISION.
110 ENVIRONMENT DIVISION.
120 DATA DIVISION.
130 WORKING-STORAGE SECTION.
140  77 REAL-PARAM BINARY PIC 9(11) RECEIVED BY REFERENCE.
150 PROCEDURE DIVISION USING REAL-PARAM.
160 START-HERE SECTION.
170 P1.
180     MOVE 15 TO REAL-PARAM.
190 STOP RUN.
```

The following ALGOL program invokes the preceding program and passes the real variable ACTUALREAL as the actual parameter. Note that the statement at line 150 in the following example specifies that the parameter be passed by value. This statement overrides the RECEIVED BY REFERENCE clause and causes the parameter to be passed by value. When the receiving program assigns a value of 15 to the formal parameter, the value of the actual parameter is not affected. Thus, the statement at line 210 displays a value of 5; but if the statement at line 150 were deleted, the statement at line 210 would display a value of 15.

```
100 BEGIN
110 FILE TERM (KIND=REMOTE);
120 TASK T;
130 REAL ACTUALREAL;
140 PROCEDURE COBOLTASK (RVAL);
150    VALUE RVAL;
160    REAL RVAL;
170    EXTERNAL;
180 ACTUALREAL := 5;
190 REPLACE T.NAME BY "OBJECT/COBOL/TASK.";
200 CALL COBOLTASK (ACTUALREAL) [T];
210 WRITE (TERM,*//,ACTUALREAL);
220 END.
```

Now suppose that the COBOL74 program was invoked by a WFL job instead. The following WFL job invokes the COBOL74 program and passes a real parameter by value (the default passing mode in WFL):

```
100 ?BEGIN JOB WFL/TEST;
110    CLASS = 2;
120    JOBSUMMARY = SUPPRESSED;
130    ELAPSEDLIMIT = 120;
140 REAL R:= 5;
150 RUN OBJECT/COBOL/TASK (R);
160 DISPLAY STRING(R,*);
170 ?END JOB
```

The statement at line 160 displays a value of 5. However, if you change the RUN statement to read *RUN OBJECT/COBOL/TASK (R REFERENCE);* then the parameter is passed by reference and the statement at line 160 displays a value of 15.

Passing Arrays

When an array is passed as a parameter, the actual and formal arrays must be of compatible data types (such as integer, real, and so on). The actual and formal arrays must also be compatible structurally. That is, the number of dimensions and the lower bounds for each dimension must be compatible.

The following subsections discuss these types of compatibility issues for arrays that are passed in process initiation statements. Note that this discussion centers on the compatibility issues the system enforces at run time. If the parameter is passed between procedures in a single program, the compiler can enforce additional restrictions at compile time. For information about any such compile-time restrictions, refer to the appropriate programming language manuals.

Matching Dimensions and Elements

When the calling program passes arrays, the actual array and the formal array must have the same number of dimensions.

However, it is not necessary for the actual array and the formal array to have the same number of elements in each dimension. Some languages allow formal array parameters that do not specify the number of elements in each dimension. For example, ALGOL does not allow upper bounds to be specified for the dimensions in a formal array parameter specification; and Pascal allows formal array parameters, called *schemata,* that are incompletely specified. (Schemata are discussed under "Passing Parameters to Pascal Schemata" later in this section.) In these cases, the system assigns the formal parameter the same number of elements as the actual parameter at run time.

Even if the formal parameter specifies the number of elements in each dimension of an array, the actual parameter can have a different number of elements. The system does not issue an error or warning in these cases. If the actual parameter passes more elements than the formal parameter can receive, the system ignores the extra elements.

Matching Unbounded Arrays

Some languages, such as ALGOL, allow formal array parameters that do not specify the lower bounds for array dimensions. Such array parameters are referred to in this guide as *unbounded array parameters*. Array parameters that explicitly specify the lower bounds are referred to as *simple array parameters*.

Be aware that parameter mismatch errors can result from passing an actual array with an unspecified lower bound to a formal array with a specified lower bound, or vice versa. For example, WFL STRING parameters are passed as unbounded real arrays. If a WFL program passes a string parameter to an ALGOL program, the ALGOL program must declare the formal parameter as unbounded; otherwise, a PARAMETER MISMATCH error occurs at run time.

The following is an example of an ALGOL program that is passed a string parameter from a WFL job.

```
100 PROCEDURE OUTER(ARR);
110    REAL ARRAY   ARR[*];
120 BEGIN
130 FILE TERM(KIND=REMOTE);
140 INTEGER ARR_SIZE;
150 POINTER P;
160 P:= ARR;
170 ARR_SIZE:= SIZE(ARR) * 6;
180 WRITE(TERM,*//,P FOR ARR_SIZE);
190 END.
```

In the preceding program, the SIZE function at line 170 returns the size of the array parameter in words. This value is multiplied by 6 to give the length of the array parameter in characters.

COBOL74 and COBOL85 are somewhat more forgiving than ALGOL in their handling of array parameters. Formal array parameters in COBOL74 or COBOL85 programs can receive either simple or unbounded actual parameters. Consider the following COBOL74 or COBOL85 program:

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 PARAM PIC X(12) DISPLAY.
PROCEDURE DIVISION USING PARAM.
START-HERE SECTION.
P1.
    DISPLAY PARAM.

    STOP RUN.
```

The preceding COBOL74 program is initiated twice by the following ALGOL program. The first time, the ALGOL program passes an unbounded array parameter. The second time, the ALGOL program passes a simple array parameter. In each case, the actual parameter is received by the formal parameter PARAM in the COBOL74 program. The COBOL74 program runs normally and displays the same output in each case.

```
BEGIN
REAL ARRAY ARRIN[0:12];
TASK T;
PROCEDURE EX1(ARRACT);
  REAL ARRAY ARRACT[*];
  EXTERNAL;
PROCEDURE EX2(ARRACT);
  REAL ARRAY ARRACT[0];
  EXTERNAL;
REPLACE ARRIN BY "HI THERE";
REPLACE T.NAME BY "(JASMITH)OBJECT/TEST/COBOL/TASK.";
```

```
210   CALL EX1 (ARRIN) [T];
220   CALL EX2 (ARRIN) [T];
230 END.
```

Note that the preceding comments about COBOL74 and COBOL85 hold true only for tasking parameters. COBOL74 and COBOL85 programs display less flexible behavior when they are invoked as libraries. In this case, the programmer must know in advance whether the actual array parameter is simple or unbounded. If the actual parameter is unbounded, the programmer must use a LOWER-BOUNDS clause in the formal array declaration, or else declare an extra BINARY parameter to receive the lower bound. Of these two techniques, the LOWER-BOUNDS clause is equally compatible with tasking or library calls, whereas the extra BINARY parameter works only for library calls.

If a program has a single parameter that is a single-dimensional unbounded array, and no parameter is passed when the program is initiated, a one word array is supplied containing nulls, 4"000000000000". An example ALGOL program heading would be

```
$ SET LEVEL 2
PROCEDURE P(A);
ARRAY     A[*];
BEGIN
```

For further information about unbounded array parameters to library procedures, refer to Section 18, “Using Libraries.”

Matching Pascal Arrays

Some special rules apply for passing parameters to a Pascal formal parameter that is either a multidimensional array or an incompletely defined array.

Passing Multidimensional Arrays

Pascal arrays are all stored internally as one-dimensional arrays. Declaring a Pascal array with multiple dimensions creates an indexing compiler scheme, which makes it appear that the array has multiple dimensions. Within the Pascal program, the fact that the array is one-dimensional is never visible. However, this fact is visible when parameters are passed to a Pascal program from a program written in another language.

Because Pascal formal array parameters are implicitly one dimensional, actual array parameters passed to Pascal programs must always be one-dimensional. The elements of the actual array are mapped into the formal array according to an algorithm that increments the indexes for the highest dimension, then the next highest dimension, and so on.

For example, suppose the actual parameter is an ALGOL EBCDIC array of one dimension, [1:27]. The initiating process could pass this parameter to a Pascal formal parameter that is a three-dimensional packed array of char. Suppose each dimension is declared with indexes [1..3]. The following table illustrates the mapping of elements from the ALGOL actual array into the Pascal formal array:

ALGOL Index	Pascal Index
1	1,1,1
2	1,1,2
3	1,1,3
4	1,2,1
5	1,2,2
6	1,2,3
7	1,3,1
8	1,3,2
9	1,3,3

The initiating process maps the remaining elements in a similar way.

Passing Parameters to Pascal Schemata

Before reading the rules for passing parameters to Pascal schemata, you should understand the following Pascal terms:

- Index

  An index specifies a location in a particular array dimension. If a dimension has indexes running from 1 to 5, then there are five indexes in that dimension.

- Discriminant

  A discriminant appears in an array declaration and specifies the highest-numbered or lowest-numbered index for a particular dimension. If the discriminant is an integer, it is called a constant discriminant. If the discriminant is a variable, it is called a dynamic discriminant.

- Element

  An element is a single location in an array. An element is identified by an index for each dimension stating the element's location in that dimension.

- Schema

  A schema is an array declaration that includes one or more dynamic discriminants. In other words, a schema is a type of incomplete array declaration. Using a schema as a formal parameter makes it possible to pass arrays with different bounds and different numbers of elements to the same formal parameter. The plural of *schema* is *schemata.*

When passing an array to a formal parameter that is a Pascal schema, the initiating process must pass one or more additional parameters. This is the only situation in which the system requires that the number of actual parameters be different from the number of formal parameters. The additional actual parameters provide information about the size of the actual array. Each of these additional parameters is a call-by-value integer.

The following are Pascal schemata types and the rules for passing parameters to each of these schemata types:

- A vlstring (variable-length string). This formal parameter receives two actual parameters: a parameter that contains the string value, followed by a call-by-value integer parameter that records the length of the string.
- A one-dimensional packed array of char whose upper discriminant is dynamic. This formal parameter receives the following two actual parameters: a one-dimensional array, followed by one call-by-value integer parameter that gives the value of the dynamic discriminant.
- Any other type of array or packed array whose declaration includes at least one dynamic discriminant. This type of formal parameter receives the following actual parameters, in the order listed:
  - A one-dimensional array of a compatible type.
  - For each dimension, a call-by-value integer parameter specifying the total number of elements in that dimension and all higher dimensions. For example, imagine an array with five indexes in the first dimension, three in the second dimension, and two in the third dimension. The first integer parameter is 30, which is the result of multiplying 5, 3, and 2 together. The second integer parameter is 6, which is the result of multiplying 3 and 2 together. The third integer parameter is 2.
  - For each dynamic discriminant, a call-by-value integer parameter giving the value of the discriminant. The order of the integer parameters is as follows: first-dimension lower discriminant, first-dimension upper discriminant, second-dimension lower discriminant, second-dimension upper discriminant, and so on. Any constant discriminants are omitted.

Examples

The following programs illustrate how an ALGOL program can pass an array to a Pascal two-dimensional packed array of char. The ALGOL program passes a one-dimensional EBCDIC array.

```
% ALGOL PROGRAM
BEGIN
   EBCDIC  ARRAY
      ALGOLARRAY[0:24];
   TASK T;

   PROCEDURE OUTSIDE(ACTUALARRAY);
      EBCDIC  ARRAY ACTUALARRAY[*];
   EXTERNAL;

REPLACE T.NAME BY "OBJECT/PASCAL/TWODIM/CHAR.";
REPLACE ALGOLARRAY[0] BY "ONETWOONETWOONETWOONETWO";
CALL OUTSIDE(ALGOLARRAY) [T];
END.

{ PASCAL PROGRAM }
program pascalarray((formalarray: formalarraytype));
   TYPE
      indexrange = 1..10;
      formalarraytype = packed array [2..5, 2..7] of char;
   VAR
      arrayindex, arrayindex2: indexrange;
BEGIN
   for arrayindex:= 2 to 5 do
      for arrayindex2:= 2 to 7 do
         formalarray[ arrayindex, arrayindex2 ]:= 'a';
END.
```

The following example shows what would happen if the formal parameter *formalarray* in the preceding example were changed from a fully specified array to a schema. Because of this change, the ALGOL program must pass additional call-by-value integer parameters.

```
% ALGOL PROGRAM
BEGIN
EBCDIC ARRAY
   ALGOLARRAY [0:24];
INTEGER
   ONEDIM, TWODIM, DISC1, DISC2;
TASK T;
PROCEDURE OUTSIDE (ACTUALARRAY, ONEDIM, TWODIM, DISC1, DISC2);
   VALUE ONEDIM, TWODIM, DISC1, DISC2;
   EBCDIC ARRAY ACTUALARRAY [*];
   INTEGER ONEDIM, TWODIM, DISC1, DISC2;
EXTERNAL;

REPLACE T.NAME BY "OBJECT/TASK/SCHEMA/PASCAL/TWODIM/CHAR.";
ONEDIM:= 24;
TWODIM:=  6;
DISC1:= 2;
DISC2:= 7;
REPLACE ALGOLARRAY [0] BY "ONETWOONETWOONETWOONETWO";
CALL OUTSIDE (ALGOLARRAY, ONEDIM, TWODIM, DISC1, DISC2) [T];
END.
```

```
{PASCAL PROGRAM }
program pascal_twodim_schema((formalschema: formalschematype));
   TYPE
      indexrange = 1..10;
      formalschematype(disc1, disc2:indexrange) =
                        packed array [disc1..5, 2..disc2] of char;
   VAR
      indexschema, indexschema2: indexrange;
BEGIN
   for indexschema:= formalschema.disc1 to 5 do
      for indexschema2:= 2 to formalschema.disc2 do
         formalschema[ indexschema, indexschema2 ]:= 'a';
END.
```

Passing COBOL74 Arrays to Bound Procedures

A COBOL74 host program can initiate a bound subprogram as a task with a PROCESS statement or with the *CALL <task identifier> WITH <section name>* form of the CALL statement. If the host program passes an array parameter to the task, the subprogram can receive various run-time errors (such as INVALID OPERATOR or SEG ARRAY ERROR) when it attempts to use the array. These errors can occur even if the arrays in the host and subprogram are the same type and length.

Specifically, arrays of usage BINARY, COMPUTATION, REAL, or DOUBLE always receive run-time errors when passed as parameters to a bound subprogram called as a task. EBCDIC arrays (01-level with usage DISPLAY) are the only type of array that can be passed successfully to such a subprogram. Nonarray items (77-level) can be passed without a problem.

If it is necessary for the bound subprogram to share a non-EBCDIC array with the host program, you can declare the array in the subprogram as a global array rather than a parameter. This method allows the same data to be shared between the subprogram and host, and does not cause run-time errors.

Section 18
Using Libraries

Introducing Libraries

A library is a collection of grouped objects that are shared with another process or processes. There are three types of libraries: *server libraries*, *client libraries*, and *connection libraries*. Unless otherwise specified, statements in this section about libraries apply to all types of libraries.

Server Libraries and Export Objects

A server library is said to *export* objects. Exporting objects enables other processes to use objects that were declared by the server library process. A process that creates a server library is referred to as a *server library process.*

Client Libraries and Import Objects

A client library is said to *import* objects. Importing objects enables the client library process to use objects that were declared by another process. A process that creates a client library is referred to as a *client process.*

Library Objects

Server libraries are usually used in combination with client libraries. The objects exported by a server library, in turn, are imported for use by a client library process. Objects that are exported by one library and imported by another are referred to as *library objects.*

Connection Libraries

Connection libraries enable a two-way exchange of objects between processes. When two connection library processes establish a connection, each library can export objects for use by the other library. Both library processes can execute in parallel. Compared to server libraries, connection libraries also provide greater ability to monitor and control linkages with other processes.

Server, Client, and Connection Library Combinations

It is possible for server library programs to export objects to connection libraries. It is also possible for client programs to import objects from connection libraries. However, this section describes library capabilities mostly in terms of unmixed usage: server libraries that are used with client programs, and connection libraries that are used with other connection libraries. The combinations involving mixed usage are discussed under "Linking Connection Libraries, Server Libraries, and Client Programs" later in this section.

Language Support

You can write server library programs in ALGOL, C, COBOL74, COBOL85, FORTRAN77, NEWP, and Pascal. You can write client programs in all of these languages, as well as in RPG. A library written in one language can be used by programs written in other languages.

You can write connection library programs only in ALGOL and NEWP.

Importing Process

This section makes some statements that apply both to client processes and to connection library processes that import objects. Because both these types of processes use imported objects, such a process is referred to by the term *importing process* in this section.

Procedure Objects

A procedure is the type of object most commonly exported by libraries. By consolidating procedures into a library, you can avoid duplicating the procedures in all the programs that need to use them. Further, you can maintain and enhance the shared procedures more easily when they reside in a library, because you don't have to repeat your work in every program that uses the procedures.

Data Objects

In addition to their role in providing shared procedures, libraries can also provide data structures to client processes. Thus,

- ALGOL libraries can export simple types of variables, events, event arrays, and other types of arrays.
- NEWP libraries can export events, event arrays, and other types of arrays.
- FORTRAN77 libraries can export files and arrays.
- Libraries in most other languages can provide client processes with indirect access to data objects that are declared in a library but not actually exported. The use of libraries to allow client processes to share data objects is discussed in "Global Objects in Server Libraries" and "Global Objects in Connection Libraries" later in this section.

Libraries Compared to Binding

Aside from libraries, the system provides several other methods by which programs can make use of a shared procedure, including binding, installation intrinsics, and separate programs. Compared to binding, libraries offer the following advantages:

- Libraries export objects at run time, whereas the Binder adds procedures from one object code file to another for permanent storage. You have to run the Binder separately for each object code file to which a procedure is to be added. You have to run the Binder again for each of these object code files whenever you make changes to the shared procedure.
- Libraries allow procedures to be shared between programs in a wider variety of languages than the Binder permits.

Libraries Compared to Installation Intrinsics

Compared to installation intrinsics, libraries offer the following advantages:

- Libraries can include objects that are declared globally to the exported procedures. These could include files, databases, and so on.
- Libraries can contain initialization and termination code.
- Individual users can create their own libraries without possessing special privileges.
- Libraries can be written in more languages than can installation intrinsics.
- More than one version of a library can be in use at a time.

Libraries Compared to Separate Programs

Another method for sharing procedures is to write each procedure as a separate program. Any other program that needs to make use of one of these procedures can initiate the appropriate program as a task. Compared to this method of sharing procedures, libraries offer the following advantages:

- The shared procedures can either be entered or initiated by the client program, whereas a separate program can only be initiated. Procedure entry takes less time and system resource than process initiation.
- There are more programming languages that provide the ability to use libraries than there are programming languages that provide the ability to initiate programs.

COBOL74 Restrictions

This section notes various restrictions on COBOL74 libraries that arise because this language does not permit nested blocks. Note that COBOL85 does permit nested blocks, and consequently provides more complete access to library features than COBOL74.

Creating Server Library Programs

Features of Server Library Programs

In most programming languages, server library programs can contain all the features of any ordinary program. What distinguishes a server library program is the inclusion of an export library, which is visible in most languages as an export list and a FREEZE statement. A server library program generally also includes features that specify the sharing and duration properties of the server library.

COBOL74 Exceptions

The following subsections outline the features required of server library programs in most languages, while also noting certain exceptions that apply to COBOL74 server libraries. The features of COBOL74 server libraries are most easily understood as a subset of the general server library features supported by the operating system.

Exporting Objects

Export Lists

A server library can contain many declarations of objects, some of which are exported and some of which are not exported. There is nothing in the declaration of an exported object that distinguishes it from a nonexported object. Instead, a separate construct called an *export list* specifies all the objects in a given block that are to be exported. The export list is used in addition to, rather than instead of, the declarations of the exported objects.

Exporting Objects in COBOL74

Export lists are not used in COBOL74. Libraries in this language always export exactly one object, which corresponds to the PROCEDURE DIVISION of the program.

Freezing the Library

FREEZE Statement

When a server library program is first initiated, it does not immediately become a library process. In most languages, a server library program is executed as an ordinary process until a FREEZE statement in the program is encountered. The FREEZE statement changes an ordinary process into a server library process. While the process is frozen, it typically does little or no work on its own; it simply remains present in memory so that client processes can link to it and use the exported objects.

Duration of the Freeze

Eventually, a server library process ceases to be a library and resumes execution as an ordinary process. The duration of the library state is specified by one of three FREEZE options. In most languages, you specify the FREEZE option in the FREEZE statement. The following are all the options supported by the system. Not all options are available in all languages.

- TEMPORARY

  The server library program ceases execution and remains available as long as clients of the library remain. A temporary library that is no longer in use resumes execution and ceases to be available for linkage. The export objects declared in the server library process do not become available to client programs again; attempts to link cause another instance of the server library to be initiated. Using the TEMPORARY option prevents memory space from being occupied by a server library that is not in use.

- PERMANENT

  The server library program ceases execution and remains available unless interrupted by an operator command or another program action. (Resuming a permanent library is discussed under “Thawing and Resuming Server Libraries” later in this section.) It is advisable to make a server library permanent if it is frequently used; the PERMANENT option prevents the system from having to re-create the server library repeatedly during the day. It is also advisable to make a server library permanent if it accesses a database or other files that need to be kept open.

- CONTROL

  The program is made available as a server library, and control passes to a local procedure in the library called the control procedure, where execution continues. The control library changes into a temporary library when the control procedure is exited.

  The CONTROL option makes it possible for a server library to decide when to resume itself. The CONTROL option is available in ALGOL and, implicitly, in FORTRAN77. The programmer typically includes statements in the control procedure to prevent it from being exited until certain conditions are met.

  In order to get the expected response, program the library to periodically check the values of its LIBRARYSTATE and LIBRARYUSERS task attributes. If the duration bit (bit 27) in the LIBRARYSTATE attribute is reset, and the value of LIBRARYUSERS is zero, the library is expected to exit the CONTROL procedure as soon as possible. Library linkages to this library are not initiated. The MCP linker waits for the library to resume normal execution before initiating a new instance of the server library.

  The library's EXCEPTIONEVENT is caused by the MCP upon the transition to a temporary duration and when the last library user delinks.

After a server library unfreezes, it cannot execute another FREEZE statement in order to become a library again.

Freezing COBOL74 Libraries

FREEZE statements are not used in COBOL74 libraries. Programs written in this language freeze automatically if they are initiated through the library linkage mechanism. (This method of initiation is discussed under "Initiating Server Library Processes" later in this section.) You can use the TEMPORARY compiler control option to specify that the library freeze should be temporary. If you do not use the TEMPORARY option, the library sharing option determines the type of freeze. A sharing value of SHAREDBYALL results in a permanent freeze, and sharing values of PRIVATE or SHAREDBYRUNUNIT result in a temporary freeze. For further information about the sharing option, refer to "Controlling Server Library Sharing" later in this section.

Freezing C Libraries

The FREEZE statement is also not used in C libraries. A C library freezes automatically if it is initiated by the library linkage mechanism. You can use the DURATION compiler option to specify whether it should be a temporary, permanent, or control freeze. If a temporary or permanent freeze is specified, the library freezes immediately after the function *main* finishes executing. If a control freeze is specified, the function *main* executes as the control procedure.

Displaying Frozen Libraries

You can use the LIBS (Library Task Entries) system command to list the frozen library processes on the system. The list includes permanent, temporary, and control libraries.

Controlling Server Library Sharing

SHARING Compiler Option

Although multiple client programs can use the same server library at the same time, they are not necessarily using the same instance of the library. You can use the compiler control option SHARING to specify whether multiple client processes access the same instance of the library. The possible values of this option are PRIVATE, SHAREDBYALL, SHAREDBYRUNUNIT, and DONTCARE.

SHARING = PRIVATE

The operating system initiates a separate instance of the server library program for each client process that links to the library. Values assigned to global objects in the library by a particular client process are visible only to that client process.

SHARING = SHAREDBYALL

All client processes share the same instance of the library. If one client process changes the value of a global object in the library, the next client process that interrogates the global object receives the changed value. The SHAREDBYALL option can be useful if the service provided by the library involves combining information from several clients or sharing resources among several clients.

The SHAREDBYALL option is not permitted in C or COBOL85.

SHARING = SHAREDBYRUNUNIT

The same instance of the server library is shared by each linkage that originates, either directly or indirectly, from the same client process. Multiple linkages can originate from the same client process in situations such as the following:

- A single client process contains multiple library declarations that link to the same library program.
- A client process imports one or more library objects provided indirectly. Though a client process might import objects from two separate libraries (LIB1 and LIB2), both of those libraries, in turn, might link to the same library (LIB3) to provide these objects. If LIB3 is SHAREDBYRUNUNIT, then the objects are imported from the same instance of library LIB3.

The SHAREDBYRUNUNIT option makes it possible for you to implement the ANSI COBOL concept of a *run unit*. A run unit consists of a client process and the libraries it calls, directly or indirectly. To meet the ANSI definition, if the client process links to the same library multiple times, the linkage must be to the same instance of that library.

However, the SHAREDBYRUNUNIT option, by itself, does not guarantee that a client process and its libraries will form a single run unit. To ensure a single run unit, you must use the following design rules:

1. The client must not use any library objects from a control procedure or from offspring processes.
2. The libraries used by the client, directly or indirectly, must all have a sharing option of SHAREDBYRUNUNIT.
3. The libraries must not link to any other libraries before or after freezing.

The following example illustrates what can happen if rule 3 is not followed:

- A client process CP1 implicitly links to a SHAREDBYRUNUNIT library named LIB1 by invoking a procedure in that library.
- The client process CP1 also implicitly links to a SHAREDBYRUNUNIT library named LIB2 by invoking procedure X in that library.
- Before LIB2 freezes, it links to LIB1. Because LIB2 is not yet frozen, LIB2 is not considered part of the run unit of CP1. Therefore, the system links LIB2 to a new instance of LIB1.

In this example, the client process gets access to a second instance of LIB1, which is not part of the ANSI run unit. If library LIB2 had frozen before linking to LIB1, then the linkage would have accessed the existing instance of LIB1, which is part of the run unit.

A run unit is not the same as a process family. For example, tasks initiated by the client process are not part of the run unit. Any other client processes linked to the library are also considered to be separate run units and receive their own instances of the library.

SHARING = DONTCARE

In most programming languages, this option is a nonpreferred synonym for SHAREDBYALL. However, in C and COBOL85, this option is a nonpreferred synonym for SHAREDBYRUNUNIT.

Default Values for SHARING

If a server library program does not include the SHARING compiler control option, then the compiler assigns a default SHARING option to the library. The default value of the SHARING option is SHAREDBYALL in ALGOL, and FORTRAN77; and SHAREDBYRUNUNIT in C, COBOL74, COBOL85, and Pascal. The SHAREDBYALL value is not available in C or COBOL85.

Sharing in COBOL74

Sharing is handled in a special way for COBOL74 libraries. If a library written in this language has a sharing value of SHAREDBYALL or SHAREDBYRUNUNIT, then multiple client processes can link to the same instance of the library. However, the operating system ensures that only one client process can be executing the procedure exported by this library at any given time. If another client process invokes the procedure while it is in use, the operating system causes this client process to wait until the procedure becomes available.

Shared Library Declarations

Note that the library sharing option affects only the relationship between library declarations and library instances. The sharing option cannot prevent multiple client processes from accessing the same server library instance through a common library declaration. For example, the outer block of an ALGOL program might include a library declaration. If this ALGOL program initiates two internal tasks, the library declaration is visible to both tasks. The two tasks could use this library declaration to access the same library instance, even if the library sharing option is PRIVATE.

Initiating Internal Server Library Processes

In ALGOL programs, an internal procedure can be initiated as a task and later freeze as a server library, if the procedure includes a FREEZE statement and an export list. NEWP programs marked with UNSAFE(TASKING) status also have this capability. However, to simplify this discussion, this section discusses server library processes as if they were always executions of an entire server library program.

Using Server Library Objects

Server libraries provide clients with access to data objects in either of two ways:

- By exporting procedures that read or update data objects in the server library program. These exported procedures provide indirect access to either local objects or global objects. Local objects are those declared within the exported procedures. Global objects are those declared globally to the exported procedures.
- By exporting the data objects themselves.

The following pages describe the use of local objects, global objects, and exported data objects.

Local Objects in Server Libraries

Local Objects Recreated

In most languages, any local variables declared in an exported procedure are recreated each time that procedure is invoked.

COBOL85 Exception

Variables declared in COBOL85 nested programs retain their values from one invocation to the next, unless the PROGRAM-ID paragraph of the nested program includes an *IS INITIAL* clause.

Global Objects in Server Libraries

Interprocess Communication

Shared server libraries provide one of the most sophisticated means for communicating information between processes. Any number of client processes can access the same object by way of a shared library. The client processes can belong to different process families, and can be written in different languages.

Capabilities

The main benefit of using a library for interprocess communication (IPC) is the flexible control it provides over the interactions between client processes and shared objects. For example,

- If a data item is being made available to many different client processes, the library can act to protect the data from being corrupted by a wrongly designed client process.
- The library can filter information, so that a particular piece of data in an object can be made visible to one client and not to others.
- A library can provide a simple interface to information that has a complex structure.

Scope of Global Library Objects

The key to using global objects for IPC lies in the addressing environment of an exported library procedure. Such a procedure can access any objects declared globally to it in the library. The rules for determining if a declaration is global to a given ALGOL procedure are discussed in Section 15, "Using Global Objects." Objects declared in COBOL74 libraries are global to the exported PROCEDURE DIVISION unless they are specified as parameters to the PROCEDURE DIVISION. For information about the scope of declarations in other languages, refer to the appropriate programming language manuals.

Parameters from Client Process

A library procedure can also access objects that are passed to it as parameters by a client process. These parameters can be used to inform the library procedure of changes that need to be made to a global object.

Typed Procedures

If the library procedure is a typed procedure, then you can use the return value to transfer information about the global object back to the client process. You can also use parameters that are passed to the procedure, by name or by reference, to transfer information back to the client process.

Accessing the Same Library Instance

For client processes to communicate through a server library, they must access the same instance of the library. You can ensure this by following these steps:

1. Set the SHARING option to SHAREDBYALL. This prevents a separate instance of the library from being initiated each time a new client process links to the library.
2. Use a permanent freeze in some cases. A library with a temporary freeze suffices for most cases, because it does not resume until all client processes have terminated. However, if there will be periods of time when no client processes are linked to the library, and the communication information needs to be preserved, then a permanent freeze must be used. This preserves the library instance until it is thawed by an operator command or a programmatic change to the STATUS task attribute.

Local Objects in Library Procedures

Note that objects declared within a library procedure cannot be used for IPC. Even if client processes access the same instance of a library, they receive different instances of any exported library procedure when they invoke that procedure. Changes made to these local objects by one client process are not visible to other client processes. OWN objects are an exception to this rule, and are discussed separately under "Restrictions on OWN Objects in Server Libraries" later in this section.

Ensuring One-at-a-Time Access to Objects

A library can be written to ensure that only one client process can access a particular global object, or group of global objects, at a time. For example, the library procedure that accesses a particular global object could be written to first procure a globally declared event, then access the global object, and then liberate the event. If all the library procedures that modify the global object are written this way, then the global object is protected from conflicting updates by different client processes.

Controlling the Order in Which Processes Access Objects

Events can also be used to ensure that client processes access a global object in a certain order. For example, assume that client process A is supposed to access a particular global object before client process B does. Client process B could invoke a library procedure that waits on a certain global event. This event might be one that is caused only at the end of the library procedure called by client process A.

Discontinued Processes and Shared Events

When designing a library, you must be aware of the fact that any of the client processes might be discontinued while executing a procedure from the library. If the library procedure being executed has procured an event, but has not yet liberated the event, then the event remains procured. Other client processes waiting to procure the event wait indefinitely. You can prevent this problem in either of the following ways:

- Include a CHANGE procedure in the server library process. Each time a client process delinks, the system passes a parameter to the CHANGE procedure indicating whether the delinkage is caused by the client process terminating abnormally. The server library could respond to this notification by, for example, liberating all global events whenever any client terminates abnormally. For information about the CHANGE procedure, refer to "Monitoring Client Process Linkage" later in this section.
- Use features such as EPILOG and EXCEPTION procedures to ensure that clients can perform cleanup actions during an abnormal termination. For further information, refer to "Discontinued Processes and Events" in Section 16, "Using Events and Interlocks."

MYSELF Task Variable

A library procedure can use the MYSELF task variable to access the task attributes of the client process. (In a procedure exported by a frozen library, MYSELF always refers to the client process, not the library.) For example, the library procedure could interrogate the USERCODE task attribute of the client process. The library procedure could be defined to provide different actions for different client processes.

Protecting Against Data Loss

Another point to be aware of is that the information stored in a permanent library can be lost if a system halt/load terminates the library process. The library can protect against this possibility by writing data out to disk files or to a database.

Shared Access Examples

For a simple example of a server library that provides client processes with shared access to a disk file, refer to "File Sharing Examples" in Section 19, "Using Shared Files."

Exported Data in Server Libraries

Language Restrictions

The only languages that can export data are ALGOL, NEWP, and FORTRAN77.

Currently, NEWP exports a more limited range of data types than ALGOL. For information about the types of data that can be exported in ALGOL and NEWP, refer to "Matching Data Types" later in this section.

This section does not discuss the exporting of data in FORTRAN77. Instead, refer to the *FORTRAN77 Programming Reference Manual*.

Comparison with Global Objects

ALGOL and NEWP server libraries can export certain types of data objects in much the same way as procedures. By exporting data, you provide client processes with direct access to the data, rather than the indirect type of access described under "Global Objects in Server Libraries" in this section. The capabilities provided by these methods are compared in the following table:

Capability	Exporting Data	Exporting Procedures That Access Global Data
Ease of Implementation	The importing program is given direct access to the data.	You have to implement library procedures that provide an interface to the data.
Maintainability	Changes to data structures, semantics, and access rules can make it necessary to revise all client programs.	Changes to data structures, semantics, and access rules can often be hidden from client programs.
Processor Overhead	Low.	Somewhat higher, due to the cost of invoking a procedure.
Sharing	In the case of a SHAREDBYALL or SHAREDBYRUNUNIT library, all clients of the same library instance receive the same instance of the exported variable.	In the case of a SHAREDBYALL or SHAREDBYRUNUNIT library, all clients of the same library instance receive indirect access to the same instance of the global variable.
Writeability	The library can specify whether clients have read-write or read-only access to the variable.	The exported procedures can each contain logic to provide read-write or read-only access.
Mutual Exclusion	The library can export an event that client processes are expected to procure before updating some particular object. The library cannot enforce the use of this event.	The exported procedures can be coded to always procure a global event before updating the global variable. In this way, the library can ensure that the event is used.
Client Identity	The library provides the same access rights to all clients. However, individual clients can reduce their own access rights by requesting read-only access to data that was exported with read-write access.	The exported procedures can include code to interrogate the client's identity, and allow different actions for different clients.

Use by Client Programs or Connection Libraries

The importing of data by client programs is supported only in ALGOL. Therefore, if a server library exports data, that data can be imported only by a connection library or an ALGOL client program.

Specifying the Access Mode

The server library can specify an access mode of READONLY or READWRITE for most exported data objects. If the server library does not specify an access mode, the default is READONLY.

In ALGOL, if all of the objects exported in an EXPORT declaration have the same access mode, you can specify the access mode within square brackets just after the EXPORT keyword. If the objects in the export list have different access modes, the access modes can be specified individually after each export object in the list.

You cannot specify an access mode for events or event arrays. However, access to events is automatically restricted in some ways, as described in the following paragraphs.

Limitations on Events

A server library program can use exported events in the same ways as any other event. However, a number of restrictions apply to the use of imported events by the importing program. The usage of imported events is restricted in the following ways:

Usage	Restriction
WAIT and WAITANDRESET Statements	An imported event cannot be used in WAIT or WAITANDRESET statements. However, imported events can be used in other event-related statements and functions, such as AVAILABLE, CAUSE, CAUSEANDRESET, FIX, FREE, HAPPENED, LIBERATE, PROCURE, RESET, and SET. Furthermore, a server library and client program could communicate by having the server library apply WAIT or WAITANDRESET statements to the exported event, and having the client library apply CAUSE or CAUSEANDRESET statements to the imported event.
Interrupts	An imported event cannot be attached to an interrupt.
LOCK Statements	An imported event cannot be used in statements of the form LOCK (<interlock>, <event>).
Direct I/O	An imported event cannot be used in direct I/O.
Parameters	An imported event cannot be passed as a parameter to a procedure.

If a program violates one of these restrictions, a syntax error results.

Syntax Examples for Exporting Data

The following ALGOL statements export several data objects. Real array A receives the default access mode of read-only, and integer I is assigned an access mode of read-write.

```
REAL ARRAY A [0:10];
INTEGER I;
EVENT E1;
EXPORT A (LINKCLASS = PROTECTED),
       I AS "I2" (READWRITE),
       E1;
```

Alternatively, the EXPORT statement can include a single access mode assignment that affects all the items in the export list, as in the following example:

```
EXPORT [READWRITE] A, I;
```

Direct, Indirect, and Dynamic Data Provision

Server libraries must use direct provision when exporting data objects; indirect and dynamic provision are not permitted. For definitions of these various types of provision, refer to “Methods of Providing Objects” later in this section.

Restrictions on OWN Objects in Server Libraries

Purpose of OWN Clause

You must take a special precaution when declaring arrays with an OWN clause in an exported library procedure. The OWN clause, which is available only in ALGOL, causes the value of an object to be saved between invocations of the procedure in which that object is declared. If multiple invocations of the procedure are running simultaneously, the OWN clause also causes all invocations of the procedure to access the same instance of the object.

OWN Array Restrictions

If an exported procedure includes an array declaration with an OWN clause, then the server library program should itself invoke the exported procedure before any client process invokes the procedure. If a client process invokes the procedure before the server library does, then the system discontinues the client process with the error "ILLEGAL OWN ARRAY." The server library process itself is not affected by this error.

OWN Simple Variables

Note that this restriction applies only to OWN arrays, not to simple variables or pointers with an OWN clause. For exported procedures that declare such variables, it does not matter whether the client process or the server library process invokes the procedure first.

Timing Issues in Shared Libraries

If the library is a shared library, then synchronization issues arise for any OWN objects declared in an exported procedure. The OWN clause allows multiple client processes to access the same instance of the same object. For example, if two client processes are concurrently executing the same library procedure, and the library procedure declares an OWN object, then any changes made by one client process to the value of the object are immediately visible to the other client process. To prevent timing ambiguities, you can use techniques such as those discussed under "Global Objects in Server Libraries" later in this section.

Restrictions on COBOL74 Libraries

Dual Nature: Server Libraries and Ordinary Programs

COBOL74 object code files are structured in a special way that allows them to be executed either as server libraries or as ordinary processes. Every program written in this language is available for use as a server library, except for programs that

- Specify in the data division that a parameter is RECEIVED BY CONTENT (that is, received as a call-by-value parameter).
- Specify parameters in the USING clause of the PROCEDURE division that are not allowed for libraries. Each data item in the USING phrase must be defined as level 01 or level 77, and the data item must not redefine another data item. Further, the parameters must be of data types that are allowed for library parameters. For a list of the allowed library parameter types for COBOL74, refer to Table 18–4, "COBOL74 Parameters."
- Are compiled with a LEVEL compiler control option that specifies a lexical level greater than 2.

Library-Capable Programs

A program that does not use any of these restricted features is said to be *library-capable*. A library-capable COBOL74 program freezes if it is initiated through the library linkage mechanism. If the program is initiated by a process-initiation statement, then the program runs as an ordinary process and does not freeze. For a discussion of the library linkage mechanism, refer to "Initiating Server Library Processes" later in this section.

Circular Linkage Disallowed

Circular library linkages are not allowed for COBOL74 libraries. If a COBOL74 library invokes itself, the operating system discontinues the library with a run-time error. If a COBOL74 library invokes a procedure in another library that in turn invokes the original library, both libraries freeze successfully but the client process hangs indefinitely in the state WAITING ON AN EVENT.

EXIT PROGRAM Versus STOP RUN

An EXIT PROGRAM statement must be used to exit a COBOL74 library and to return to the calling program. By contrast, a STOP RUN statement causes the client process to terminate at the point of the statement that invoked the library procedure.

Sharing in COBOL74

The system prevents multiple clients of a shared COBOL74 library from executing the exported procedure at the same time. Refer to "Controlling Server Library Sharing" earlier in this section.

Global Nature of Declarations

Because only the PROCEDURE DIVISION of a COBOL74 library is exported, most objects declared in the DATA DIVISION or ENVIRONMENT DIVISION are considered as global to the exported procedure. Within each library instance, these objects retain their values from one procedure call to the next. In a shared library, any changes made to the values of these objects by a client process are visible to all other client processes.

Objects in USING Clause

One exception to this behavior occurs for objects that are referred to in the USING clause of the PROCEDURE DIVISION. This clause causes the objects to be treated as parameters to the PROCEDURE DIVISION. Each client process receives a separate instance of these objects, and the values of the objects are not retained from one procedure call to the next.

No Initiation or Termination Code

Another limitation arises from the fact that the entire PROCEDURE DIVISION is exited. There is no place in a COBOL74 library to specify initialization or termination code. By contrast, the outer block of an ALGOL library can contain statements that execute before the FREEZE statement or that execute after the library unfreezes.

Monitoring Server Library Linkage

Server libraries can optionally include a procedure that notifies the library of changes in linkage state. For more information about this procedure, called the CHANGE procedure, refer to “Monitoring Client Process Linkage” later in this section.

Thawing and Resuming Server Libraries

Thawing a server library is the act of changing the frozen library process from a permanent library into a temporary library. By contrast, the act of *resuming* a library causes the server library to be deactivated, and causes the server library process to resume execution as an ordinary process. Execution of the process resumes with the first statement after the FREEZE statement.

STATUS Attribute and THAW Command

You can thaw a server library process programmatically through assignments to the STATUS task attribute of a process, or operationally through the THAW (Thaw Frozen Library) system command. For details, refer to Section 6, “Monitoring and Controlling Process Status.”

CANCEL Statement

Additionally, you can resume a server library process programmatically with the CANCEL statement in ALGOL, as discussed under “Delinking from Server Libraries” later in this section.

Creating Client Programs

Features of Client Programs

In general, client programs can include all of the features of an ordinary program. Further, a client program can itself be a server library program that uses objects imported from other libraries.

Import Declarations and Client Library Declarations

In most programming languages, client programs are distinguished by the inclusion of import declarations and client library declarations. These declarations, and their equivalents in COBOL74, are described in the following subsections.

Importing Procedures to Client Programs

Import Declarations

Import declarations include information such as the name of the imported procedure, the type of procedure, and the library from which it is imported. Declarations of imported procedures must also include parameter specifications for any parameters accepted by the procedure. However, the procedure body (including all local declarations and statements) is omitted. For example, in ALGOL an imported procedure might be declared as follows:

```
BOOLEAN PROCEDURE X (I);
   INTEGER I;
   LIBRARY LIB1;
```

Use of Imported Objects

In languages that include library declarations and import declarations, the client program can use imported objects just as if they were local objects of the client program.

COBOL74 Imports

In COBOL74, import objects are not explicitly declared. Instead, when invoking an imported procedure, you can use a special form of the CALL statement that specifies both the name of the procedure and that of the library. If the CALL statement does not explicitly specify a procedure name, the procedure name is assumed as PROCEDUREDIVISION.

Objects That Are Not Imported

In addition to import declarations, the client program can contain declarations of objects that are not imported from libraries.

Importing Data to Client Programs

Server libraries written in ALGOL or NEWP can export data objects, as discussed under "Exported Data in Server Libraries" earlier in this section. Client programs written in ALGOL can import these data objects by listing them in the LIBRARY declaration. (NEWP does not yet support the importing of data objects by client programs.)

Syntax for Importing Data

For example, the following ALGOL library declaration specifies several imported data items (M, N, G, and E):

```
LIBRARY L(LIBACCESS = BYFUNCTION, FUNCTIONNAME = "MYSUPPORT")
   [ INTEGER M (READWRITE),
             N (READONLY);
     ARRAY G[0] (ACTUALNAME = "GG"),
           E[0]  ];
```

Matching Access Modes

The client program can specify whether each imported data item is read-only or read-write. The default access mode is read-only.

The server library also specifies the access mode for exported data. The following table shows the actual access modes resulting from the possible combinations of values:

Exported Access Mode	**Imported Access Mode**	**Resulting Access Mode**
READONLY or unspecified	READONLY or unspecified	READONLY
READONLY or unspecified	READWRITE	Objects cannot match.
READWRITE	READONLY or unspecified	READONLY
READWRITE	READWRITE	READWRITE

If the import and export objects do not match due to conflicting access modes, library linkage can still complete successfully. The LINKLIBRARY result indicates that some requested objects were not available. If the client process attempts to use the imported data object, the client process is discontinued with the error message "OBJECT <object name> ACCESS MODE MISMATCH." The client process can prevent this error by checking the availability of the data object with the ISVALID function as described under "Matching Client and Server Library Objects" later in this section.

Data Type Matching

The exported data object and the imported data objects must be of the same or compatible types. Refer to "Matching Data Types" later in this section.

Limitations on Events

If a client library imports an EVENT or EVENT ARRAY, the client library must explicitly specify an access mode of READWRITE, or a syntax error results.

Exported events can be used in the same ways as any other event. Refer to the topic "Limitations on Events" in "Exported Data in Server Libraries" discussed earlier in this section.

If a program violates one of these restrictions, a syntax error results.

Specifying Client Libraries

Client Library Declarations

In most languages, the client program must include explicit declarations of all client libraries and library objects used by the program. A client library declaration specifies the identifier by which the library is known throughout the client program. A client library declaration can also include library attribute assignments.

Library Attributes

Library attributes should not be confused with task attributes or file attributes. Library attributes provide information that help monitor and control libraries.

Changes to Library Attributes

The client process can change library attribute values repeatedly, if the client library is not currently linked to a server library. The operating system ignores any changes made to the attributes of a client library while the client library is linked to the server library. The exception to this rule is the AUTOLINK attribute, which can be changed while the client library is linked to the server library.

COBOL74 Client Libraries

In COBOL74, client programs do not include library declarations. However, this language does allow you to assign library attributes to a client library.

CHANGE Attribute Assignment

Unlike most library attributes, the CHANGE attribute can be assigned in the client library program, the server library program, or both.

The following paragraphs briefly explain the use of library attributes related to client libraries.

LIBACCESS, TITLE, and FUNCTIONNAME Library Attributes

A client process uses the LIBACCESS attribute of the client library to specify how to select the server library. The possible values are BYTITLE, BYFUNCTION, and BYINITIATOR.

If LIBACCESS = BYTITLE, then the server library program is selected by its title. The client process assigns the library code file title to the TITLE attribute of the client library. No value is assigned to the FUNCTIONNAME library attribute.

If LIBACCESS = BYFUNCTION, then the server library program is selected by its function name, which is defined and mapped to a library code file beforehand by the SL (Support Library) system command. The client process assigns the desired function name to the FUNCTIONNAME attribute of the client library. No value is assigned to the TITLE library attribute.

If LIBACCESS = BYINITIATOR, then the client process must be either a task initiated by a library, or a library initiated by the library linkage mechanism after another library invokes it. In either case, BYINITIATOR causes the client process to link to the library that originated the client process. No value is assigned to the FUNCTIONNAME or TITLE attributes of the client library.

LIBPARAMETER Library Attribute

The client process can use the LIBPARAMETER library attribute for cases where the server library provides objects dynamically. The LIBPARAMETER value can include application-defined information to help the server library decide to which secondary library to link. For more information, refer to "Dynamic Provision" later in this section.

If the client library is linking to a connection library instead of to a server library, the client process can use the LIBPARAMETER attribute to pass information to the APPROVAL procedure. Refer to description of the APPROVAL library attribute under "APPROVAL" later in this section.

AUTOLINK Library Attribute

The client process can use the AUTOLINK library attribute to specify whether this client library can be linked implicitly to the server library. For more information, refer to "Linking to Server Libraries."

INTERFACENAME Library Attribute

If the client library links to a connection library instead of a server library, the client process assigns an INTERFACENAME attribute to the client library. Refer to "Linking a Client Library to a Connection Library," later in this section.

For detailed descriptions of these and other library attributes, refer to "Using Library Attributes" later in this section.

Linking to Server Libraries

Types of Linkages

Linkage between a client library and a server library is established at run time by the operating system, based on the values of the library attributes of the client library. There are three ways a client library can link to a server library: implicitly, explicitly, or directly.

Linkage Errors

Linkage errors have different effects, depending on the type of linkage that is used. If an implicit linkage results in an error, that error is fatal for the client program. On the other hand, if an explicit or direct linkage results in an error, the error is nonfatal and is communicated to the client program as a LINKLIBRARY function result.

Implicit Initiations

If a process attempts to link to a server library, and no frozen instance of that library is currently available, the system performs an implicit initiation of the library. This implicit initiation can occur regardless of whether the linkage attempt was an implicit linkage, an explicit linkage, or a direct linkage. Refer to "Implicitly Initiating a Server Library" later in this section.

Implicitly Linking to Server Libraries

Initiating Implicit Linkage

Library linkage typically occurs when the client process first uses an object imported through a particular client library. For example, suppose a client process has import declarations for three procedures, PROC1, PROC2, and PROC3, which all come from client library LIB1. Note that the order in which these imports are declared might not be the same as the order in which they are invoked. Thus, PROC1 might be declared first, but the client process might invoke PROC3 first. In this case, the linkage of client library LIB1 to a server library is implicitly requested by the attempt to invoke PROC3.

Allowing or Preventing Implicit Linkages

The AUTOLINK library attribute determines whether such implicit linkage attempts are allowed. If AUTOLINK is FALSE, the linkage fails and the client process incurs a fatal error. If AUTOLINK is TRUE (which is the default for client libraries), then the system proceeds with an attempt to link the client library to the server library.

Missing Server Library

If the requested server library program cannot be found, the client process becomes suspended at this point. If the LIBACCESS value is BYTITLE, then a "NO LIBRARY" message is displayed as the RSVP message. If the LIBACCESS value is BYFUNCTION, and the FUNCTIONNAME library attribute requests a function name that does not exist, the RSVP message is "FUNCTION <function name> IS NOT DEFINED, SL, FA, OR DS."

Missing Object in Library

If the requested server library program is found, the system links the client library to an existing instance of the server library or initiates a new instance as discussed under "Implicitly Initiating a Server Library" later in this section. The system then determines if the requested object is exported by the server library. If the requested object is not available, the system returns one of the errors discussed under "Matching Client and Server Library Objects" later in this section.

Missing Objects and Indirect or Dynamic Provision

If the requested object is provided indirectly or dynamically, then linkage can succeed even if the object is not available for use. This behavior is possible because indirect and dynamic linkage involve one or more intermediate libraries— aside from the library that ultimately provides the object. The requested object might be exported by an intermediate library, and still not be available from the library that is ultimately supposed to provide the object.

For example, suppose client library CL1 imports procedure PROC1 from server library SL1. Server library SL1 provides PROC1 indirectly by importing it from server library SL2. Now suppose that server library SL2 is unable to freeze or does not export a procedure called PROC1. In this case, the system successfully links CL1 to SL1. However, an error occurs if CL1 attempts to invoke PROC1. CL1 could prevent this error by using the ISVALID function before invoking PROC1; refer to "Matching Client and Server Library Procedures" later in this section.

Explicitly Linking to Server Libraries

LINKLIBRARY Function

Because of the possibilities for fatal errors in linking client libraries to server libraries, you might want to consider using the LINKLIBRARY function to make the linkage. This function, which is available only in ALGOL, NEWP, and Pascal, makes a conditional attempt to establish linkage with the server library. If the attempt fails, the function returns a value indicating the reason for the failure, but no error messages are displayed. If the attempt succeeds, the function returns a value indicating whether all the import objects defined in the client program for the client library were really present in the server library.

LINKLIBRARY Parameters

The LINKLIBRARY function can include the following parameters:

- Substitute values for any of the following library attributes: FUNCTIONNAME, INTERFACENAME, and TITLE. These values temporarily override any values previously assigned to these attributes of the client library.
- An option indicating the action to take if the library is not available. The following are the possible values of this option and their effects:
  - DONTWAIT

    The linkage attempt proceeds only if a library instance is immediately available (that is, already frozen or readied by the READYCL function). Otherwise, LINKLIBRARY returns an error result.
  - DONTWAITFORFILE

    If a library instance is not available, the system initiates a new instance of the library code file. If the library code file is also not available, LINKLIBRARY returns an error result.
  - WAITFORFILE

    The linkage attempt continues regardless of whether a library instance or library code file is available. If a library instance is not available, the system initiates a new instance of the library code file. If the library code file is also not available, the linking process is suspended with an RSVP message and waits for an operator response.

LINKLIBRARY Result

LINKLIBRARY returns a result indicating whether the linkage attempt was successful or not. The possible LINKLIBRARY results are documented in the *ALGOL Programming Reference Manual, Volume 1: Basic Implementation.*

Directly Linking to Server Libraries

Linking a Client Library to a Task Variable

If a process has access to a declaration of a client library and the task variable of a server library, the process can link the client and server libraries together by specifying the library declaration and the task variable in the LINKLIBRARY function. This type of linkage is referred to as *direct linkage,* because it bypasses the mechanisms that the system otherwise would use to locate the matching library. When you use direct linkage, the system ignores the values of the library attributes FUNCTIONNAME, INTERFACENAME, LIBACCESS, and TITLE.

LINKLIBRARY Syntax

The direct form of the LINKLIBRARY function is available only in ALGOL and NEWP. The following is an example of such a LINKLIBRARY call:

```
RSLT := LINKLIBRARY(CLIB1, LIBRARY = TVAR);
```

In this example, CLIB1 is the client library identifier and TVAR is the task variable of the server library process. The STATUS of TVAR must be FROZEN for linkage to succeed.

Limits on LINKLIBRARY Options

The direct form of the LINKLIBRARY function cannot include a DONTWAIT/ DONTWAITFORFILE/WAITFORFILE option or any library attribute values.

Previously Specified Library Attributes

Other library attributes such as LIBPARAMETER have the same effects that they do for implicit or explicit linkages.

Matching Client and Server Library Objects

Matching Import and Export Objects

During the linkage process, the system attempts to establish matches between objects imported by the client library and objects exported by the server library. The rules governing this matching are discussed under “Methods of Providing Objects” and “Type Matching”.

Missing Objects Might Not Prevent Linkage

Library linkage can proceed successfully even if the system does not find matches for all the import objects. However, the following minimum matches must succeed in order for linkage to take place:

- If any objects are imported, then at least one object must match for linkage to succeed.
- If the linkage was initiated implicitly, by accessing an import object in an unlinked library, then a match must be found for that particular import object.

Error When Missing Object Is Used

The system returns an error for a missing object when that object is first used. (In the case of implicit linkage, these errors can occur at linkage time if they apply to the object whose usage caused the implicit linkage.)

The following are the errors that the system can return:

- MISSING OBJECT <object name> IN LIBRARY <library name>

  No object with the requested name is exported by the server library. The system discontinues the client process.
- OBJECT <object name> LINKAGE CLASS VIOLATION IN LIBRARY <library name>

  The client process lacks the linkage class necessary to use the exported object. The system discontinues the client process.
- Object <procedure name>: Type or parameter mismatch in interface <client library identifier> to library <library name>

  An export object with the requested name exists, but the type or parameters of the import and export objects do not match. The system discontinues the client process. For an explanation of the rules used to match object types and parameters, refer to "Type Matching" later in this section.
- OBJECT <object name> ACCESS MODE MISMATCH

  A client process attempted to import a data object as read-write when the object was exported as read-only. The system discontinues the client process.

Detecting Whether Objects Are Missing

If the LINKLIBRARY function is used to establish the linkage, the return value indicates whether there were any problems in finding object matches.

Checking the Availability of an Object

To prevent the possibility of a fatal error when using an imported object, the client program can use the ISVALID function in ALGOL or NEWP. The ISVALID function returns a Boolean result indicating whether a given object is valid. For example, the following statement invokes procedure PROC1 only if that procedure is valid:

```
IF ISVALID(PROC1) THEN PROC1;
```

For imported data, the ISVALID function can check whether the data is available for reading or for reading and writing. The following statement reads integer J if it is available, regardless of whether the access mode is read-only or read-write:

```
IF ISVALID(J) THEN I := J;
```

The following statement writes to integer J if it is available and the access mode is read-write.

```
IF ISVALID(J, READWRITE) THEN J := 43;
```

Initiating Server Library Processes

Server library processes can be initiated in either of two ways: implicitly, through the library linkage mechanism, or explicitly, by a process-initiation statement in another process.

Implicitly Initiating a Server Library

Initiations Caused by Linkage Attempts

If a client process attempts to link to a server library, then in some cases the library linkage mechanism attempts to initiate a new instance of the server library. This initiation is referred to as an *implicit initiation*.

The linkage attempt that causes the implicit initiation could be an explicit linkage, an implicit linkage, or a direct linkage.

Task Attribute Inheritance

The library process inherits task attributes from the client process if the library is initiated implicitly and the linkage mode is BYTITLE. The library process inherits the same task attributes that an ordinary task inherits from its initiator.

If the linkage mode is BYFUNCTION or BYINITIATOR, the library process does not inherit any attributes from the client process.

Waiting for the Library to Freeze

While the system is performing an implicit initiation, the client process enters a waiting state. The STATUS task attribute is still ACTIVE, but the stack state in the Y (Status Interrogate) system command is WAITING ON AN EVENT. The library linkage mechanism of the operating system automatically initiates the server library program, which executes normally until it freezes and becomes a server library process. At this point, the system completes the linkage between the client library and the server library, and the client process resumes execution.

If you do not wish to expose a client process to the potential delay involved in an implicit initiation, then you should use explicit linkage with the LINKLIBRARY function, and specify the DONTWAIT option. If a frozen instance of the library is not available, the LINKLIBRARY function returns with an error result. The following is an example of such a function:

```
RSLT:= LINKLIBRARY (LIB1, DONTWAIT);
```

Libraries with Limited Sharing

Even if an instance of the server library is already frozen, the system might initiate a new instance of the server library for the client library to link to. For example, if the sharing option of the server library is PRIVATE, then the system initiates a new instance of the library for each client library. If the sharing option is SHAREDBYRUNUNIT, then the system initiates a new instance of the server library program for each run unit that uses the server library. (Refer to "Controlling Server Library Sharing" for more information.)

Limited Sharing Causes Limited Library Duration

If a PRIVATE or SHAREDBYRUNUNIT server library is initiated through the library linkage mechanism and the library requests a permanent freeze, the system actually freezes the library as a temporary library. The system does this because a PRIVATE or SHAREDBYRUNUNIT library instance can only be linked to by a client library once. Thus, no purpose would be served by allowing the library instance to linger after the original client delinks.

Programs That Cannot Become Libraries

If an attempt is made to link a client library to a program that is not capable of becoming a library, then the library linkage mechanism issues the error "LIBRARY WAS NOT INITIATED: <library name>" and discontinues the client process. This error can happen, for example, if the client process attempts to link to an ALGOL program that does not contain a FREEZE statement.

Waiting for a Delayed Freeze

Because a server library program initially runs as a regular program, the flow of execution can be such that the execution of a FREEZE statement is conditional and can occur anywhere in the program. If the library linkage mechanism initiates such a program, and the client process waits longer than one minute for the library to freeze, the system displays the following message for the user process:

```
Waiting For Library (Mix <mix number>) to Freeze
```

If you miss seeing the original display of this message, you can use the Y (Status Interrogate) system command. The message appears in the Display line of the Y command output.

Libraries That Never Freeze

If the library process terminates without ever having executed a FREEZE statement, the system returns an error condition. For implicit linkage attempts, this condition causes the client process to be discontinued and displays the following message:

```
LIBRARY DID NOT FREEZE:  <library name>
```

For explicit linkage attempts, the LINKLIBRARY function returns an error indication, and the client process continues execution.

Explicitly Initiating a Server Library

Synchronous Tasks Cannot Freeze

A server library process can be explicitly initiated by a process initiation statement in a program. However, the resulting process can freeze only if it is an independent process or an asynchronous dependent process. If the program is initiated as a synchronous dependent process, by a WFL *RUN* statement, for example, then when the process attempts to freeze, it is discontinued with the error "FREEZE FAILED, TASK TYPE NOT PROCESS OR RUN."

Freeze Duration

When a server library process is meant to be explicitly initiated, the library should typically specify a freeze duration of PERMANENT or CONTROL. If the freeze is TEMPORARY, then the process can freeze successfully only if the process is an internal process initiated by a PROCESS statement.

Library Sharing

For server libraries that are explicitly initiated, some special considerations apply to the sharing option. If the sharing option is PRIVATE or SHAREDBYRUNUNIT, then the server library instance is not directly available to client processes. However, such a server library instance can serve as a secondary library in a dynamic provision mechanism. (This type of linkage is discussed under "Dynamic Provision" later in this section.) If the sharing option is SHAREDBYALL or DONTCARE, then any client process can link to that server library instance.

Determining How a Library Was Initiated

A server library program can determine whether it was initiated explicitly or by the library linkage mechanism by interrogating the LIBRARYSTATE task attribute. Bit [0:1] of the LIBRARYSTATE value stores a 1 if the process was initiated by the library linkage mechanism. For example, an ALGOL program can use the following expression to determine whether it was initiated by the library linkage mechanism:

```
IF BOOLEAN(MYSELF.LIBRARYSTATE) THEN...
```

Monitoring Client Process Linkage

CHANGE Procedures for Server or Client Libraries

Server libraries and client libraries can each specify CHANGE procedures that monitor changes in the library linkage state. You can specify a CHANGE procedure for only the client library, for only the server library, for both the client library and the server library, or for neither.

Declaring a CHANGE Procedure

To use a CHANGE procedure in a server library, you must declare a procedure in the library that accepts a standard set of parameters from the operating system. You must also include an EXPORTLIBRARY statement that includes an assignment to the CHANGE library attribute. The CHANGE attribute specifies the name of the procedure to be used as the CHANGE procedure. For example, the following ALGOL statement in a server library specifies that the library is using a CHANGE procedure called CHG1:

```
  EXPORTLIBRARY (CHANGE = CHG1);
```

To use a CHANGE procedure in a client program, you must declare a procedure with the required parameters in the client program. You must also assign the procedure name to the CHANGE attribute in the client library declaration.

When the CHANGE Procedure Is Invoked

The system invokes the CHANGE procedure for the server library whenever any client process completes linking to the server library or begins delinking from the server library. The system invokes the CHANGE procedure for the client library whenever that client process completes linking to or begins delinking from the server library.

Parameters to the CHANGE Procedure

The system passes parameters to each invocation of the CHANGE procedure including the new linkage state, the reason for the change in linkage state, and an indication of whether the client process is terminating abnormally. The CHANGE procedure can read the parameter values but does not return any value itself.

Modifying Global Objects

The CHANGE procedure can modify globally-declared objects such as events in the program that declared the CHANGE procedure (either the server library or the client process). These modifications can be used to inform the server library process or client process of the change in linkage state.

Control Libraries and CHANGE Procedures

If a server library has performed a control freeze, the control procedure can include statements to monitor and respond to any changes made to global objects by the CHANGE procedure. If the server library has a permanent or temporary freeze, the server library cannot directly detect any changes to global objects made by the CHANGE procedure. However, before the server library performs a permanent or temporary freeze, the server library could initiate an asynchronous task to monitor changes to global objects in the library.

Keeping Track of Client Processes

The CHANGE procedure does not provide any indication to a server library process of which client process has just linked or delinked. However, the server library process could keep a running count of the number of linkages and delinkages, and so know how many client processes are linked at a given time.

Note: *A server library* cannot *determine its current number of clients by reading the LIBRARYUSERS task attribute inside the CHANGE procedure, because the MYSELF task variable in a CHANGE procedure does not refer to the server library process.*

For a description of the CHANGE procedure, refer to the discussion of the CHANGE library attribute under "Using Library Attributes" later in this section.

Delinking from Server Libraries

Delinking Libraries Early

The system automatically delinks a client library from a server library when the client process exits the block in which the client library is declared. However, at times it can be useful to delink a client library from a server library at an earlier point. For example, delinking a client process from a server library enables the client process to modify one or more of the client library attributes. Delinking a client process also allows a temporary library process to unfreeze and resume execution. With the exception of the AUTOLINK attribute, none of the attributes of a client library can be modified while the client library is linked to the server library.

Delinking a library is a very costly operation. For more information, refer to "Overuse of Delinkage."

Two features you can use to explicitly delink a client library from a server library are the DELINKLIBRARY function and the CANCEL statement.

Delinking without Affecting Other Users

The DELINKLIBRARY function delinks the client library from the server library without affecting any of the other processes using the server library. The server library remains frozen, unless it is a temporary library and the delinked process was the only process using the library. The DELINKLIBRARY function is available in ALGOL, NEWP, and Pascal.

Canceling and Unfreezing a Library

The CANCEL statement also delinks the client library from the server library, but has the following additional effects:

- Delinks any other clients that are currently linked to the library, and discontinues any of those clients that are doing any of the following:
  - Executing a procedure imported from that library.
  - Waiting to procure an event imported from that library.
  - Using certain UNSAFE features of DMALGOL or NEWP to access objects in the library.
- Causes the server library process to unfreeze and resume execution as a regular process. This is true regardless of whether the server library has a permanent or temporary freeze.

The CANCEL statement is available in ALGOL, COBOL74, COBOL85, and Pascal.

Canceling SHAREDBYALL Libraries Not Allowed

Only server libraries with a sharing option of PRIVATE or SHAREDBYRUNUNIT can be canceled. An attempt to cancel a client library that is linked to a SHAREDBYALL server library results in the warning message "CANCEL WARNING, SHARED LIBRARY WAS DELINKED." In this case, the client library is delinked as if it had performed a DELINKLIBRARY function, and the server library process remains frozen unless it is a temporary library with no other clients.

Canceling SHAREDBYRUNUNIT Libraries

If a client process cancels a client library linked to a SHAREDBYRUNUNIT server library, then any other client processes in the same run unit that are currently linked to the server library lose their linkage. The next time one of these processes uses an object in the server library, the system initiates a new instance of the server library and links the client library to the new library instance.

Effect on Clients Sharing the Same Declaration

Note that internal tasks of a client process can link to a server library by way of a single client library declaration, declared globally in the client process. If such an internal task is using a library object when the parent client process executes a DELINKLIBRARY function or a CANCEL statement, the internal task is discontinued.

Creating Connection Library Programs

Connection libraries are a special type of library that can be implemented only in ALGOL or NEWP. The use of connection libraries provides the following features that are not available to server libraries:

- A two-way exchange of objects between libraries through a linkage called a *connection.*
- The ability of a single program to export more than one library simultaneously.
- The ability to specify multiple connections for a library, where the connections can lead to different libraries in different programs.
- The ability to monitor and control each connection with another library separately.
- The ability to declare objects that are local to each connection of the library.

When to Use Server Libraries Instead

Connection libraries are a good choice for projects that make use of these special features. However, if you do not need to use any of these features, you might find it easier to implement the project using server libraries.

Mixing Connection Libraries, Server Libraries, and Client Libraries

Linkages can exist between connection libraries, between a connection library and a server library, or between a connection library and a client program. However, to simplify the explanation, this section generally discusses connection libraries as if they could link only to other connection libraries. For an overview of linkages between connection libraries, server libraries, and client programs, refer to "Linking Connection Libraries, Server Libraries, and Client Programs".

Connection Library Performance Consideration

When libraries are used for a two-way exchange of objects, such as in the following cases, the system must execute expensive cleanup code when a library delinks:

- A connection library that both imports and exports objects
- Multiple connection libraries between two stacks, some of which export and some of which import objects
- Circular libraries

This expense can become extreme when many stacks are included in the process families. It is advisable to design applications to avoid a two-way exchange of objects between stacks whenever possible. For more information, refer to "Hazards of Circular Connections."

Declaring a Connection Library

Connection Blocks

To declare a connection library, you must first declare a *connection block.* The connection block is a type declaration that acts as a model for the definition of one or more connection libraries. The following ALGOL statements define a connection block called CL_TYPE and two different connection libraries that are based on this connection block:

```
TYPE CONNECTION BLOCK CL_TYPE;
   BEGIN
   REAL PROCEDURE PROC1;
      BEGIN
      % Procedure body statements
      END;
   EXPORT PROC1;
   END;

CL_TYPE SINGLE EXPORTING LIBRARY CL_ONE (INTERFACENAME="CLTESTLIB1.");

CL_TYPE EXPORTING LIBRARY CL_TWO (INTERFACENAME="CLTESTLIB2.",
                        CONNECTIONS=3);
```

Connection Libraries Based on the Same Connection Block

The actual connection libraries are CL_ONE and CL_TWO. These two libraries are completely separate and independent of each other. Nevertheless, the two libraries do follow a common pattern. Hence, CL_ONE and CL_TWO each export a procedure called PROC1. To create connection libraries that contain different procedure declarations, you must base the connection libraries on separate connection blocks.

Multiple Connections of the Same Connection Library

Each connection library supports one or more *connections.* A connection is used to establish a linkage between a connection library and another connection library.

Examples of Single and Multiple Connections

In the previous example, library CL_ONE can support only one connection, because CL_ONE is declared with the keyword SINGLE. The CONNECTIONS attribute cannot be increased by later assignment statements.

By contrast, library CL_TWO initially supports up to 3 connections, because the SINGLE keyword is omitted and the CONNECTIONS library attribute is set to 3. The use of 3 connections enables CL_TWO to link to up to 3 other connection libraries. Further, because the SINGLE keyword is omitted, the CONNECTIONS attribute can be increased by later assignment statements.

Numbering of Connections

The numbering of connections is zero-relative. Thus, a CONNECTIONS value of 3 enables the use of connections numbered 0 through 2.

One-Way Connections

The connections defined in these examples only export data. The EXPORTING keyword is used to specify this and to ensure that this library is only used in one-way connections. Using the EXPORTING or IMPORTING keyword eliminates your ability to do two-way data sharing in this program, but has the significant advantage of helping to ensure that library delinkage does not become an application performance problem.

Establishing Connections

Requesting and Responding Libraries

In any linkage attempt, one library is known as the *requesting library,* and the other library is known as the *responding library.* The same connection library can be used as the requesting library to establish one of its connections, as well as the responding library when establishing another of its connections.

Explicit, Implicit, and Direct Linkage

You can establish linkage between connection libraries in any of the following ways:

- Using *explicit linkage.* In this method, the requesting library process executes a LINKLIBRARY function, and the responding library process executes a READYCL function.
- Using *implicit linkage.* In this method, the requesting library process simply accesses an import object in an unlinked connection, and the responding library process executes a READYCL function.
- Using *direct linkage.* In this method, a process specifies a LINKLIBRARY function that specifies the identifiers of both the requesting library and the responding library.

Linkage Errors

Linkage errors have different effects, depending on the type of linkage that is used. If an implicit linkage results in an error, that error is fatal for the client program. On the other hand, if an explicit or direct linkage results in an error, the error is nonfatal and is communicated to the client program as a LINKLIBRARY function result.

Matching Libraries

Relative to any connection library, the library on the other end of the connection can be referred to as the *matching library.*

Approving and Monitoring Linkages

Each library program can contain APPROVAL and CHANGE procedures that determine whether to allow the linkage and monitor the status of the linkage.

Sharing Instances of Connection Library Programs

Each process that is an execution of the same connection library program is referred to as an *instance* of that library program. You use the sharing option of a library program to tell the system to decide whether to initiate a new instance of the library when the library is being linked to. You can also initiate connection library programs explicitly through task initiation statements.

Equal Partnership in Connections

Once two connection libraries are linked, the distinction between requesting and responding is no longer meaningful. Both libraries are equal partners in the connection, and each can import objects from or export objects to the matching library. Further, either side can sever the connection, as described in "Delinking Connection Libraries".

The following pages describe each of these connection library features in more detail.

Explicitly Linking Connection Libraries

Required Steps for Linking Libraries

To explicitly link two libraries, you must assign appropriate library attributes to each library, code an appropriate LINKLIBRARY function for the requesting library, and code a READYCL function for the responding library.

Order of LINKLIBRARY and READYCL Functions

Either the LINKLIBRARY function or the READYCL function can be executed first. If the LINKLIBRARY function is executed first, then the requesting library program might be delayed while the system waits for the responding library to execute the READYCL function.

Assigning Connection Library Attributes

For linkage to occur, you must assign library attributes to both the requesting and the responding libraries. However, different library attributes are required in each case.

Attributes for the Requesting Library

The following attributes of the requesting library affect linkage. For descriptions of these and other library attributes, refer to "Using Library Attributes" later in this section.

Library Attribute	**Value and Meaning**
LIBACCESS	Specifies how the system is to search for the code file of the responding library. The following are the possible values and their meanings: • BYTITLE The responding library code file is searched for by the value stored in the TITLE library attribute. • BYFUNCTION The responding library code file is searched for by the value stored in the FUNCTIONNAME library attribute. • BYINITIATOR The requesting library must be part of a process that was either initiated by another library process, or initiated by the library linkage mechanism after being invoked by another library. In either case, the BYINITIATOR value causes the requesting library to be linked to the library that originally initiated this instance of the requesting library.
FUNCTIONNAME	If LIBACCESS = BYFUNCTION, specifies the SL function name of the responding library code file. If FUNCTIONNAME is not explicitly assigned, it defaults to the value stored in the INTNAME library attribute.
TITLE	If LIBACCESS = BYTITLE, specifies the title of the library code file. If TITLE is not explicitly assigned, it defaults to the value stored in the INTNAME library attribute.
INTERFACENAME	Specifies which connection library to use in the code file of the responding library. It must match the INTERFACENAME value of the responding library. The INTERFACENAME values must match, even if the responding library program contains only one connection library and regardless of whether the LIBACCESS value is BYTITLE, BYFUNCTION, or BYINITIATOR. If INTERFACENAME is not explicitly assigned, it defaults to the value stored in the INTNAME library attribute. The INTERFACENAME value is used only for implicit and explicit linkages; it is ignored for direct linkages.

Linking Connections to Different Libraries

To cause the various connections of the requesting library to link to different responding libraries, you must vary these library attributes in one of the following ways:

- By specifying substitute values for library attributes in the LINKLIBRARY function. You can specify values for FUNCTIONNAME, TITLE, or INTERFACENAME in this function.
- By assigning different library attributes to the requesting library before each LINKLIBRARY invocation.

Attributes for the Responding Library

The following attributes of the responding library affect linkage. For detailed descriptions of these and other library attributes, refer to "Using Library Attributes" later in this section.

LIBACCESS, FUNCTIONNAME, and TITLE are not used for linkage attempts when this library is the responding library. For INTERFACENAME, a different INTERFACENAME value should be assigned to each connection library in the responding connection library program, so that each library has a unique identification. When a requesting library attempts to link to a responding library, the system selects the responding library with an INTERFACENAME value matching that of the requesting library.

If INTERFACENAME is not explicitly assigned, it defaults to the value stored in the INTNAME library attribute.

Note: *The preceding attributes are ignored for direct linkages.*

Attribute Assignment Examples

The following are examples of attribute assignments in ALGOL:

```
SERVER_CL_TYPE LIBRARY SERVER_CL(CONNECTIONS = 10,
                                 LIBACCESS = BYFUNCTION,
                                 INTERFACENAME = "SECONDLINK.");

LIBRARY (SERVER_CL).CONNECTIONS := 10;
LIBRARY (SERVER_CL).LIBACCESS := VALUE(BYFUNCTION);
REPLACE LIBRARY(SERVER_CL).INTERFACENAME BY "SECONDLINK.";
```

Using LINKLIBRARY for the Requesting Library

LINKLIBRARY Parameters

You can use the LINKLIBRARY function to explicitly initiate linkage of the requesting connection library to another connection library. The LINKLIBRARY function can include the following parameters:

Parameter	**Description**
Connection Library Specifier	For non-SINGLE connection libraries, the connection library specifier can include a connection index. Alternatively, you can use the APPROVAL procedure to select the connection. Note that the index specifies only the connection to use in the requesting library. The requesting library cannot anticipate, and does not need to know, what connection index of the responding library it will be linked to. The requesting library connection links to an unused connection index in the responding library, unless the responding library has an APPROVAL procedure that specifies a particular connection index.
DONTWAIT/ DONTWAITFORFILE/ WAITFORFILE Option	This option indicates the action to take if the matching library is not available. The following are possible values of this option and their effects: • DONTWAIT The linkage attempt proceeds only if a library instance is immediately available (that is, already frozen or readied by the READYCL function). Otherwise, LINKLIBRARY returns an error result. • DONTWAITFORFILE If a library instance is not available, the system initiates a new instance of the library code file. If the library code file is also not available, LINKLIBRARY returns an error result. • WAITFORFILE The linkage attempt continues regardless of whether a library instance or library code file is available. If a library instance is not available, the system initiates a new instance of the library code file. If the library code file is also not available, the linking process is suspended with an RSVP message and waits for an operator response.
Attribute Values	Substitute values for any of the following library attributes: FUNCTIONNAME, INTERFACENAME, or TITLE. These values temporarily override any values previously specified for these attributes.

Required Versus Optional Parameters

Of these parameters, only the connection library specifier is required. The DONTWAIT/ DONTWAITFORFILE/ WAITFORFILE parameter and the library attribute values are optional.

LINKLIBRARY Result

LINKLIBRARY returns a result indicating whether the linkage attempt was successful or not. The possible LINKLIBRARY results are documented in the *ALGOL Programming Reference Manual, Volume 1: Basic Implementation.*

LINKLIBRARY Example

The following ALGOL program fragment declares a connection library and then executes a LINKLIBRARY function to initiate linkage to another connection library:

```
EBCDIC ARRAY ARR[0:35];
REAL RSLT;

TYPE CONNECTION BLOCK TEST1;
   BEGIN
   PROCEDURE PROC2;
      BEGIN
      % Procedure body statements
      END;
   EXPORT PROC2;
   END;

TEST1 EXPORTING LIBRARY CL1 (LIBACCESS = BYTITLE, TITLE = "OBJECT/CLTEST.",
                   CONNECTIONS = 3);

REPLACE ARR BY "CLTEST.";
RSLT:= LINKLIBRARY (CL1[0], WAITFORFILE, INTERFACENAME = ARR);
```

In this example, the LINKLIBRARY function attempts to link connection 0 of CL1 to the library with an INTERFACENAME of CLTEST in the connection library program OBJECT/CLTEST. If program OBJECT/CLTEST is not available, then this process is suspended with an RSVP message.

In response to the LINKLIBRARY function, the system initiates OBJECT/CLTEST, if necessary. Then the system waits for OBJECT/CLTEST to ready connection library CLTEST. Thereafter, the system can complete the linkage.

Using READYCL for the Responding Library

READYCL Compared to FREEZE

For the responding connection library, the READYCL function indicates that the export objects in the connection library are now available. Thus, READYCL serves a purpose roughly similar to that of a FREEZE statement in a server library. However, unlike the FREEZE statement, the READYCL function never halts execution of the connection library program. Nor does the READYCL function offer any equivalent to the PERMANENT, TEMPORARY, and CONTROL options of the FREEZE statement.

READYCL Syntax

Unlike the LINKLIBRARY function, the READYCL function applies to the whole connection library instead of to a single connection. READYCL also returns a real value as a result. Thus, for a connection library named CL1 and a real variable named RSLT, the following ALGOL statement could be used:

```
RSLT:= READYCL (CL1);
```

READYCL Result

Field [15:16] of the return value stores the result of the operation. A zero indicates success. Odd numbers are errors, which indicate that the function failed. Even numbers are reserved for future use. The possible READYCL results are documented in the *ALGOL Programming Reference Manual, Volume 1: Basic Implementation.*

READYCL Example

The following statements in OBJECT/CLTEST declare library CL1 and ready it.

```
TYPE CONNECTION BLOCK TEST1;
   BEGIN
   PROCEDURE PROC2;
      IMPORTED;
   PROCEDURE PROC1;
      BEGIN
      % Procedure body statements
      END;
   EXPORT PROC1;
   END;

REAL RYRSLT;

TEST1 SINGLE LIBRARY CL2 (INTERFACENAME="CLTEST.");

RYRSLT:= READYCL (CL2);
```

In the preceding example, the READYCL function makes it possible for another connection library to be linked to this one.

Reversing a READYCL Action

You can reverse the effects of the READYCL function by using the UNREADYCL function, as discussed in “Unreadying a Connection Library”.

Implicitly Linking Connection Libraries

Implicit linkage can occur if a process accesses an import object in a connection and no matching library has yet been linked to that connection.

Example of Causing Implicit Linkage

For example, suppose a connection library LIB1 has import declarations for three procedures: PROC1, PROC2, and PROC3. Suppose also that LIB1 has two connections, and the connection library process uses the LINKLIBRARY function to initiate linkage for connection 0. The connection library process then invokes PROC2 in connection 1. Note that, at this point, connection 1 has not yet been linked to a matching library.

AUTOLINK Library Attribute

The AUTOLINK library attribute determines what happens at this point. If AUTOLINK is FALSE (the default value), then the attempt to invoke PROC2 fails with a fatal run-time error. If AUTOLINK is TRUE, then the system proceeds with an attempt to link the connection to a matching library. The system uses library attributes to search for the match, as previously described in "Assigning Connection Library Attributes".

READYCL Function Required

For linkage to reach completion, the matching library must be readied by a READYCL function, as described in "Using READYCL for the Responding Library".

Missing Library

If the matching library program is not ready or otherwise cannot be found, the invoking process becomes suspended at this point. If the LIBACCESS value is BYTITLE, then a "NO LIBRARY" message is displayed as the RSVP message. If the LIBACCESS value is BYFUNCTION, and the FUNCTIONNAME library attribute requests a function name that does not exist, the RSVP message is "FUNCTION <function name> IS NOT DEFINED, SL, FA, OR DS."

Missing Objects

If the matching library program is found, the system links the requesting library to an existing instance of the matching library program or initiates a new instance as discussed under "Implicitly Initiating a Connection Library" later in this section. The system then checks to see if the requested object is exported by the matching library. If the requested object is not available, the system returns one of the errors discussed in "Matching Connection Library Objects".

Directly Linking Connection Libraries

If a process has access to the declarations of two different connection libraries, the process can link the libraries together by specifying both library identifiers in the LINKLIBRARY function. This type of linkage is referred to as *direct linkage,* because it bypasses the mechanisms that the system otherwise would use to locate the matching library.

Syntax for Direct Linkage

The direct form of the LINKLIBRARY function is available in both ALGOL and NEWP. The following is an example of such a LINKLIBRARY call:

```
RSLT := LINKLIBRARY(CLIB1, LIBRARY = CLIB2);
```

Specifying a Connection Index

In this example, CLIB1 and CLIB2 are both connection library identifiers. Optionally, you can include a connection index for either or both of the connection libraries, as in the following example:

```
RSLT := LINKLIBRARY(CLIB1[1], LIBRARY = CLIB2[3]);
```

Requesting and Responding Library

The system treats the first library (CLIB1) as the requesting library and the second library (CLIB2) as the responding library.

READYCL Not Necessary

It is not necessary to ready the responding library with the READYCL function.

Most Library Attributes Ignored

When you use direct linkage, the system ignores the values of the library attributes FUNCTIONNAME, INTERFACENAME, LIBACCESS, and TITLE.

LINKLIBRARY Parameters Restricted

The direct form of the LINKLIBRARY function cannot include a DONTWAIT/ DONTWAITFORFILE/WAITFORFILE option or any library attribute values.

APPROVAL Procedure Skipped

The system omits executing the APPROVAL procedure for either library unless the APPROVAL procedure is needed to determine the connection index. That is, the system executes the APPROVAL procedure for either library only if that library supports multiple connections and no connection index was specified for that library in the LINKLIBRARY statement.

Note that the LIBPARAMETER library attribute provides input to the APPROVAL procedure. If the APPROVAL procedure is going to be skipped, there is no point in assigning a LIBPARAMETER value.

Other Library Attributes

Other library attributes such as CHANGE and LIBPARAMETER have the same effects that they do for implicit or explicit linkages.

Controlling Connection Library Sharing

SHARING Option

You can use the compiler control option SHARING to specify whether one or many processes can link to the same instance of a connection library. The SHARING option has the same range of values for connection libraries that it does for server libraries. The meanings are also roughly the same.

Responding Library Determines Sharing

When one connection library links to another, the SHARING option of the responding library determines whether a new library instance is created.

Same Sharing for All Libraries in a Program

Because the SHARING option can be specified only once in a program (at the start), all the connection libraries in a single program use the same SHARING value.

The following paragraphs explain the special implications that each SHARING value has for a connection library.

SHARING = PRIVATE

The system initiates a separate instance of the connection library program each time a process attempts to link to any connection library in the program. Even if two processes link to different connection libraries in the PRIVATE library program, the system still initiates two separate instances of the PRIVATE library program. As a result, processes linking to a connection library in the PRIVATE library program will never use more than the first connection specified for that library

However, the PRIVATE library program could itself use repeated LINKLIBRARY functions to create multiple connections between a connection library and other libraries.

The PRIVATE sharing option is less useful for connection libraries than it is for server libraries, because SHAREDBYALL connection libraries can use connection objects to store data that is different for each user. Refer to “Connection Objects in Connection Libraries”.

SHARING = SHAREDBYALL

All processes that link to the library program share the same instance of the library. If one such process changes the value of a global object in the library, the next process that interrogates the global object receives the changed value. However, if one such process changes the value of a connection object in the library, the change is visible only to the process that is linked through that connection. The SHAREDBYALL option is the SHARING option typically used for connection library programs

The SHAREDBYALL option should not typically be used for connection libraries that have the SINGLE keyword. The SHAREDBYALL option ensures that all processes that link to the library will access the same library instance, but the SINGLE keyword ensures that only one connection is available for linking to that library instance. Therefore, only one process at a time can be linked to the library. If any other process attempts to link to the connection library when the library is already in use, the linkage attempt fails immediately and returns an error result.

SHARING = SHAREDBYRUNUNIT

This value cannot be used for connection libraries. If a program has SHARING = SHAREDBYRUNUNIT, then any attempt to perform a READYCL function for a connection library declared in that program results in a runtime error.

SHARING = DONTCARE

This option is a nonpreferred synonym for SHAREDBYALL

Default SHARING Value

If a connection library program does not include the SHARING compiler control option, then the compiler assigns a default SHARING option of SHAREDBYALL to the program.

Approving a Connection

APPROVAL Procedure

Connection libraries can optionally include a procedure to determine whether any given process should be permitted to link to the library. This procedure is called the APPROVAL procedure. If there is an APPROVAL procedure, you must include an APPROVAL attribute assignment in the connection library declaration. The APPROVAL attribute specifies the name of the procedure to be used as the APPROVAL procedure. The APPROVAL procedure itself must be declared outside the connection library.

Dual APPROVAL Procedures

When a connection library attempts to link to another connection library, there can be two different APPROVAL procedures involved. Each connection library program can contain its own APPROVAL procedure, and each connection library declaration can contain its own APPROVAL attribute assignment. The APPROVAL procedure declared by the requesting library process is executed before the APPROVAL procedure declared by the responding library process.

Parameters to the APPROVAL Procedure

The system passes parameters to the APPROVAL procedure including the task variable of the process that declared the matching library and the LIBPARAMETER attribute specified by the matching process, if any. The APPROVAL procedure can read the task attributes and the LIBPARAMETER value. The APPROVAL procedure then returns a real value indicating whether linkage is permitted. The APPROVAL procedure for either library can also select which connection to use for that end of the linkage. If the APPROVAL procedure does not permit linkage, then the system does not link the two processes together.

For a description of the APPROVAL procedure, refer to the discussion of the APPROVAL library attribute under "Using Library Attributes" later in this section.

APPROVAL Procedure Example

The following ALGOL example declares and readies a connection library, CL1. The APPROVAL procedure in the example ensures that only processes with a given usercode are allowed to link to the connection library:

```
TYPE CONNECTION BLOCK TEST1;
   BEGIN
   PROCEDURE PROC1;
      BEGIN
      % Procedure body statements
      END;
   EXPORT PROC1;
   END;

REAL PROCEDURE CL1_APPROV (OWNER, LIBPAR, LEN, WAIT);
      VALUE        LEN, WAIT;
      TASK         OWNER;
      EBCDIC ARRAY LIBPAR[*];
      INTEGER      LEN;
      BOOLEAN      WAIT;
   BEGIN
   EBCDIC ARRAY UC[0:17];
   REPLACE UC BY OWNER.USERCODE;
   IF UC = "TRUSTME." THEN
        CL1_APPROV.[47:4]:= 2 % Allow linkage; MCP chooses connection.
   ELSE
      BEGIN
      CL1_APPROV.[47:4]:= 3; % Forbid linkage
      CL1_APPROV.[38:39]:= 510; % Notify that a bad usercode was used.
                                % Meaning of this value is established
                                % by convention between the applications.
      END;
   END;

TEST1 LIBRARY CL1 (INTERFACENAME="CLTEST.", CONNECTIONS = 10,
                   APPROVAL = CL1_APPROV);

RYRSLT:=READYCL (CL1);
```

Monitoring Connection State

A program can monitor the status of a connection in either of two ways: by using the STATE connection attribute or the CHANGE library attribute.

Checking the Current State

The STATE attribute is a read-only attribute that returns the current state of a connection. The possible mnemonic values are NOTLINKED, LINKING, LINKED, or DELINKING. The following ALGOL statement checks the state of connection 4 of connection library CL1:

```
IF LIBRARY(CL1[4]).STATE = VALUE(LINKED) THEN...
```

Notification of Changes

The STATE attribute returns the current state of a connection at the time that you interrogate the attribute. By contrast, the CHANGE attribute allows you to arrange for automatic notification whenever any connection happens to change state.

CHANGE Procedure

The CHANGE attribute specifies the name of a procedure called the CHANGE procedure. You must implement this procedure yourself, including the standard set of parameters. The CHANGE procedure, if there is one, must be declared within the connection block of the connection library.

Transition to New State

Whenever the linkage state of any connection to the library changes, the system calls the CHANGE procedure. Note that the connection cannot fully transition to the new state until the CHANGE procedure exits. Therefore, when you design a CHANGE procedure, you should be careful to ensure that the CHANGE procedure exits quickly. For example, the CHANGE procedure should not wait for a port file read or for an event that might never be caused.

Dual CHANGE Procedures

When a particular connection changes state, there can be two different CHANGE procedures involved: one associated with the requesting library and one associated with the responding library. The requesting library is the library that caused the linkage or delinkage. The responding library is the library on the receiving end of the linkage or delinkage. Note that the library that is requesting during the linkage process might be responding during the delinkage process, and vice versa. The CHANGE procedure declared by the requesting library is executed before the CHANGE procedure declared by the responding library.

Parameters to CHANGE Procedure

The system passes the following information as parameters to the CHANGE procedure:

- The index of the connection that has changed state
- The new linkage state
- The reason for the change in linkage state
- The task variable of the active library process (the process that caused the change in linkage state)
- An indication of whether the active library is terminating abnormally

Results Returned by CHANGE Procedure

The CHANGE procedure can read the parameter values, but does not return any value itself. However, the CHANGE procedure can modify globally-declared objects in the connection library program where it is declared. These objects can include event variables. These modifications can be used to inform the connection library of the change in linkage state.

For a description of the CHANGE procedure, refer to the discussion of the CHANGE library attribute under “Using Library Attributes” later in this section.

CHANGE Procedure Example

The following ALGOL statements declare a connection library called CL1 with a CHANGE procedure called CL1_CHG. An underlying assumption is that each connection of the library might be linked to and delinked from repeatedly. The CL1_CHG procedure increments the value of connection object LINKNUM each time a process completes linkage through that connection. Note that, because LINKNUM is declared in the connection block, the system creates a separate instance of LINKNUM for each connection. Each instance of LINKNUM thus keeps a running count of the number of times that connection has been used to link to CL1.

```
INTEGER GLOBALINT;
REAL RSLT;

TYPE CONNECTION BLOCK TEST1;
   BEGIN

   INTEGER LINKNUM;

   PROCEDURE CL1_CHG (CONN_INDEX, NEW_STATE, REASON, ACTOR, IMDSED);
         VALUE   CONN_INDEX, NEW_STATE, REASON, IMDSED;
         INTEGER CONN_INDEX, NEW_STATE, REASON;
         TASK    ACTOR;
         BOOLEAN IMDSED;
      BEGIN
      IF NEW_STATE = VALUE(LINKED) THEN
         LINKNUM:= * + 1;
      END;

   PROCEDURE PROC1;
      BEGIN
      % Procedure body statements
      END;

   EXPORT PROC1;

   END;


TEST1 LIBRARY CL1 (INTERFACENAME="CLTEST.", CONNECTIONS = 10,
                   CHANGE = CL1_CHG);

RSLT:= READYCL(CL1);
```

Note that the CL1_CHG procedure in the preceding example updates only a connection object. If CL1_CHG updated a global object (such as GLOBALINT in the preceding example), then the possibility would arise that more than one invocation of CL1_CHG might be attempting to update the global object at the same time. To prevent updates from overwriting each other, you would need to add a timing mechanism involving the use of PROCURE and LIBERATE statements and a shared global event.

Matching Connection Library Objects

As part of the linkage process resulting from a LINKLIBRARY request, the system checks to see whether the objects imported and exported by the requesting library and the responding library match up.

Missing Objects during Linkage

Ideally, for each object imported by one library, an exported object of the same name should be declared by the other library. However, library linkage can succeed even if the system does not find matches for some of the imported and exported objects. The following minimum object matches must succeed in order for linkage to take place:

- If any objects are imported on either side, then a minimum of one object must match.
- If the linkage was initiated implicitly, by invoking an import object in an unlinked library, then a match must be found for that particular import object.

Error When Missing Object Is Used

The system returns an error for a missing object when that object is first used. (In the case of implicit linkage, these errors can occur at linkage time if they apply to the object whose usage caused the implicit linkage.)

The following are the errors that the system can return:

- MISSING OBJECT <object name> IN LIBRARY <library name>

  No object with the requested name is exported by the matching library. The system discontinues the importing process.
- OBJECT <object name> LINKAGE CLASS VIOLATION IN LIBRARY <library name>

  The importing process lacks the linkage class necessary to use the exported object. The system discontinues the importing process.
- Object <procedure name>: Type or parameter mismatch in interface <client library identifier> to library <library name>

  An export object with the requested name exists, but the type or parameters of the import and export objects do not match. The system discontinues the importing process. For an explanation of the rules used to match object types and parameters, refer to “Type Matching” later in this section.
- OBJECT <object name> ACCESS MODE MISMATCH

  A process attempted to import a data object as read-write when the object was exported as read-only. The system discontinues the importing process.

Checking the Availability of an Object

To prevent the possibility of a fatal error when using an imported object, the library can use the ISVALID function in ALGOL or NEWP. The ISVALID function returns a Boolean result indicating whether a given object is valid. For example, the following statement invokes procedure PROC1 only if that procedure is valid:

```
IF ISVALID(PROC1) THEN PROC1;
```

For imported data, the ISVALID function can check whether the data is available for reading or for reading and writing. The following statement reads integer J if it is available, regardless of whether the access mode is read-only or read-write:

```
IF ISVALID(J) THEN I := J;
```

The following statement writes to integer J if it is available and the access mode is read-write.

```
IF ISVALID(J, READWRITE) THEN J := 43;
```

Initiating Connection Libraries

Connection library processes can be initiated in either of two ways: implicitly, because of a LINKLIBRARY function, or explicitly, by a process-initiation statement in another process.

Implicitly Initiating a Connection Library

Library Linkage Mechanism

If a connection library process attempts to link to another connection library, and no appropriate instance of the responding connection library is currently available, then the requesting library process enters a waiting state. The STATUS task attribute is still ACTIVE, but the stack state in the Y (Status Interrogate) system command is WAITING ON AN EVENT. The library linkage mechanism of the operating system automatically initiates the program containing the responding connection library. The responding connection library program executes normally and at some point executes a READYCL function for the connection library. At this point, the system completes the linkage between the requesting library and the responding library, and the requesting library process resumes execution.

Multiple Library Instances

Even if an instance of the responding connection library program is already available, the system might initiate a new instance of the library program for the requesting library to link to. For example, if the sharing option of the connection library is PRIVATE, then the system initiates a new instance of the connection library program each time a process attempts to link to the connection library.

SHAREDBYALL Libraries with All Connections in Use

If the requesting library attempts to link to a responding library program that is SHAREDBYALL, and no connections of the responding library are available for use, the linkage attempt fails immediately and returns an error result. The system does not initiate a new instance of the responding library program.

Programs That Cannot Become Libraries

If the process attempts to link to a program that is not capable of becoming either a server library or a connection library, then the library linkage mechanism issues the error "LIBRARY WAS NOT INITIATED: <library name>." This error can happen, for example, if the client process attempts to link to an ALGOL program that does not contain a FREEZE statement or a READYCL function.

Programs That Never READYCL or FREEZE

Because a library program initially runs as a regular program, the flow of execution can be such that the execution of a READYCL function or FREEZE is conditional. If the library linkage mechanism initiates such a program, and the resulting process terminates without ever having executed a READYCL function or FREEZE statement, the library linkage fails and the LINKLIBRARY function returns an error.

Explicitly Initiating a Connection Library

Process Initiation Statements

A program containing connection libraries can be explicitly initiated by a process initiation statement, just like any other program.

Types of Library Processes

Connection library programs can be initiated as any type of process: independent, synchronous dependent, or asynchronous dependent. (By contrast, server libraries cannot be initiated as synchronous dependent processes.)

Internal Tasks

In ALGOL programs, an internal procedure can be initiated as a task and can later participate in connection library linkages by executing READYCL or LINKLIBRARY functions. NEWP programs marked with UNSAFE(TASKING) status also have this capability. However, to simplify this discussion, this section discusses connection library processes as if they were always executions of an entire program

Sharing Option

For connection library programs that are explicitly initiated, some special considerations apply to the sharing option. If other processes need to be able to link to the connection library instance, then a sharing option of SHAREDBYALL must be used. However, the connection library program can initiate linkages to other processes regardless of whether its own sharing value is SHAREDBYALL, PRIVATE, or SHAREDBYRUNUNIT.

Determining How the Program Was Initiated

A connection library program can determine whether it was initiated explicitly or by the library linkage mechanism by interrogating the LIBRARYSTATE task attribute. Bit [0:1] of the LIBRARYSTATE value stores a 1 if the process was initiated by the library linkage mechanism.

Unreadying a Connection Library

Preventing Further Linkages

You can use the UNREADYCL function to make a given connection library unready for new linkages. While a connection library is unready, it cannot serve as the responding side in an implicit or explicit linkage attempt. However, an unready connection library can serve as the responding side in a direct linkage attempt or as the requesting side in any type of linkage attempt.

UNREADYCL has no effect on existing linkages.

Compare with READYCL Function

UNREADYCL performs a function opposite to that performed by the READYCL function, as discussed in “Using READYCL for the Responding Library,” earlier in this section. You can use these functions to ready and unready a connection library repeatedly.

Delinking Connection Libraries

Delinkage Due to Block Exits

The system automatically delinks two connection libraries when a process exits a block in which one of the connection libraries is declared.

Requesting Delinkage

Alternatively, either of the linked library processes can use the DELINKLIBRARY function to delink the connection. When used with connection libraries, the DELINKLIBRARY function affects only one connection and must specify a particular connection index or a SINGLE connection library. DELINKLIBRARY also provides two options for connection libraries that are not available for server libraries: the ORDERLY/ABORT option and the WAIT/DONTWAIT option.

Delinking a library is a very costly operation. For more information, refer to “Overuse of Delinkage.”

ORDERLY/ABORT Option

The ORDERLY/ABORT option indicates what to do if any objects exported through this connection from either side are currently being used. The following are the possible values and their meanings:

- A value of ORDERLY indicates that the delinkage is delayed until none of the exported objects are in use. The ORDERLY value also prevents any new access to the exported objects while delinkage is taking place. ORDERLY is the default value of the ORDERLY/ABORT option.
- A value of ABORT causes any processes using objects exported through this connection to be discontinued (possibly including the process that invoked DELINKLIBRARY). The discontinued processes receive a HISTORYCAUSE task attribute value of 2 (PROGRAMCAUSEV) and a HISTORYREASON task attribute value of 101 (LIBCANCELERRV).

WAIT/DONTWAIT Option

The WAIT/DONTWAIT option indicates when control should return from the DELINKLIBRARY function. The following are the possible values and their meanings:

- A value of WAIT specifies that the DELINKLIBRARY function does not finish until the delinkage is complete. WAIT is the default value of the WAIT/DONTWAIT option.
- A value of DONTWAIT specifies that the DELINKLIBRARY function finishes as soon as it has initiated the delinkage. If the DONTWAIT value is used, the program can later interrogate the connection attribute STATE to determine if delinkage has completed. Alternatively, the program can issue another DELINKLIBRARY call, perhaps with the WAIT value specified.

Using PROLOG and EPILOG Procedures

A connection block declaration can contain two special types of procedures: a PROLOG procedure, an EPILOG procedure, or both. These procedures are available in both ALGOL and NEWP.

PROLOG Procedures

For each connection in any connection library based on that connection block TYPE, the PROLOG procedure is executed the first time that connection is used in any way. The first use of a connection might occur in any of the following ways:

- The connection might be specified in a LINKLIBRARY statement in the same program.
- A statement elsewhere in the same program might access an object in the connection library. The object might be an imported object or a local object.
- Another connection library might initiate a linkage to this one.

EPILOG Procedures

By contrast, an EPILOG procedure specifies the actions performed for each connection just before the connection block is deleted. The connection block is deleted when a program exits the procedure (or outer block) in which the connection block is declared. A separate instance of the EPILOG procedure is executed for each active connection of each connection library that is based on that connection block. In this context, a connection is considered active if it is currently linked or was previously linked and then delinked. (If a particular connection was declared, but never used, then the EPILOG procedure is not executed for that connection.)

Uses Outside Connection Blocks

PROLOG and EPILOG procedures can also be declared in structure blocks, where they behave similarly to the way they behave for connection blocks. Furthermore, EPILOG procedures can be declared in any procedure, and specify actions to be performed when the procedure is exited (refer to "Using EPILOG and EXCEPTION Procedures" in Section 16, "Using Events and Interlocks").

Order of Execution for EPILOG Procedures

A connection block EPILOG procedure is executed before any EPILOG procedure for the procedure that declares the connection block.

PROLOG and EPILOG Example

The following ALGOL example includes various combinations of PROLOG and EPILOG procedures:

```
PROCEDURE OUTERPROC;
   BEGIN
   EBCDIC ARRAY LIBF[0:23];

   EPILOG PROCEDURE OUTER_EPILOG;
      BEGIN
      % Procedure body statements
      END;

   TYPE CONNECTION BLOCK CL_TYPE;
      BEGIN
      INTEGER CONN_INT;

      PROLOG PROCEDURE CL_PROLOG;
        BEGIN
        % Procedure body statements
        END;

      EPILOG PROCEDURE CL_EPILOG;
        BEGIN
        % Procedure body statements
        END;

      PROCEDURE CL_INIT(I);
        INTEGER I;
```

```
340        BEGIN
350        CONN_INT:= I;
360        END;
370
380      PROCEDURE CL_EXP;
390        BEGIN
400        % Procedure body statements
410        END;
420      EXPORT CLEXP;
430      END;
440
450   CL_TYPE LIBRARY CL1 (LIBACCESS=BYFUNCTION, FUNCTIONNAME="F1.",
455                        CONNECTIONS=3);
460
470   % End declarations, begin statements of procedure OUTERPROC
480   CL1[0].CL_INIT(5);
485   REPLACE LIBF BY "CLTEST1.";
490   LINKLIBRARY (CL1[0], WAITFORFILE, INTERFACENAME = LIBF);
500   REPLACE LIBF BY "CLTEST2.";
510   LINKLIBRARY (CL1[1], WAITFORFILE, INTERFACENAME = LIBF);
520
530 END OUTERPROC;
```

In the preceding example, the statement at line 480 is the first statement to use connection 0, and therefore causes the CL_PROLOG procedure to be executed for connection 0. The statement at line 510 causes the CL_PROLOG procedure to be executed for connection 1. The CL_PROLOG procedure is never executed for connection 2, because connection 2 is never actually used in this example.

When the OUTERPROC procedure is being exited, the system invokes separate instances of the CL_EPILOG procedure for connections 0 and 1. CL_EPILOG is not invoked for connection 2, because connection 2 is never used in this example. Then the system invokes the OUTER_EPILOG procedure. OUTER_EPILOG is executed only once, because OUTER_EPILOG is associated with the procedure OUTERPROC rather than with the connection block CL_TYPE.

Using Connection Library Objects

Like server libraries, connection libraries provide access to objects that are local to exported procedures, objects that are global to such procedures, and data objects that are explicitly exported. However, connection libraries also support objects that are local to a particular connection. The following pages explain the differences between these various types of objects.

Global Objects in Connection Libraries

Scope of Global Objects

The global objects for a connection library consist of all the objects that are visible to the connection library but that are declared outside of the connection block. The rules governing the scope of declarations in ALGOL are discussed under "Communication through Global Objects in ALGOL" in Section 15, "Using Global Objects." The same scope rules apply to NEWP programs.

Interprocess Communication and Timing

The use of global objects in connection libraries provides a flexible tool for interprocess communication, with the same advantages previously described for server libraries under "Global Objects in Server Libraries." The use of global objects also makes it necessary to consider using events or other such mechanisms to regulate the timing of processes that use a global object.

Global Object Example

Procedures in a connection library can read or change the values of global objects. Any changes are visible to later invocations of the same or different procedures in the same or different connections of that library. The following DMALGOL example includes a connection library procedure PROC1 making changes to the global object GLOBINT:

```
BEGIN

REAL RSLT;
INTEGER GLOBINT;
EVENT GLOBACCESS;
EBCDIC ARRAY LIBF[0:23];

LIBRARY MCPSUPPORT (LIBACCESS = BYFUNCTION);

REAL PROCEDURE EVENT_STATUS(EV);
     EVENT EV;
     LIBRARY MCPSUPPORT;

DEFINE LOCKOWNERF = [18:15] #; % Lock owner field in EVENT_STATUS result

TYPE CONNECTION BLOCK CL_TYPE;
   BEGIN

   REAL PROCEDURE PROC1;
      BEGIN
      INTEGER TEMPINT;
      PROTECTED EXCEPTION PROCEDURE KLEENUP;
         BEGIN
         IF EVENT_STATUS(GLOBACCESS).LOCKOWNERF = PROCESSID THEN
            BEGIN
            IF TEMPINT = GLOBINT THEN
               GLOBINT:= * + 10;
            LIBERATE (GLOBACCESS);
            END;
         END;
      PROCURE (GLOBACCESS);
```

```
            TEMPINT:= GLOBINT;
            GLOBINT:= * + 10;
            PROC1:= GLOBINT;
            KLEENUP;
            END;
         EXPORT PROC1;
         END;

      CL_TYPE LIBRARY CL1 (LIBACCESS = BYFUNCTION, FUNCTIONNAME = "TESTF.",
                          CONNECTIONS=2);

      GLOBINT:= 0;
      REPLACE LIBF BY "LIB0.";
      RSLT:= LINKLIBRARY(CL1[0], INTERFACENAME = LIBF);
      REPLACE LIBF BY "LIB1.";
      RSLT:= LINKLIBRARY(CL1[1], INTERFACENAME = LIBF);

      END.
```

Explanation of Example: GLOBINT

This library links to two matching libraries. Each time either matching library invokes procedure PROC1, the procedure increments the value of GLOBINT by 10.

The current value of GLOBINT is therefore available to three different processes: the connection library program that declares GLOBINT and the two other connection library processes that link to library CL1. The event GLOBACCESS is used to ensure that no two processes perform conflicting, simultaneous updates or interrogations of GLOBINT.

Explanation of Example: KLEENUP Procedure

Procedure PROC1 includes a protected EXCEPTION procedure called KLEENUP. The purpose of KLEENUP is to make sure that the event GLOBACCESS is liberated, and GLOBINT is updated, if a process is discontinued while executing PROC1. Protected EXCEPTION procedures are the most reliable technique for performing such cleanup work, because the PROTECTED clause prevents the procedure from being discontinued. However, the PROTECTED clause is available only in DMALGOL and NEWP. For information about this use of EXCEPTION procedures, refer to “Using EPILOG and EXCEPTION Procedures” in Section 16, “Using Events and Interlocks.”

Explanation of Example: EVENT_STATUS

The KLEENUP procedure itself invokes an MCP library procedure called EVENT_STATUS. The purpose of this call is to determine whether PROC1 had succeeded in procuring the event GLOBACCESS before being discontinued. KLEENUP liberates GLOBACCESS only if EVENT_STATUS indicates that the current process has already succeeded in procuring the event.

Connection Objects in Connection Libraries

Objects Declared in Connection Blocks

The connection objects for a connection library consist of all the objects that are declared in the connection block, including all procedures or variables. The system creates a separate instance of each connection object for each connection of the library.

References Inside the Connection Block

All connection objects can be referenced by any procedures that are declared later in the same connection block.

References Outside the Connection Block

In NEWP, statements outside the connection block can reference any objects declared in the connection block by prefixing the object with the connection library name and the connection index.

In ALGOL, procedures, events, and event arrays are the only connection objects that can be referenced by statements outside the connection block. Such statements must prefix the object name with the connection library name and the connection index. To make it possible for statements outside the connection block to change the values of other types of connection variables, you must implement connection procedures that provide such access indirectly.

Usage Before and After Linkage

For either ALGOL or NEWP, when objects in a connection library are used by statements in the same program, the following rules apply:

- Imported objects cannot be used until the connection has been linked to a matching connection library.
- Exported objects, or objects neither imported nor exported, can be used either before or after library linkage.

Usage by the Matching Library Program

The matching connection library program has more limited access to objects in this connection library. The matching program can access only objects that are exported by the connection library, and can access these objects only after library linkage. The matching program does not have access to any variables declared in the connection block, unless those variables are explicitly exported. However, exported procedures can provide indirect access to connection block variables or procedures that are not exported.

Values Retained between Repeated Linkages

If a connection is linked, delinked, and relinked, all connection objects retain their last value. The connection objects are not automatically reinitialized during linkage. However, you can design a CHANGE or APPROVAL procedure to include statements that initialize connection objects.

Connection Objects Example

The following ALGOL example illustrates several issues regarding the visibility of connection objects:

```
REAL RVAL;
EBCDIC ARRAY LIBF[0:23];

TYPE CONNECTION BLOCK CL_CSPEC;
   BEGIN
   INTEGER CONN_INT;

   PROCEDURE PROC1;
      BEGIN
      CONN_INT:= * + 10;
      END;

   PROCEDURE PROC2;
      IMPORTED;

   REAL PROCEDURE PROC3;
      BEGIN
      PROC3:= CONN_INT;
      END;
   EXPORT PROC3;
   END;

CL_CSPEC LIBRARY CL_ONE (LIBACCESS = BYFUNCTION,
                         FUNCTIONNAME = "TESTF.",
                         CONNECTIONS = 2);

CL_ONE[0].PROC1;
CL_ONE[1].PROC1;
RVAL:= CL_ONE[0].PROC3;

REPLACE LIBF BY "LIB1.";
LINKLIBRARY (CL_ONE[0], INTERFACENAME = LIBF);
REPLACE LIBF BY "LIB2.";
LINKLIBRARY (CL_ONE[1], INTERFACENAME = LIBF);

CL_ONE[1].PROC2;
```

Explanation of Example: Objects Visible within Program

In the preceding example, PROC1 is provided to allow statements in the same program to indirectly access the connection object CONN_INT. Any changes made to CONN_INT are local to a single connection of the library.

Explanation of Example: Procedures Invoked Before and After Linkage

Procedures PROC1 and PROC3 can be invoked before library linkage because neither is an imported procedure. Procedure PROC2, which is imported, can be invoked only after library linkage. This illustrates using connection libraries to simultaneously import and export objects. However, the potential performance problems of using two-way connections that contain both imported and exported objects should be considered before designing an application in this way. For more information, refer to “Design Strategies for Linking Libraries.”

Explanation of Example: Objects Visible to Matching Libraries

Of the objects declared by library CL_ONE, the only object that can be used by the matching libraries is PROC3. This is true because the only object listed in the EXPORT statement is PROC3. However, PROC3 does provide the matching library with indirect access to the connection variable CONN_INT.

Local Objects in Connection Libraries

Local Variables Reinitialized

Any local variables declared in a procedure exported by a connection library are reinitialized each time that procedure is invoked.

Local Variables Example

The following ALGOL example declares a connection library CL_ONE that exports procedure PROC1. Procedure PROC1 declares a variable LOCAL_INT and increments the value of this variable.

```
EBCDIC ARRAY LIBF[0:23];

TYPE CONNECTION BLOCK CL_CSPEC;
   BEGIN

   PROCEDURE PROC1;
      BEGIN
      INTEGER LOCAL_INT;
      LOCAL_INT:= * + 10;
      END;

   EXPORT PROC1;

   END;

CL_CSPEC LIBRARY CL_ONE (LIBACCESS = BYFUNCTION,
                         FUNCTIONNAME = "TESTF.",
                         CONNECTIONS = 2);

REPLACE LIBF BY "LIB1.";
LINKLIBRARY (CL_ONE[0], INTERFACENAME = LIBF);
REPLACE LIBF BY "LIB2.";
LINKLIBRARY (CL_ONE[1], INTERFACENAME = LIBF);
```

In this example, variable LOCAL_INT will never exceed a value of 10, no matter how many times procedure PROC1 is invoked.

Exported and Imported Data in Connection Libraries

Advantages and Disadvantages

By exporting data, you provide the matching library process with direct access to the data, rather than the indirect type of access described in "Global Objects in Connection Libraries" and "Connection Objects in Connection Libraries" earlier in this section. The following table compares these methods of providing data:

Capability	Exporting Data	Exporting Procedures That Access Global Data
Ease of Implementation	The importing program is given direct access to the data.	You have to implement library procedures that provide an interface to the data.
Maintainability	Changes to data structures, semantics, and access rules may require revision of all client programs.	Changes to data structures, semantics, and access rules may be hidden from client programs.
Processor Overhead	Low.	Somewhat higher, due to the cost of invoking a procedure.
Sharing	For a multiple-connection library, each connection exports a *different* instance of the data object.	For a multiple-connection library, all matching libraries receive indirect access to the *same* instance of the global variable. However, exported procedures could provide access to connection objects, of which a *different* instance exists for each connection.
Writeability	The library can specify whether clients have read-write or read-only access to the variable.	The exported procedures can each contain logic to provide read-write or read-only access.
Importing Process Identity	The library provides the same access rights to all importing processes. However, individual importing processes can reduce their own access rights by requesting read-only access to data that was exported with read-write access.	The exported procedures can include code to interrogate the identity of the importing process, and allow different actions for different clients.

Mutual Exclusion

Mutual exclusion is the ability of a process to secure exclusive access to data before updating it.

For a connection library providing indirect access to a global object, the exported procedures can be coded to always procure a global event before updating the global variable. In this way, the library can ensure that the event is updated by no more than one process at a time.

For data exported by connection libraries, mutual exclusion might not be necessary because each importing library receives a different instance of the exported data object. However, you might need to ensure mutual exclusion in any of the following cases:

- The exporting connection library process and the importing connection library process both access the exported data object.
- One of the importing library processes initiates internal tasks using the library through the same library declaration.

If mutual exclusion is necessary, the exporting connection library can export an event to be used together with the exported data object. Each importing process is expected to procure the event before updating the data object. However, the exporting connection library cannot enforce the use of this event.

Data Types That Can Be Exported

Currently, NEWP exports a more limited range of data types than ALGOL. For information about the types of data that can be exported and imported in ALGOL and NEWP, refer to “Matching Data Types” later in this section.

Specifying the Access Mode

Data can be exported with an access mode of read-only or read-write. Omitting an access mode assignment is the same as assigning read-only access. The access mode can be specified by both the exporting library and the importing library. The following table shows how the exported access mode and imported access mode combine to determine the actual access mode that is used:

Exported Access Mode	**Imported Access Mode**	**Resulting Access Mode**
READONLY or unspecified	READONLY or unspecified	READONLY
READONLY or unspecified	READWRITE	Objects cannot match.
READWRITE	READONLY or unspecified	READONLY
READWRITE	READWRITE	READWRITE

If the import and export objects do not match because of conflicting access modes, library linkage can still conclude successfully. The LINKLIBRARY result indicates that some requested objects were not available. If the client process attempts to use the imported data object, the client process is discontinued with the error message “OBJECT <object name> ACCESS MODE MISMATCH.” The client process can prevent this error by checking the availability of the data object with the ISVALID function, as described in “Matching Connection Library Objects”.

Data Type Matching

The exported data object and the imported data objects must be of the same or compatible types. Refer to “Matching Data Types” later in this section.

Syntax for Exporting Data

The following ALGOL statements export several data objects. Real array A receives the default access mode of read-only, and integer I is assigned an access mode of read-write.

```
TYPE CONNECTION BLOCK CL_TYPE;
   BEGIN
   REAL ARRAY A [0:10];
   INTEGER I;
   EVENT E1;

   EXPORT A (LINKCLASS = PROTECTED),
         I AS "I2" (READWRITE),
         E1;
   END;

CL_TYPE SINGLE LIBRARY CL_ONE (INTERFACENAME="CLTESTLIB1.");
```

Alternatively, the EXPORT statement can include a single access mode assignment that affects all the items in the export list, as in the following example:

```
EXPORT [READWRITE] A, I;
```

Syntax for Importing Data

The following ALGOL library statements import the data items that were exported by the previous example:

```
TYPE CONNECTION BLOCK CL_TYPE;
   BEGIN
   IMPORTED ARRAY A2[0] (ACTUALNAME = "A");
   IMPORTED INTEGER I2 (READWRITE);
   IMPORTED EVENT E1;
   END;

CL_TYPE SINGLE LIBRARY CL_ONE (INTERFACENAME="CLTESTLIB1.");
```

Direct, Indirect, and Dynamic Data Provision

Connection libraries must use direct provision for any exported objects; indirect and dynamic provision are not permitted. For definitions of these various types of provision, refer to “Methods of Providing Objects” later in this section.

Limitations on Events

Events cannot be assigned an access mode. Exported events can be used in the same ways as any other event. However, the usage of imported events is restricted in the following ways:

Usage	Restriction
WAIT and WAITANDRESET statements	Imported events can be used in WAIT or WAITANDRESET statements only if the compiler control option WAITIMPORT has been set.
Interrupts	An imported event cannot be attached to an interrupt.
LOCK statements	An imported event cannot be used in statements of the form LOCK (<interlock>, <event>).
Direct I/O	An imported event cannot be used in direct I/O.
Parameters	An imported event cannot be passed as a parameter to a procedure.

If a program violates one of these restrictions, a syntax error results.

Predeclared Objects: THIS and THISCL

ALGOL and NEWP provide two functions called THIS and THISCL. These functions return references to the current connection and the current connection library, respectively.

Referencing the Connection with THIS

The THIS function returns a reference to the connection in which the current procedure is being executed.

THIS Function Example

In the following ALGOL example, the exported procedure CLPROC calls the global procedure OTHERPROC, passing THIS as a parameter. OTHERPROC uses the parameter to assign a value to the connection object called CONN_VAL.

```
TYPE CONNECTION BLOCK CLTYPE;
   FORWARD;

PROCEDURE OTHERPROC(CONNECTION_REF);
   CLTYPE CONNECTION_REF;
   FORWARD;

TYPE CONNECTION BLOCK CLTYPE;
   BEGIN

   INTEGER CONN_VAL;

   PROCEDURE SET_CONN_VAL (I);
```

```
          VALUE I;
          INTEGER I;
          BEGIN
          CONN_VAL:= I;
          END;

       PROCEDURE CLPROC;
          BEGIN
          OTHERPROC (THIS);
          END;

       EXPORT CLPROC;
       END;

    PROCEDURE OTHERPROC(CONN_REF);
      CLTYPE CONN_REF;
       BEGIN
       CONN_REF.SET_CONN_VAL(6);
       END;
```

Note that this example is intended merely to illustrate the syntactical possibilities. In practice, procedure CLPROC could modify CONN_VAL directly, without having to call a global procedure to do it.

Referencing the Connection Library with THISCL

The THISCL function returns a reference to the connection library in which the current procedure is declared. The THISCL function can be used in library attribute queries and assignments. If THISCL is used with a connection attribute (like STATE), the attribute of the current connection is returned. Note that THISCL provides access only to attributes; THISCL cannot be used to access variables within a connection library.

THISCL Function Example

In the following ALGOL example, the exported procedure CLPROC increments the value of the CONNECTIONS attribute of this connection library by the value of INC and returns the value of the STATE attribute to the caller.

```
  TYPE CONNECTION BLOCK CLTYPE;
    BEGIN

    INTEGER PROCEDURE CLPROC(INC);
    VALUE INC; INTEGER INC;
      BEGIN
      THISCL.CONNECTIONS:= THISCL.CONNECTIONS + INC;
      CLPROC:= THISCL.STATE;
      END;
    EXPORT CLPROC;

    END;

  CLTYPE LIBRARY CLLIB(CONNECTIONS=10);
```

Note that the THISCL references occur in the connection block, which is defined before the connection library. Therefore, it is not possible to simply use the connection library name CLLIB instead of THISCL.

Passing Connections as Parameters

There are two methods of passing a connection as a parameter to a procedure: using a formal parameter based on a particular connection block, or using a parameter of type CONNECTION. The following table compares these two methods:

	Formal Parameter Based on Connection Block	**Formal Parameter of Type CONNECTION**
Language Support	ALGOL and NEWP	ALGOL and NEWP
Actual Parameter	Must be based on a connection library type specified by the formal parameter.	Can be a connection of any connection library.
Connection Objects	In ALGOL, provides access to procedures declared in the connection block. In NEWP, provides access to any object declared in the connection block.	Does not provide access to connection objects.
Library Attributes	Provides access only to STATE attribute.	Provides access only to STATE attribute.
Parameters to Library Procedures	Cannot be used as a formal parameter to an imported or exported procedure.	Can be used as a formal parameter to an imported or exported procedure.

Connection Parameters Example

The following NEWP program illustrates both of these methods of passing connections as parameters:

```
BEGIN

TYPE CB1_TYPE = CONNECTION BLOCK
   BEGIN
   PROCEDURE PROC1;
      IMPORTED;
   END;

TYPE CB2_TYPE = CONNECTION BLOCK
   BEGIN
   PROCEDURE PROC2(CLE);
      CONNECTION CLE;
      IMPORTED;
   END;

CB1_TYPE LIBRARY LIB1(LIBACCESS = BYTITLE,
260                  CONNECTIONS = 9,
270                  TITLE = "OBJECT/LIB1.");
280
290 CB2_TYPE LIBRARY LIB2(LIBACCESS = BYTITLE,
300                  CONNECTIONS = 3,
310                  TITLE = "OBJECT/LIB2.");
320
330 PROCEDURE PROC3(CB);
340       CB1_TYPE CB;
350    BEGIN
360    INTEGER I;
370    I := LIBRARY(CB).STATE;
380    IF I = VALUE(LINKED) THEN
390       CB.PROC1;
400    END;
410
420 PROCEDURE PROC4(CN);
430       CONNECTION CN;
440    BEGIN
450    INTEGER I;
460    I := CN.STATE;
470    END;
480
510 LINKLIBRARY(LIB1[0]);
520 LINKLIBRARY(LIB2[0]);
530 PROC3(LIB1[0]);
540 PROC4(LIB1[0]);
550 PROC4(LIB2[0]);
560 LIB2[0].PROC2(LIB1[0]);
570
580 END.
```

Explanation of Example: Connection Block Parameter

In this example, procedure PROC3 (at line 330) declares a parameter CB that is based on connection block CB1_TYPE. As a result, statements in PROC3 are able to read the STATE attribute of the connection, and to initiate procedure PROC1 (which is declared in CB1_TYPE). However, the statement at line 530 that invokes PROC3 must pass a connection of a library (LIB1) that is based on CB1_TYPE. This statement could not have passed a connection of LIB2, because LIB2 is based on a different connection block type.

Explanation of Example: CONNECTION Parameter

Procedure PROC4 (at line 420) declares a parameter of type CONNECTION. As a result, this procedure cannot include any statements invoking procedures declared by the connection. On the other hand, PROC4 can receive connections from libraries based on different connection blocks. Thus, the statements at lines 540 and 550 pass connections of LIB1 and LIB2, which are based on two different connection block types.

Restrictions on OWN Objects in Connection Libraries

There are some restrictions on the use of OWN objects in connection libraries. These same restrictions apply to server libraries. Refer to “Restrictions on OWN Objects in Server Libraries” earlier in this section.

Linking Connection Libraries, Server Libraries, and Client Programs

So far, this section has discussed only linkages between client libraries and server libraries, and linkages between connection libraries and other connection libraries. However, linkages can also be established between connection libraries and server libraries or client libraries. The following pages briefly illustrate the use of the READYCL and LINKLIBRARY functions for:

- Linking a client library to a connection library
- Linking a connection library to a server library
- Linking a process to a program that includes both a server library and one or more connection libraries
- Directly linking a process to a combination of client libraries, server libraries, and connection libraries

Linking a Client Library to a Connection Library

Client Library Declarations

In this context, a client program can be defined as a program that links to a connection library through a client library declaration rather than a connection block and connection library declaration.

INTERFACENAME Attribute

A client library can link to a connection library in much the same way as to a server library. The only difference is that, when linking to connection libraries, the INTERFACENAME attribute specifies the connection library to which to link.

The INTERFACENAME attribute does not replace the LIBACCESS, TITLE, and FUNCTIONNAME attributes. The latter attributes serve their usual purpose of identifying the library program to which to link. The INTERFACENAME attribute supplements this information by identifying a particular connection library within the connection library program.

INTERFACENAME Default Value

If the client program does not explicitly assign INTERFACENAME to the client library, then the INTNAME value is used as the default for INTERFACENAME; if INTNAME is also not assigned, then the library identifier is used.

Types of Linkage

A client program can link to a connection library through explicit linkage, implicit linkage, or direct linkage.

Example

The following ALGOL statements reside in a client program. These statements declare a client library CL1 and initiate explicit linkage for that library:

```
LIBRARY CL1 (LIBACCESS = BYTITLE, TITLE = "OBJECT/CLTEST.",
             INTERFACENAME = "CLTEST.");

PROCEDURE PROC1;
   LIBRARY CL1;

RSLT:= LINKLIBRARY (CL1);
```

The following statements reside in OBJECT/CLTEST, the connection library program that is being linked to:

```
TYPE CONNECTION BLOCK TEST1;
   BEGIN

   PROCEDURE PROC1;
      BEGIN
      % Procedure body statements
      END;

   EXPORT PROC1;

   END;

TEST1 LIBRARY CL1 (INTERFACENAME="CLTEST.", CONNECTIONS = 2);

RYRSLT:= READYCL (CL1);
```

In the preceding example, the READYCL function is necessary so that the LINKLIBRARY function in the client program can succeed.

Linking a Connection Library to a Server Library

Example

The following ALGOL statements declare a connection library and then execute a LINKLIBRARY function to link to a server library:

```
TYPE CONNECTION BLOCK TEST1;
   BEGIN
   PROCEDURE PROC2;
      IMPORTED;
   END;

TEST1 LIBRARY CL1 (LIBACCESS = BYTITLE, TITLE = "OBJECT/SERVLIB.",
                   CONNECTIONS = 3);

RSLT:= LINKLIBRARY (CL1[0]);
```

The library program OBJECT/SERVLIB referenced in the preceding example can be a server library program. In this case, it is not necessary to specify an INTERFACENAME because a server library program can contain no more than one server library.

Server Library Linked to by Connection Library

Such a server library might contain ALGOL statements such as the following:

```
PROCEDURE PROC2;
  BEGIN
  % Procedure body statements
  END;

EXPORT PROC2;

FREEZE (TEMPORARY);
```

Thus, connection 0 of connection library CL1 in the first example imports procedure PROC2 from the server library in the second example.

Linking to a Connection Library in a Server Library Program

A server library program can also contain one or more connection libraries. Hereafter, we shall refer to such a library program as a *composite library.* A process that initiates linkage to such a library program, either through the LINKLIBRARY function or by accessing an imported object, is referred to as the *linking process.*

INTERFACENAME and Composite Libraries

The INTERFACENAME specified by the linking process determines whether the linkage is made to the server library or one of the connection libraries, as follows:

- If the linking process specifies an INTERFACENAME that matches one of the eligible connection libraries in the composite program, then the system attempts to create the linkage to that connection library. For implicit or explicit linkage attempts, only connection libraries that have been readied by a READYCL statement are eligible. For direct linkage attempts, all connection libraries in the composite program are eligible.
- If the linking process specifies an INTERFACENAME that does not match any of the eligible connection libraries in the composite program, then the system creates the linkage to the server library.
- If the linking process does not specify an INTERFACENAME, the system applies the default value for INTERFACENAME and then makes one of the two determinations listed previously. The default value of INTERFACENAME is inherited from the INTNAME attribute; if INTNAME is not assigned, then INTERFACENAME inherits the library identifier as its value.

Composite Library Example

The following is an example of an ALGOL composite library program:

```
BEGIN

PROCEDURE DOSTUFF1;
   BEGIN
   % Procedure body statements
   END;

TYPE CONNECTION BLOCK TEST1;
   BEGIN
   PROCEDURE DOSTUFF2;
      BEGIN
      % Procedure body statements
      END;
   EXPORT DOSTUFF2;
   END;

TEST1 LIBRARY CL1 (INTERFACENAME="CONLIB.");

RSLT:= READYCL (CL1);

EXPORT DOSTUFF1;

FREEZE (PERMANENT);
END.
```

In this library program, the server library exports procedure DOSTUFF1, and the connection library CL1 exports procedure DOSTUFF2. Suppose that this library program is titled OBJECT/TESTLIB, and that the following SL command has been used to define a function name mapping:

```
SL TESTLIB = OBJECT/TESTLIB
```

Examples of Linking to the Composite Library

The following are the results of various sorts of library declarations in the linking process, and the results of linkage attempts that use those declarations:

- LIBRARY LIB1(LIBACCESS = BYFUNCTION, FUNCTIONNAME = "TESTLIB.", INTERFACENAME = "WRONGNAME.");

  Because *WRONGNAME* does not match the INTERFACENAME of the connection library in OBJECT/TESTLIB, the linking process is linked to the server library.

- LIBRARY LIB1(LIBACCESS = BYFUNCTION, FUNCTIONNAME = "CONLIB.");

  In this example, the FUNCTIONNAME attribute is wrongly assigned a value corresponding to the INTERFACENAME of the connection library instead of the SL function name of the library program. Because of this mistake, library linkage fails.

- LIBRARY LIB1(LIBACCESS = BYTITLE, TITLE = "OBJECT/TESTLIB.");

  The system attempts to link the process to the library program OBJECT/TESTLIB. Because OBJECT/TESTLIB includes a connection library, the system inspects the INTERFACENAME specified by the linking process.

  Because the linking process did not assign the INTERFACENAME or INTNAME attributes, INTERFACENAME inherits the library identifier, which is LIB1. Because there is no connection library with an INTERFACENAME of LIB1 in OBJECT/TESTLIB, the system links the process to the server library.

- LIBRARY LIB1(LIBACCESS = BYTITLE, TITLE = "OBJECT/TESTLIB.", INTERFACENAME = "CONLIB.");

  The system attempts to link the process to the library program OBJECT/TESTLIB. Because OBJECT/TESTLIB includes a connection library, the system inspects the INTERFACENAME specified by the linking process. Because the INTERFACENAME matches the connection library in OBJECT/TESTLIB, the system links the process to the connection library.

Directly Linking Client, Server, and Connection Libraries

Direct linkage is a method of linking libraries by way of their library identifiers or task variables. Direct linkage was introduced previously in this section under "Directly Linking to Server Libraries" and "Directly Linking Connection Libraries."

LINKLIBRARY Function for Direct Linkage

The direct form of the LINKLIBRARY function is available in both ALGOL and NEWP. The following is an example of such a LINKLIBRARY call:

```
LINKLIBRARY(LIB1, LIBRARY = LIB2);
```

In this example:

- LIB1 must be the library identifier from a client library declaration or a connection library declaration. If LIB1 is a connection library identifier, it can optionally include a connection index (for example LIB1[3]).
- LIB2 must be a connection library identifier (with or without a connection index), a client library identifier, or a task variable. If a task variable is used, it must be the task variable of a frozen server library process.

Direct Linkage Combinations

Thus, depending on the types of identifiers you specify as the first and second parameters, you can establish any of the following direct linkages:

- Client library to server library
- Client library to connection library
- Connection library to connection library
- Connection library to server library
- Connection library to client library

Using Library Attributes

Library attributes specify properties of a particular library.

Server Libraries and Client Libraries

For server libraries, most attributes are specified in the client program. Such attributes can be included in the library declaration, or assigned to the library identifier later in the client program. However, the CHANGE attribute of a library is specified either in the client program, in an EXPORTLIBRARY statement in the server library program, or both.

Connection Libraries

For connection libraries, most attributes can be assigned in either or both of the programs joined by a connection. The attributes can be assigned in the connection library declaration, or assigned to the library identifier later in the program. The STATE attribute is unique in that it applies to a single connection rather than to an entire connection library.

Properties

The following pages describe all the library attributes. For each attribute, the following properties are described:

- Kind

  This property indicates the type of library to which the attribute applies.

  For library attributes that control library linkage, the Kind value indicates the library declaration to which the attribute is assigned, rather than the matching library that is being linked to.

  One or more of the following Kind values are listed for each library attribute:

  – Client Libraries. This value indicates library attributes that are used within the client program. For example, the FUNCTIONNAME attribute can be assigned to a client library to indicate the library program being linked to.

  – Server Libraries. This value indicates library attributes that are specified for a server library inside the server library program itself. For example, an ALGOL server library program can use the EXPORTLIBRARY statement to assign a CHANGE attribute to the server library.

  – Connection Libraries. This value indicates library attributes read or assigned to a connection library by the connection library program.

  – Single Connections. This value indicates library attributes that apply to a single connection of a connection library.

- Type

  This property indicates what type of object can be assigned to the attribute, or what type of object the attribute value can be read into. Examples are integer, Boolean, EBCDIC string, procedure, and mnemonic. Mnemonics are numeric values that can be represented in a program by the VALUE function and a mnemonic identifier.

Note that attribute types of EBCDIC string are pointer expressions in NEWP, but can be either pointer expressions or true EBCDIC strings in ALGOL.

- Read

  This property indicates whether, and when, the value of the attribute can be read.

- Write

  This property indicates whether, and when, the value of the attribute can be written.

  - While not linked.   The attribute can be written only before library linkage or after library delinkage. Most library attributes, if writeable, fall into this category.
  - While not ready.   The attribute can be written only before the library is readied by a READYCL statement or after the library is unreadied by an UNREADYCL statement.
  - Anytime.   The attribute can be written even while the library is linked.

- Default

  This property indicates the default value, if any.

These property descriptions are followed by one or more of the following headings:

- Explanation

  Text under this heading explains the basic function of the attribute. For attributes that apply to both server libraries and connection libraries, this text explains only the features that are common to both types of libraries.

- Client Libraries

  For attributes that apply to both client libraries and connection libraries, this text explains features that apply only to client libraries.

- Server Libraries

  For attributes that apply to both server libraries and connection libraries, this text explains features that apply only to server libraries.

- Connection Libraries

  For attributes that apply to both server libraries and connection libraries, this text explains features that apply only to connection libraries.

- COBOL74 Considerations

  This text explains features that apply only to COBOL74 libraries. Note that such features do not apply to COBOL85 libraries.

Note: *The library attributes described in the following pages are those that are applicable to a client program, server library program, or connection library program. Libraries can have additional, security-related attributes that are assigned by the operating system or by an SL (Support Library) system command. For descriptions of those attributes, refer to "Security Considerations for Libraries" later in this section.*

APPROVAL

Property	Value
Kind	Connection library
Type	Procedure (see below)
Read	Never
Write	Only in connection library declaration
Default	See below

Explanation

The APPROVAL attribute specifies a procedure that can prevent linkage from occurring if the matching library is unauthorized. Additionally, the APPROVAL procedure can select the connection number to be used.

The person who implements the library program is responsible for writing this procedure (hereafter referred to as the APPROVAL procedure).

If there is an APPROVAL procedure for a connection library, the APPROVAL procedure must be declared outside the connection block for that library.

The APPROVAL attribute can be assigned to the connection library on either side of a connection. That is, an APPROVAL attribute can be specified in a connection library declaration regardless of whether that connection library is to be used in READYCL or LINKLIBRARY functions.

If both ends of a connection specify APPROVAL procedures, then the APPROVAL procedure declared by the process requesting the LINKLIBRARY is executed first, followed by the APPROVAL procedure declared by the process that performed the READYCL. Both APPROVAL procedures can optionally select a connection index; each connection index applies to the side of the connection that executes the APPROVAL procedure.

If an error or fault occurs in an APPROVAL procedure, the system prevents the linkage from completing. However, neither the linking process nor the library being linked to incurs an error.

Default

If no APPROVAL procedure is specified, then permission for the linkage is granted and the operating system selects the connection to use for the linkage.

MYSELF Task Variable

Currently, when the MYSELF task variable is used inside an APPROVAL procedure, MYSELF refers to the process that initiated the linkage. This process is not necessarily the declarer of either library involved in the linkage.

Note: *The meaning of the MYSELF task variable in APPROVAL procedures is subject to change in future software releases. Therefore, you should avoid using MYSELF in APPROVAL procedures.*

The OWNER parameter to the APPROVAL procedure is also a task variable, as defined later under this heading. In most cases, you can use OWNER for the same purposes as MYSELF (that is, for checking the USERCODE and other attributes of the other library process).

Direct Library Linkages

If a process uses direct linkage to link two libraries, the system does not execute the APPROVAL procedure for either library unless it is necessary to resolve the connection index. That is, the system executes the APPROVAL procedure for either library only if that library supports multiple connections and no connection index was specified in the LINKLIBRARY statement. For information about direct linkage, refer to "Directly Linking Connection Libraries," earlier in this section.

Initiating Library Linkages from the APPROVAL Procedure

In general, the system does not permit APPROVAL procedures to execute statements that cause a new library linkage to occur; such statements result in a failed linkage attempt. However, the system *does* permit APPROVAL procedures to link to certain system libraries such as GENERALSUPPORT and MCPSUPPORT. When a statement in an APPROVAL procedure links to a system library, the system does not execute any CHANGE or APPROVAL procedure that the linking process has declared for the system library itself.

Parameters and Return Value

The system passes parameters to the APPROVAL procedure that identify the other process involved in the linkage and the LIBPARAMETER, if any, specified by the linking process. The APPROVAL procedure returns a real result that indicates whether the linkage is to be allowed and, optionally, indicates which connection to use on the responding side.

The APPROVAL procedure must have the following form:

```
REAL PROCEDURE APPROVAL (OWNER, LIBPAR, LEN, WAIT);
   VALUE        LEN, WAIT;
   TASK         OWNER;
   EBCDIC ARRAY LIBPAR[*];
   INTEGER      LEN;
   BOOLEAN      WAIT;
```

The APPROVAL procedure should read, but not modify, the procedure parameters. The following are the meanings of these parameters:

Parameter	Meaning
OWNER	The task variable of the process that declared the connection library process being linked to. For an APPROVAL procedure declared by the connection library that is being linked to, the OWNER parameter is the task variable of the process that declared the connection library that initiated the linkage. The APPROVAL procedure can read (but cannot modify) the task attributes of this task variable.
LIBPAR	The LIBPARAMETER library attribute value of the requesting connection library.
LEN	The length of the LIBPARAMETER value in bytes.
WAIT	The value of the WAITFORFILE parameter of the LINKLIBRARY function. The APPROVAL procedure can use this information in cases where a delay is necessary before a linkage can be approved. In such a case, • If WAIT = TRUE, then the APPROVAL procedure should wait as long as necessary before deciding to approve the linkage. • If WAIT = FALSE, then the APPROVAL procedure should exit immediately and return an indication that the permission to link was denied.

The result of the APPROVAL procedure is composed of the following fields:

Field	Value and Meaning
[47:04]	Permission to link. This field must be assigned a nonzero value. The following are the possible values and their meanings: 1 Permission granted. The number of the connection to use is stored in field [38:39]. 2 Permission granted. If the LINKLIBRARY statement specifies a connection index for this library, the connection with that index is used. Otherwise, the operating system selects the connection to use for the linkage, and increases the value of the CONNECTIONS attribute if necessary. 3 Permission denied. An error number is stored in field [38:39].
[38:39]	If field [47:04] = 1, this field stores the number of the connection to be used. The number specified must be of a connection that is not currently in use. If the LINKLIBRARY statement specifies a connection index for this library, the number specified in [38:39] must match the connection index specified in the LINKLIBRARY statement. This value must not exceed the value of the CONNECTIONS attribute. If field [47:04] = 2, this field has no meaning. If field [47:04] = 3, this field stores an error number. If the error number is a value from 501 to 1000, then the LINKLIBRARY function returns the negative of the error number to the linking process. If the error number falls outside this range, then LINKLIBRARY returns a value of –500.

AUTOLINK

Property	Value
Kind	Client library or connection library
Type	Boolean
Read	Anytime
Write	Anytime
Default	Client libraries: TRUE
	Connection libraries: FALSE

Explanation

The AUTOLINK library attribute specifies whether a library can be linked to implicitly. Implicit linkage occurs when a process uses an imported object without having first executed a LINKLIBRARY function. If AUTOLINK is TRUE, the system automatically attempts to link the importing process to the library. If AUTOLINK is FALSE, the importing process incurs a fatal error and terminates.

Setting AUTOLINK to TRUE is useful if your program uses LINKLIBRARY to initiate the linkage and you want to catch any unexpected implicit linkages.

Client Libraries

The AUTOLINK attribute defaults to TRUE, but can be explicitly assigned a value of either TRUE or FALSE.

Connection Libraries

The AUTOLINK attribute defaults to FALSE, but can be explicitly assigned a value of either TRUE or FALSE.

CHANGE

Property	Value
Kind	Server library, client library, or connection library
Type	Procedure (see below)
Read	Never
Write	Server libraries: only in an EXPORTLIBRARY statement in the server library program
	Client libraries: only in the library declaration
	Connection libraries: only in the connection library declaration
Default	Not applicable

Explanation

The CHANGE attribute specifies a procedure that notifies the library process of any changes in the status of a particular connection. The person who implements the library program is responsible for writing this procedure (hereafter referred to as the CHANGE procedure).

The system passes parameters to the CHANGE procedure that identify the new state of the connection and the reason that the state is changing. The CHANGE procedure can notify the library process of the changed status by modifying global objects in the library.

The connection does not fully transition to the state reported by the CHANGE procedure until the CHANGE procedure exits. Therefore, it is important not to include statements in the CHANGE procedure that could cause the CHANGE procedure to enter any prolonged waiting state.

If an error or fault occurs in a CHANGE procedure and the new state is LINKING or LINKED, the system prevents the change in connection state from completing. Further, the error or fault is propagated to the requesting process (the process that initiated the linkage or delinkage that is in progress). If the requesting process does not include code to handle the fault, the system discontinues the requesting process.

Library Linkages

In general, the system does not permit CHANGE procedures to execute statements that cause a new library linkage to occur; such statements result in a failed linkage attempt. However, the system *does* permit CHANGE procedures to link to certain system libraries such as GENERALSUPPORT and MCPSUPPORT. When a statement in a CHANGE procedure links to a system library, the system does not execute any CHANGE or APPROVAL procedure that the linking process has declared for the system library itself.

Client Libraries and Server Libraries

A CHANGE procedure can be specified inside the server library program or inside the client program.

For server libraries, the CHANGE attribute can be assigned only in an EXPORTLIBRARY statement. For example, the following ALGOL statement specifies that when the current block freezes as a library, the procedure CHG1 should be used as the CHANGE procedure for that library:

```
EXPORTLIBRARY (CHANGE = CHG1);
```

When CHANGE is specified for a server library, the CHANGE procedure is executed whenever any client process finishes linking to, or starts delinking from, the library.

For client programs, the CHANGE attribute can be assigned only in a library declaration. In this case, the CHANGE procedure is executed whenever that particular client program finishes linking to or starts delinking from the library.

For server libraries and client libraries, the system invokes the CHANGE procedure only when the connection state changes to LINKED or to DELINKING. In some cases, it is possible that the system will invoke the CHANGE procedure more than once for the DELINKING state.

Note: *A server library* cannot *determine its current number of clients by reading the LIBRARYUSERS task attribute inside the CHANGE procedure, because the MYSELF task variable does not refer to the server library process. Refer to the discussion of the MYSELF task variable later under this heading.*

Connection Libraries

The CHANGE attribute can be assigned only in a connection library declaration, and the CHANGE procedure must be declared inside the connection library block.

The CHANGE attribute can be assigned to the connection library on either side of a connection. That is, a CHANGE attribute can be specified in a connection library declaration regardless of whether that connection library is to be used in READYCL or LINKLIBRARY functions.

When a linkage attempt for a connection library fails, it does not go through the normal state progression from NOTLINKED to LINKING to LINKED to DELINKING and back to NOTLINKED. Instead, the connection library either does not undergo state transitions or traverses the states from LINKING to DELINKING and back to NOTLINKED.

MYSELF Task Variable

Currently, when the MYSELF task variable is used inside a CHANGE procedure, MYSELF refers to the process initiating the linkage or delinkage. This process is also referenced by the task-valued parameter ACTOR, which is described later under this heading.

Note: *The meaning of the MYSELF task variable in CHANGE procedures is subject to change in future software releases. Therefore, it is preferable to use the ACTOR parameter instead of MYSELF in CHANGE procedures.*

Parameters and Return Value

The CHANGE procedure must have the following form:

```
PROCEDURE CHANGE (CONN_INDEX, NEW_STATE, REASON, ACTOR, IMDSED);
   VALUE   CONN_INDEX, NEW_STATE, REASON, IMDSED;
   INTEGER CONN_INDEX, NEW_STATE, REASON;
   TASK    ACTOR;
   BOOLEAN IMDSED;
```

The CHANGE procedure should read, but not modify, the procedure parameters. The following are the meanings of these parameters:

Parameter	**Meaning**
CONN_INDEX	Server libraries: This parameter has no meaning. Connection libraries: The index of the connection that has changed state. The index can be different on the requesting library side than on the responding library side. CONN_INDEX reflects the value from the point of view of the requesting or responding connection library to which this CHANGE procedure is assigned.
NEW_STATE	The value that the STATE attribute now returns. The following descriptions include the STATE attribute mnemonic corresponding to each numeric value: • 1 (NOTLINKED) Server libraries: Value is never returned. Connection libraries: The connection specified by CONN_INDEX has completed delinkage and imported library objects cannot be used. If any connection libraries are referenced, the CHANGE procedure causes a fault and ceases processing. • 2 (LINKING) Server libraries: Value is never returned. Connection libraries: A linkage is initiated, but not complete, for the connection index specified by CONN_INDEX. The APPROVAL procedure, if any, is called. However, no linkages have been established between export and import objects and imported library objects cannot be used. If any connection libraries are referenced, the CHANGE procedure causes a fault and ceases processing. The linkage fails with error -23 (CHANGE procedure faulted). • 3 (LINKED) Server libraries: A client process has completed linking to the library. Connection libraries: The connection specified by CONN_INDEX has completed linkage.

Parameter	Meaning
NEW_STATE *(cont.)*	• 4 (DELINKING) Server libraries: A client process has started delinking from the library. Connection libraries: Delinkage is initiated, but not complete, for the connection index specified by CONN_INDEX.
REASON	The reason for the state change.
Field [03:03]	If the NEW_STATE value is 2 (linking) or 4 (delinking), then the following are the possible values of this field and their meanings: • 0 The change resulted from a LINKLIBRARY, DELINKLIBRARY, or CANCEL function. • 1 The change resulted from implicit library linkage or from delinkage due to block exit. If the NEW_STATE value is 1 (not linked), then the following are the possible values of this field and their meanings: • 0 Linkage removed by DELINKLIBRARY or CANCEL. • 1 Linkage terminated due to block exit. • 2 Linkage attempt failed because templates didn't match. • 3 Linkage attempt failed because of a CHANGE procedure fault. • 4 An MCP error occurred.
Field [00:01]	The locality bit. The following are the possible values and meanings: • 0 This CHANGE procedure is the CHANGE procedure for the process initiating the linkage or delinkage. • 1 This CHANGE procedure is the CHANGE procedure for the library being linked to or delinked from.
ACTOR	The task variable for the process whose action caused the state change. If the new state is 2 (linking) or 3 (linked), the ACTOR process is the one that initiated the link, either by calling LINKLIBRARY or by implicit linkage. Statements in the CHANGE procedure can read the task attributes of the ACTOR parameter, but cannot modify any of these attributes. Attempts to modify such task attributes result in the warning message *MAY NOT MODIFY THIS TASK*, and the attribute values are not changed.

Parameter	Meaning
IMDSED	• TRUE The ACTOR process is terminating abnormally, either due to a program fault or a DS (Discontinue) system command. The ACTOR process is being delinked because of this abnormal termination. • FALSE The ACTOR process is not terminating abnormally.

CLUSAGE

Property	Value
Kind	Connection library
Type	Integer
Read	Anytime
Write	Never
Default	None

Explanation

The CLUSAGE library attribute returns the number of active connections from a connection library.

CONNECTIONS

Property	Value
Kind	Connection library
Type	Integer
Read	Anytime
Write	Anytime
Default	1

Explanation

The CONNECTIONS library attribute specifies the number of connections this connection library is currently prepared to handle.

The maximum CONNECTIONS value that can be included in a connection library declaration is 1048575. If CONNECTIONS is assigned a larger value in the declaration, the compiler generates a syntax error.

However, it might be possible to assign CONNECTIONS a larger value in statements outside the declaration. The actual maximum value is machine-dependent, and corresponds to the maximum number of rows supported for a segmented array on the system. Attempting to assign CONNECTIONS a larger value results in a run-time error message that states the maximum value allowed for that machine. The error is not fatal, but the CONNECTIONS assignment is not made.

Because connection indexes are zero-relative, the highest possible connection index is one less than the value of CONNECTIONS. For example, CONNECTIONS = 10 allows the use of connections numbered 0 through 9.

The CONNECTIONS value can be increased, but not decreased, by assignment statements in a program. If CONNECTIONS is decreased, an attribute error results.

The CONNECTIONS value might also increase because of library linkage. Refer to "APPROVAL" earlier in this section.

When a program attempts to set the CONNECTIONS attribute to an erroneous value that is less than 1, the CONNECTIONS attribute is set to 1 and the following system message is given:

```
  LIBRARY ATTRIBUTE ERROR: CONNECTIONS MUST EXCEED ZERO
```

If CONNECTIONS is not specified in the connection library declaration, then CONNECTIONS defaults to 1, but can be increased later.

If the SINGLE modifier is used in the connection library declaration, then only a single connection is allowed. In this case, the CONNECTIONS attribute is treated as read-only. Any attempt to modify it results in an attribute error.

DELINKEVENT

Property	Value
Kind	Connection library
Type	Event
Read	Anytime
Write	Anytime (see explanation)
Default	None

Explanation

The DELINKEVENT library attribute accesses a predeclared event that is associated with a connection library.

The DELINKEVENT attribute is caused when the number of active connections from a connection library is decremented to 0 (zero). The attribute is reset whenever a new connection is created.

Write

A process can cause or reset the DELINKEVENT attribute at any time. However, a process can never assign an event variable to the DELINKEVENT attribute.

FUNCTIONNAME

Property	Value
Kind	Client library or connection library
Type	EBCDIC string
Read	Anytime
Write	Client libraries: While not linked
	Connection libraries: While not ready and no connections are linked
Default	Value of INTNAME library attribute

Explanation

If the LIBACCESS library attribute is set to BYFUNCTION, then the FUNCTIONNAME library attribute identifies the function name of the library program that is to be linked to. The function name is associated with a library object code file title by the SL (Support Library) system command. You can also establish SL function names programmatically through the DCALGOL *SETSTATUS* function.

The operating system stores the mappings between SL function names and library programs and links to the appropriate code file if a function name is used. The function name makes it possible to change to a differently titled library program without having to recompile all the client programs that use the library.

Not all library programs have associated function names, although any library program can be assigned one or more function names by an SL command. A library program that has an associated function name is called a *support library*.

An SL command can do any of the following:

- Display the current function names and their associated object code files.
- Assign a particular library object code file to a function name, without affecting any programs that are currently using the library. In some cases, the change does not take effect until the current invocation of the library thaws and resumes execution. The associations between function names and library object code files survive a halt/load or a CM (Change MCP) system command.
- Create a new function name and assign the object code file that is initially associated with it.
- Delete an existing function name, without affecting any processes that are currently using the library. In some cases, the deletion is denied if a frozen invocation of the library currently exists.

Client programs can access a library directly by object code file title even if a function name has been defined for the library. However, undesirable side effects can result if the first client program to access a library does so by object code file title instead of by function name.

For example, if a library has a SHARING value of SHAREDBYALL and the first client program accesses the library by object code file title, then the library process inherits several task attributes of the client program, including USERCODE, FAMILY, and SOURCESTATION. However, if the same library is first accessed by function name, these task attributes are not inherited.

In addition, system libraries must first be accessed by function name, so they receive their special privileges. For a discussion of this and other security issues related to the SL command, refer to "Security Considerations for Libraries" later in this section.

The FUNCTIONNAME library attribute can be specified in a library declaration, a library attribute assignment statement, or a LINKLIBRARY function.

Direct Library Linkages

The FUNCTIONNAME library attribute has no effect when direct library linkage is used. Refer to "Directly Linking to Server Libraries" and "Directly Linking Connection Libraries," earlier in this section.

Connection Libraries

For connection libraries, the FUNCTIONNAME library attribute has meaning only for the library on the requesting side of the linkage attempt. If the requesting library has a LIBACCESS value of BYFUNCTION, then the FUNCTIONNAME value of the requesting library identifies the library program that contains the responding library.

The FUNCTIONNAME attribute affects the entire connection library. If you want to link the various connections BYFUNCTION to different library programs, you can specify a different FUNCTIONNAME attribute in each LINKLIBRARY request. The action of specifying a different FUNCTIONNAME causes a value override to the BYFUNCTION value for the LIBACCESS attribute for the LINKLIBRARY request. For example, the following ALGOL statements link connections 0 and 1 of connection library CL1 to different library programs:

```
CLTYPE LIBRARY CL1(INTERFACENAME = "CLF.", CONNECTIONS = 2);
REPLACE FNAME BY "DELTA.";
RSLT:= LINKLIBRARY (CL1[0], FUNCTIONNAME = FNAME);
REPLACE FNAME BY "GAMMA.";
RSLT:= LINKLIBRARY (CL1[1], FUNCTIONNAME = FNAME);
```

In order for linkage to complete, the requesting connection library and the responding connection library must also have matching INTERFACENAME library attribute values.

INTERFACENAME

Property	Value
Kind	Client library or connection library
Type	EBCDIC string
Read	Anytime
Write	Client libraries: While not linked
	Connection libraries: While not ready and no connections are linked
Default	Value of INTNAME library attribute

Explanation

The INTERFACENAME library attribute identifies a particular connection library in a connection library program.

The INTERFACENAME library attribute does not identify the program that contains the connection library. To identify that program, you must use the LIBACCESS library attribute, usually in combination with the TITLE library attribute or the FUNCTIONNAME library attribute.

Direct Library Linkages

The INTERFACENAME library attribute has no effect when direct library linkage is used. Refer to “Directly Linking to Server Libraries” and “Directly Linking Connection Libraries,” earlier in this section.

Connection Libraries

Connection library programs can potentially contain more than one connection library. To provide unique identification, you should assign a different INTERFACENAME value to each connection library that is declared in the same program.

Client Libraries

You can assign the INTERFACENAME value to a client library. This attribute has no effect if the client library links to a server library. However, if the client library links to a connection library, the INTERFACENAME attribute of the client library specifies the connection library to link to in the connection library program.

Also, if the client library links to a server library program that includes a connection library, and the INTERFACENAME value matches the connection library, then the system links the client library to the connection library instead of to the server library.

INTNAME

Property	Value
Kind	Client library or connection library
Type	EBCDIC string
Read	Anytime
Write	Client libraries: While not linked
	Connection libraries: While not ready and no connections are linked
Default	Library identifier, except in COBOL74; see following discussion

Explanation

The INTNAME library attribute specifies the internal name for the library.

One use of the internal name is in assignments to the LIBRARY task attribute. You can use the LIBRARY task attribute to alter the behavior of client programs. For example, suppose a client program declares a library with an internal name of LIB1, a LIBACCESS value of BYTITLE, and a TITLE value of OBJECT/LIB1. When you run the client program, you can assign the LIBRARY task attribute a value of LIBRARY LIB1(TITLE =OBJECT/OTHERLIB). This has the effect of causing the client program to link to a different library than it otherwise would.

INTNAME also serves as the default value for the FUNCTIONNAME, INTERFACENAME, and TITLE attributes.

COBOL74 Considerations

Because libraries cannot be explicitly declared in COBOL74, the compiler constructs the default INTNAME for a library from the first reference to that library title in the client program. If the title includes multiple nodes separated by slashes (/), the INTNAME is formed from the final node of the title.

The first reference to the library title might be in the CALL statement that invokes the library, or in a CHANGE statement that assigns attributes to the library. If either of the following two statements is the first reference to a library in a COBOL74 client program, the library receives an INTNAME of LIB1:

```
CHANGE ATTRIBUTE FUNCTIONNAME OF "OBJECT/LIB1" TO "TESTSUPPORT.".
CALL "FACT IN OBJECT/LIB1" USING PARAM.
```

To prevent any two libraries in a COBOL74 program from receiving the same INTNAME, you should assign each library a TITLE attribute value that has a different value in the final node.

LIBACCESS

Property	Value
Kind	Client library or connection library
Type	Mnemonic
Read	Anytime
Write	Client libraries: While not linked
	Connection libraries: While no connections are linked
Default	BYTITLE, except in some LINKLIBRARY requests affecting connection libraries; see following discussion

Explanation

The LIBACCESS library attribute specifies the method the linking library uses to identify the library being linked to. The linking library is either a client library or else a connection library that is acting as the requesting side in a linkage attempt.

LIBACCESS has one of the following mnemonic values: BYFUNCTION, BYINITIATOR, or BYTITLE. The following are the effects of these values:

- BYFUNCTION

  The FUNCTIONNAME library attribute specifies the library being linked to. The value of the TITLE library attribute is ignored.

- BYINITIATOR

  This value has meaning only when preceded by one of the following activities:

  – The originating library uses a server library linkage mechanism to link a library, which becomes the initiating process.

  – The originating library initiates a dependent process, which becomes the initiating process.

  – The originating library initiates a link to a connection library through a client library declaration. The connection library contains a local client library declaration, which is used by the initiating process.

  The BYINITIATOR value causes the initiating process to link to the originating library. When the BYINITIATOR value is used, the values of the FUNCTIONNAME and TITLE library attributes are ignored.

Note: *If the process was initiated by the library linkage mechanism, the BYINITIATOR value can be used only while the original linkage exists. If the initiating library delinks from the process, the process uses the BYINITIATOR value, and the WAITFORFILE option is set, the system suspends the process with the message*

```
BYINITIATOR IS SPECIFIED, BUT THE INITIATOR IS NOT A LIBRARY
```

The operator can use the FA (File Attributes) system command to specify the location of the appropriate library, or use the DS (Discontinue) system command to terminate the program.

- BYTITLE

  The TITLE library attribute specifies the library being linked to. The value of the FUNCTIONNAME library attribute is ignored.

Connection Libraries

The LIBACCESS attribute affects the entire connection library. To link the various connections to different library programs, you can use different implicit settings for the LIBACCESS attribute in each LINKLIBRARY request. If a TITLE is specified in the request, a LIBACCESS value of BYTITLE is assumed. If a FUNCTIONNAME is specified in the request, a LIBACCESS value of BYFUNCTION is assumed. For example, the following ALGOL statements link connections 0 and 1 of connection library CL1 to different programs, using different values of LIBACCESS:

```
CLTYPE LIBRARY CL1 (INTERFACENAME = "CLF.", CONNECTIONS = 2);
REPLACE TITL BY "OBJECT/DELTA ON DISK.";
RSLT := LINKLIBRARY (CL1[0], TITLE = TITL);
REPLACE FNAME BY "GAMMA.";
RSLT := LINKLIBRARY (CL1[1], FUNCTIONANME = FNAME);
```

To complete the linkage, the requesting connection library and the responding connection library must have matching INTERFACENAME attribute values.

If the library being linked to is a connection library, then the INTERFACENAME library attribute also plays an essential role in the linkage attempt, regardless of the value of the LIBACCESS library attribute. Refer to “INTERFACENAME” earlier in this section.

Direct Library Linkages

The LIBACCESS library attribute has no effect when direct library linkage is used. Refer to “Directly Linking to Server Libraries” and “Directly Linking Connection Libraries,” earlier in this section.

LIBERROR

Property	Value
Kind	Client library or connection library
Type	Boolean
Read	Anytime
Write	Never
Default	None

Explanation

The LIBERROR library attribute indicates whether the most recent library attribute action for this library incurred an error. If LIBERROR is TRUE, then an error occurred. If LIBERROR is FALSE, then no error occurred.

When a program finishes reading the LIBERROR attribute, the system automatically resets the LIBERROR value to FALSE.

LIBPARAMETER

Property	Value
Kind	Client library or connection library or SINGLE connection
Type	EBCDIC string
Read	Anytime
Write	Client libraries: While not linked
	Connection libraries (SINGLE): While not linked
	Connection libraries (Non-SINGLE): Anytime
	SINGLE connections: While not linked
Default	Null string

Explanation

The LIBPARAMETER library attribute is used to transmit information from the linking library to the selection procedure of the library or to the approval procedure of the connection library being linked to. Selection procedures are used by libraries that use dynamic provision to export objects. For more information, refer to “Dynamic Provision” in this section.

The linking library can be a client library or a connection library. The primary library linked to can be a server library or a connection library. However, the selection procedure in the primary library must select a secondary library that is a server library (not a connection library).

For connection libraries, the system also passes the LIBPARAMETER library or connection attribute of the requesting library to the APPROVAL procedure or procedures. The connection attribute is passed if it is set. If the connection attribute is not set, the library attribute is used. For more information, refer to “APPROVAL” in this section.

The LIBPARAMETER attribute for a single connection cannot be library-equated.

SINGLE

Property	Value
Kind	Connection library
Type	Boolean
Read	Anytime
Write	Never
Default	Not applicable

Explanation

The SINGLE library attribute returns an indication of whether or not the connection library was declared with the SINGLE modifier. The SINGLE modifier specifies that a connection library can have no more than one connection. If SINGLE is TRUE, then the CONNECTIONS attribute cannot be modified.

STATE

Property	Value
Kind	Connection library
Type	Mnemonic
Read	Anytime
Write	Never
Default	Not applicable

Explanation

The STATE library attribute returns the logical state of one of the connections of a connection library.

When a program reads the STATE attribute, the program must specify a connection index unless the connection library was declared with the SINGLE keyword. For example, the following ALGOL statement reads the STATE attribute for connection 3 of connection library CLIB:

```
IF LIBRARY(CLIB[3]).STATE = VALUE(NOTLINKED) THEN...
```

The following are the possible STATE values and their meanings:

Mnemonic Value	Integer Value	Meaning
NOTLINKED	1	The connection is not linked.
LINKING	2	Linkage is initiated but is not complete. The APPROVAL procedure, if any, is called. However, no procedure linkages are established yet.
LINKED	3	The connection is now fully linked.
DELINKING	4	The connection is in the process of being delinked.

The current state of a connection is also automatically passed to the CHANGE procedure, if any. Refer to “CHANGE” earlier in this section.

TITLE

Property	Value
Kind	Client library or connection library
Type	EBCDIC string
Read	Anytime
Write	Client libraries: While not linked
	Connection libraries (SINGLE): While not linked
	Connection libraries (Non-SINGLE): Anytime
Default	Value of INTNAME library attribute, except in COBOL74; see following discussion

Explanation

The TITLE library attribute specifies the object code file title of the library. The TITLE attribute has meaning only if the LIBACCESS library attribute is set to BYTITLE.

Connection Libraries

The TITLE attribute affects the entire connection library. If you want to link the various connections BYTITLE to different library programs, you can specify a different TITLE attribute in each LINKLIBRARY request. The action of specifying a different TITLE causes a value override to the BYTITLE value for the LIBACCESS attribute for the LINKLIBRARY request. For example, the following ALGOL statements link connections 0 and 1 of connection library CL1 to different library programs:

```
CLTYPE LIBRARY CL1(INTERFACENAME = "CLF.", CONNECTIONS = 2);
REPLACE TITL BY "OBJECT/DELTA ON DISK.";
RSLT:= LINKLIBRARY (CL1[0], TITLE = TITL);
REPLACE TITL BY "OBJECT/GAMMA ON DISK.";
RSLT:= LINKLIBRARY (CL1[1], TITLE = TITL);
```

In order for linkage to complete, the requesting connection library and the responding connection library must also have matching INTERFACENAME library attribute values. Refer to “INTERFACENAME” earlier in this section.

COBOL74 Considerations

Because libraries cannot be explicitly declared in COBOL74, the compiler constructs the default title for a library from the first reference to that library in the client program. The first reference to the library title might be in the CALL statement that invokes the library, or in a CHANGE statement that assigns attributes to the library. If either of the following two statements is the first reference to a library in a COBOL74 client program, the library receives a title of OBJECT/LIB1:

```
CHANGE ATTRIBUTE FUNCTIONNAME OF "OBJECT/LIB1" TO "TESTSUPPORT.".
CALL "FACT IN OBJECT/LIB1" USING PARAM.
```

To prevent any two libraries in a COBOL74 program from receiving the same INTNAME, you should give each library a title that has a different value in the final node of the title. Refer to “INTNAME” earlier in this section.

Direct Library Linkages

The TITLE library attribute has no effect when direct library linkage is used. Refer to “Directly Linking to Server Libraries” and “Directly Linking Connection Libraries,” earlier in this section.

Methods of Providing Objects

Libraries provide export objects in one of three ways: *directly, indirectly,* and *dynamically.* The declaration of each export object in the library program specifies which of these provision methods is used. The method chosen depends on whether the export object originates in the library program, or if the library itself imports the object from another library.

Additionally, a series of indirect provisions can result in an object being ultimately provided by the very library that is importing it. This type of provision is said to be circular.

Direct Provision

Direct provision occurs when the library program contains the complete declaration of the object that is named in the export list of the library. For example, if a procedure is exported, the library contains all the statements that make up the procedure.

Direct provision is the only type of provision that is provided by COBOL74 libraries.

Indirect Provision

Indirect provision occurs when the library program exports an object that is, in turn, imported from another library. The system then attempts to link the client process to this second library, which can provide the exported object directly, indirectly, or dynamically. A chain of indirect or dynamic provisions must eventually end in a library that provides the object directly.

Exported Data

Indirect provision can be used only for exported procedures, not for exported data.

Indirect Provision and Server Library Objects

The following is an example of an ALGOL server library that provides procedure PROC1 indirectly, by importing PROC1 from another server library called OBJECT/PROVIDER:

```
LIBRARY OTHERLIB(LIBACCESS = BYTITLE, TITLE = "OBJECT/PROVIDER");

PROCEDURE PROC1;
   LIBRARY OTHERLIB;

EXPORT PROC1;

FREEZE(PERMANENT);
```

Indirect Provision and Connection Library Objects

Server libraries can provide objects indirectly, but connection libraries cannot. That is, a connection library cannot import an object and then export the same object. Further, a connection library cannot use an object that is indirectly provided through a server library.

For example, a connection library could not link to the library in the preceding example and invoke procedure PROC1.

It is possible for a server library to indirectly provide procedures that originate in a connection library. However, to do this, the server library must use procedure references or procedure reference arrays. Procedure references and procedure reference arrays are supported in NEWP and in all varieties of ALGOL.

The following ALGOL server library imports the procedure PROC1 from the connection library with the INTERFACENAME of CLTEST in the program OBJECT/CLTEST. This server library then indirectly exports PROC1 to any client programs that link to this server library:

```
LIBRARY CL1 (LIBACCESS = BYTITLE, TITLE = "OBJECT/CLTEST.",
             INTERFACENAME = "CLTEST.");

PROCEDURE PROC1;
   LIBRARY CL1;

PROCEDURE REFERENCE PREF;

RSLT:= LINKLIBRARY (CL1);
PREF:= PROC1;
EXPORT PREF AS "PROC1";
FREEZE (PERMANENT);
```

Note that this server library exports a procedure reference PREF instead of PROC1 itself. The use of the procedure reference is necessary because this server library imports PROC1 from a connection library. The use of procedure references is the only way to indirectly provide a connection library procedure.

Indirect Provision Compared with Indirect Usage

A program can obtain an effect similar to indirect provision by directly exporting a procedure that includes a statement that uses an imported object. For example, the following ALGOL example imports procedure PROC1 from library LIB1. This program then freezes as a server library, exporting procedure OTHERPROC, which includes an invocation of LIB1.

```
LIBRARY LIB1 (LIBACCESS=BYFUNCTION,FUNCTIONNAME="LIBTEST.",
              INTERFACENAME="CONLIB.");

PROCEDURE PROC1;
   LIBRARY LIB1;

PROCEDURE OTHERPROC;
   BEGIN
   PROC1;
   END;

EXPORT OTHERPROC;
FREEZE (TEMPORARY);
```

In this example, PROC1 could be imported from another server library or from a connection library. The usual restriction on indirect provision of connection library objects does not apply here, because OTHERPROC is not actually linked to PROC1.

Dynamic Provision

Dynamic provision is similar to indirect provision in that it enables a library to export an object that, in turn, was imported from another library. However, dynamic provision allows the primary library to import objects with the same name from multiple secondary libraries. Whenever a client process attempts to import an object with that name, the primary library can dynamically select the version of the object to provide to the client process.

For an example of a library using dynamic provision, refer to “ALGOL Library: OBJECT/SAMPLE/DYNAMICLIB” later in this section.

Restrictions

- The dynamic provision feature is supported only in ALGOL.
- Server libraries can provide objects dynamically, but connection libraries cannot.
- Dynamic provision can be used only for exported procedures, not for exported data.

BY CALLING Clause for Exporting Procedures

Procedures that are exported dynamically include a BY CALLING clause, as in the following example:

```
PROCEDURE READFILE;
    BY CALLING SELECTION;
```

Selection Procedure

The BY CALLING clause specifies the name of a *selection procedure*, which you must declare elsewhere in the library program. Whenever a client process first links to a library, the system checks to see if any objects imported by that client process are provided by the library with a BY CALLING clause. If so, the system invokes the selection procedure. The selection procedure must accept the following two parameters from the system:

- A parameter of type EBCDIC string, which the system uses to pass in the value of the LIBPARAMETER library attribute as specified by the client process. This parameter enables the client process to convey information to the library that might help the library decide from which secondary library to import the object.
- A parameter of type procedure, which the system uses to pass in an MCP procedure. This procedure itself has a parameter, which is a task variable. The selection procedure must invoke the MCP procedure before exiting, and must pass to it the task variable of the secondary library that has been selected. Otherwise, the linking process is discontinued.

Secondary Library

The secondary library that is selected can provide the requested object directly, indirectly, or dynamically. A chain of indirect or dynamic provisions must eventually end in a library that provides the object directly.

Note that the secondary library must be a server library, not a connection library.

Selection Occurs during Linkage

The selection procedure is invoked only at library linkage time. All links to exported objects in the library are resolved during linkage. To cause the selection of a different secondary library after linkage, the client process must first delink from the dynamic library. The client process can then modify the LIBPARAMETER library attribute to request a different secondary library, and relink to the dynamic library.

Circular Provision

Circular Linkage

A *circular library linkage* is a series of library linkages that flow in the same direction and return to the first program in the series. The operating system permits certain types of circular linkages, but disallows other types of circular linkages.

Circular Provision

Similarly, a *circular provision* is the provision of a particular library object through circular library linkage. In other words, a library can indirectly import an object from the same library. However, the import object in the library must ultimately be provided by a different export object in that library.

Compared to Connection Libraries

Circular linkage is an advanced technique that can become quite complex to use. Connection libraries were invented, in part, as a replacement for the use of circular library linkage. If you are designing an application to use circular linkage, you should consider replacing the circular linkage with a pair of connection libraries that export objects to each other. For more information, refer to “Hazards of Circular Connections.”

Nevertheless, circular linkages are still allowed, and connection libraries can even participate in them. The following subsections explain the types of linkages that are considered to be circular, and then list the restrictions that apply to circular linkages.

Understanding Circular Linkage and Circular Provision

The following paragraphs illustrate the nature of circular linkage and circular provision through several examples.

Example 1: Circular Provision of a Single Procedure

Figure 18–1 shows one example of a circular linkage:

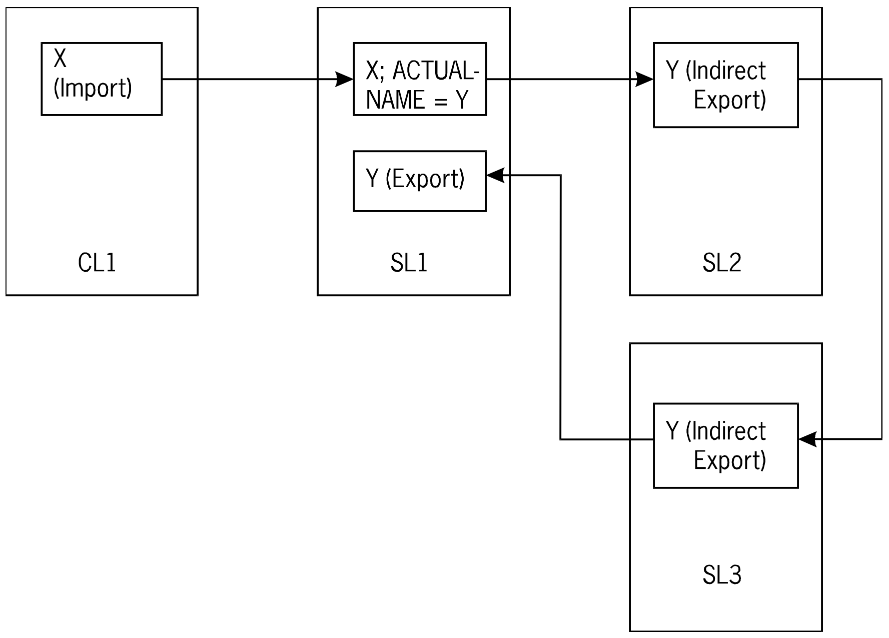

Figure 18–1. Circular Provision of a Single Procedure

In Figure 18–1,

- The arrows point from import procedures to the corresponding export procedures.
- Client program CL1 imports procedure X from server library SL1.
- SL1 provides procedure X indirectly by importing procedure X with an ACTUALNAME of Y from server library SL2.
- SL2 provides procedure Y indirectly by importing procedure Y from server library SL3.
- SL3 provides procedure Y indirectly by importing procedure Y from server library SL1.

In this example, the library linkage is circular because server library SL1 is both the first and the last library in the linkage series.

The provision of procedure X is also circular, because import object X in SL1 is ultimately provided by export object Y in the same library, SL1.

Example 2: Circular Linkage without Circular Provision

The definition of circular library linkages is actually broader than Example 1 would suggest. The operating system might consider a series of linkages to be circular even if no single object is exported through the entire chain. Figure 18–2 shows an example of this type of circular linkage:

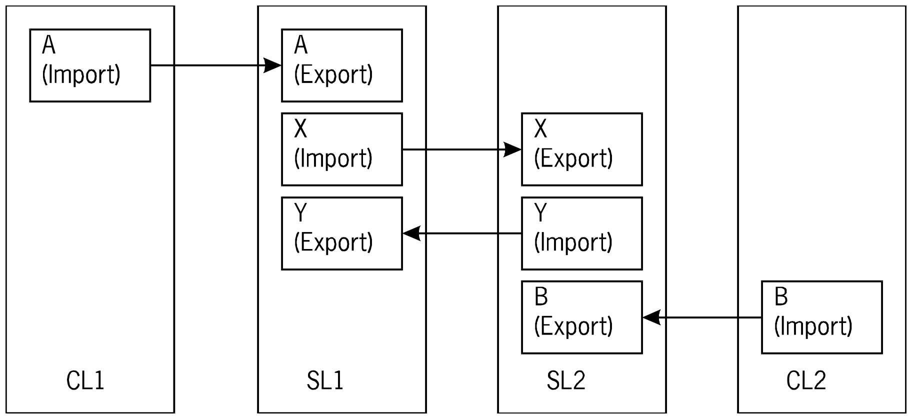

Figure 18–2. Circular Linkage without Circular Provision

In Figure 18–2,

- The arrows point from import procedures to the corresponding export procedures.
- Client library CL1 imports procedure A from server library SL1.
- Server library SL1 imports procedure X from server library SL2.
- Client library CL2 imports procedure B from server library SL2.
- Server library SL2 imports procedure Y from SL1.

In this example, library programs SL1 and SL2 are linked in a way that the operating system considers circular. The linkage is considered circular because each library imports an object from the other library (even though it is not the same object).

On the other hand, this example is not an example of circular provision. Procedures A, B, X, and Y are each provided directly.

Example 3: Circular Linkage Including a Connection Library

A connection library can be included in a circular linkage. Consider the following example:

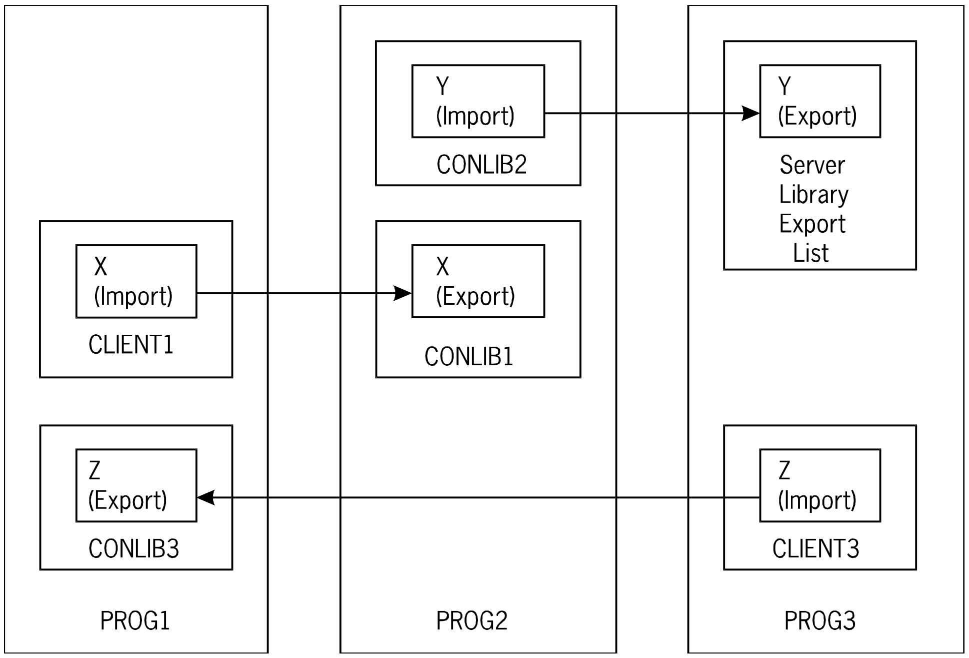

Figure 18–3. Circular Linkage Including a Connection Library

In Figure 18–3,

- The arrows point from import procedures to the corresponding export procedures.
- Program PROG1 declares a client library CLIENT1, which imports procedure X from connection library CONLIB1 in program PROG2.
- Program PROG2 contains two connection libraries: CONLIB1 and CONLIB2. Connection library CONLIB1 exports procedure X to CLIENT1 in PROG1. Connection library CONLIB2 imports procedure Y from server library program PROG3.
- PROG3 includes a client library CLIENT3, which imports procedure Z from connection library CONLIB3 in program PROG1.

Note that, in this example, the linkage among library declarations is not continuous. That is, CLIENT1 links to CONLIB1, and CONLIB2 links to PROG3, but CONLIB1 never links to CONLIB2. Nevertheless, this example is considered a circular linkage, because the linkage proceeds from a library in PROG1 to a library in PROG2, from a library in PROG2 to a library in PROG3, and from a library in PROG3 back to a library in PROG1.

On the other hand, this example is not an example of circular provision. Procedures X, Y, and Z are each provided directly.

Example 4: Circular Linkage with an Extra Connection Library Linkage

Although circular linkages can include connection libraries, a circular linkage cannot depend on a link between two connection libraries. Thus, in Example 3, the connection library CONLIB1 is linked to client library CLIENT1, and connection library CONLIB2 is linked to server library PROG3. The circularity does not depend on any linkages between two connection libraries.

Now, suppose that a linkage is added to Example 3, so that CONLIB1 imports a procedure from CONLIB2. The result is shown in Figure 18–4.

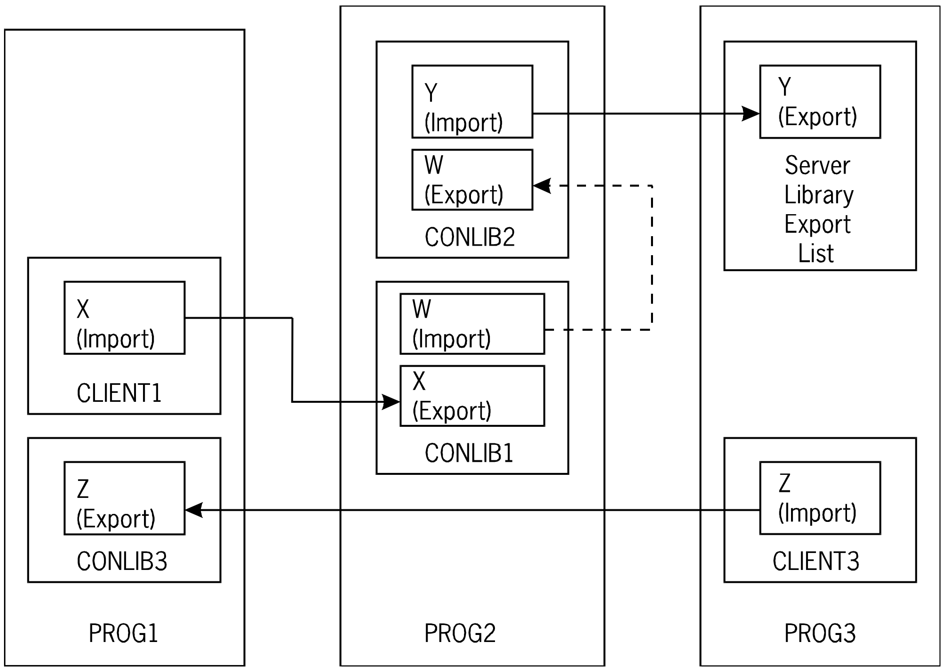

Figure 18–4. Circular Linkage with an Extra Connection Library Linkage

In this example, the arrows point from import procedures to the corresponding export procedures.

In this case, the operating system still considers linkage as circular, but it does not include the link between CONLIB1 and CONLIB2. In general, to determine whether a given series of linkages is circular, you should ignore those linkages that exist between two connection libraries, and then see if the linkage series is still circular.

Example 5: Noncircular Linkage Involving Connection Libraries

The following is an example of a linkage that appears to be circular, but is not considered circular by the operating system:

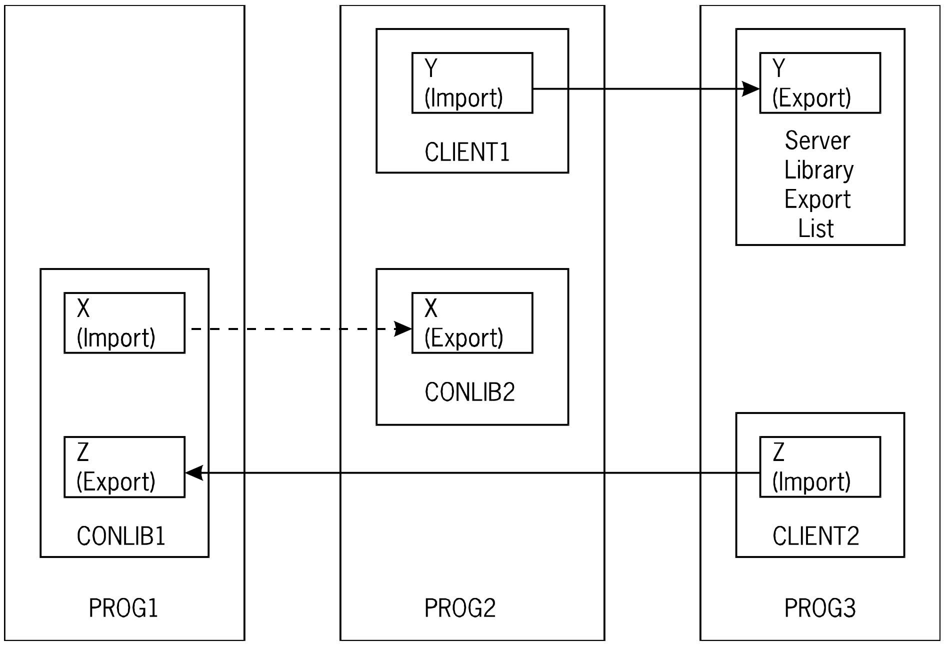

Figure 18–5. Noncircular Linkage Involving Connection Libraries

In this example,

- The arrows point from import procedures to the corresponding export procedures.
- Program PROG1 declares a connection library CONLIB1, which imports procedure X from connection library CONLIB2 in program PROG2.
- Program PROG2 contains both connection library CONLIB2 and client library CLIENT1. Client library CLIENT1 imports procedure Y from server library program PROG3.
- PROG3 includes a client library CLIENT2, which imports procedure Z from connection library CONLIB1 in program PROG1.

The preceding example *appears* to include a circular linkage, because the linkage proceeds from a library in PROG1 to a library in PROG2, from a library in PROG2 to a library in PROG3, and from a library in PROG3 back to a library in PROG1. However, if you ignore the linkage between the two connection libraries (CONLIB1 and CONLIB2), you can see that there is no longer any link between programs PROG1 and PROG2. Because this linkage series depends on a linkage between two connection libraries, this linkage series is not considered circular.

Example 6: Circular Linkage with a Connection Library as Client

A connection library can import an object from a server library that, in turn, is part of a circular linkage. For example:

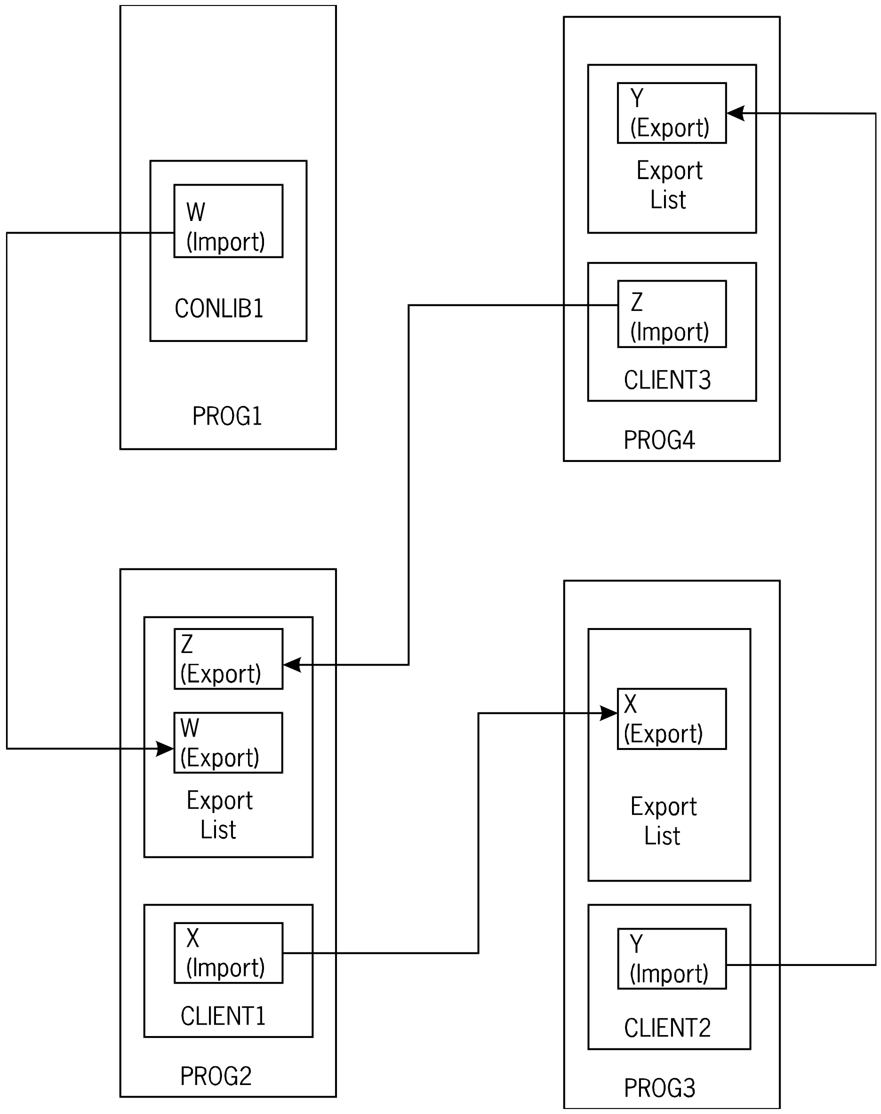

Figure 18–6. Circular Linkage with a Connection Library as Client

In this example,

- The arrows point from import procedures to the corresponding export procedures.
- Program PROG1 contains connection library CONLIB1, which imports procedure W from server library PROG2.
- Server library PROG2 imports procedure X from server library PROG3.

- Server library PROG3 imports procedure Y from server library PROG4.
- Server library PROG4 imports procedure Z from server library PROG2.

In this example, the server libraries PROG2, PROG3, and PROG4 are linked circularly. However, procedure W is not actually provided circularly. Procedure W, which is imported by CONLIB1, is provided directly by PROG2. If PROG2 provided procedure W indirectly by importing W from another library, then the system would not allow CONLIB1 to invoke W. In general, the system does not allow connection libraries to use objects that are provided indirectly, as discussed under "Indirect Provision" earlier in this section.

Restrictions on Circular Linkage and Circular Provision

Requirements for Circular Linkage

If the system determines that a particular linkage attempt would complete a circular linkage, then the system imposes the following restrictions on that linkage attempt:

- A circular linkage can be made only if all the libraries involved are either frozen server libraries or ready connection libraries, and at least one of them was already frozen or ready at the time it linked to one of the other libraries in the circle.
- No more than one of the libraries in a circular linkage can have a freeze type of CONTROL.
- A given object exported by a library cannot be provided circularly by the same object in the same library. That is, if library L exports procedure X, the chain of linkages that provide that procedure cannot lead back to procedure X in library L. However, the linkages could lead back to some other procedure, for example Y, in library L.

  For an example of libraries that violate this restriction, refer to "Example 1: Indirect Self Referencing," under "ALGOL Incorrect Circular Libraries."

Error for Incorrect Linkages

If the client process makes a procedure call that results in a circular linkage of two or more libraries that violates one of these restrictions, the system discontinues the client process and displays the error message "CURRENT CIRCULAR LIBRARY REFERENCE STRUCTURE IS NOT ALLOWED: <library name>." The library name identifies the library at the point in the chain where the linkage became circular. If the client process initiated the chain of circular linkages with a LINKLIBRARY function, the linkage fails and the function returns a value of –7.

Linkages That Cause the Client to Hang

Additionally, some incorrect types of circular linkage can result in the client process hanging indefinitely. This situation occurs if

- A library provides an object by direct provision of the same object in the same library. Refer to "Example 2: Direct Self-Referencing," under "ALGOL Incorrect Circular Libraries."
- Two libraries are waiting on each other to freeze. Refer to "Example 3: Libraries that Wait on Each Other," under "ALGOL Incorrect Circular Libraries."

In either case, the Y (Status Interrogate) system command shows the client process to have a STACK STATE of WAITING ON AN EVENT. However, the STATUS task attribute value remains ACTIVE, and the client process does not appear in the W (Waiting Mix Entries) system command display. This situation continues until an operator enters a DS (Discontinue) system command or until the system is halt/loaded.

For a correct example of circular library linkage, refer to "ALGOL Circular Client Programs" later in this section.

Matching the Object Name

The system matches import objects to export objects only if they have the same name. In general, the name matching is based on the identifier specified in the import or export declaration. However, there are a couple of exceptions to this rule.

Importing Procedures with a Different ACTUALNAME

ALGOL client programs can declare import objects under one name, and cause them to match export projects with a different name, by including an ACTUALNAME clause in the import declaration. For example:

```
PROCEDURE READIT;
  LIBRARY LIB1(ACTUALNAME = "READLINE");
```

Because of the ACTUALNAME assignment, the system looks for a matching export object named READLINE instead of READIT.

Importing Data with a Different ACTUALNAME

For data objects, the ACTUALNAME assignment can be included in the LIBRARY declaration. The following example matches imports object M with export object GG:

```
LIBRARY L (LIBACCESS = BYFUNCTION, FUNCTIONNAME = "MYSUPPORT")
   [ INTEGER M (READWRITE, ACTUALNAME = "GG") ];
```

Changing the ACTUALNAME Outside the Import Declaration

You can also change the actual name of an imported procedure or data item outside the declaration, by using the SETACTUALNAME function. For example:

```
I := SETACTUALNAME (READIT, "READLINE");
```

The SETACTUALNAME function makes the requested change, if possible; otherwise, SETACTUALNAME returns a value indicating why the actual name could not be changed. One reason SETACTUALNAME can fail is that the actual name of an import object cannot be changed while the client process is linked to the library from which the object is imported.

Exporting Objects with a Different ACTUALNAME

Similarly, ALGOL library programs can declare export objects under one name, and cause them to match import objects with a different name, by including an AS clause in the export declaration. For example, the following export declaration causes an object named PROC_READ to be exported under the name READLINE:

```
EXPORT PROC_READ AS "READLINE";
```

Interlanguage Communication

One of the main uses of the ACTUALNAME clause, SETACTUALNAME function, and AS clause is to facilitate interlanguage communication. For example, identifiers in COBOL74 can include hyphens (-), whereas identifiers in ALGOL cannot. If a COBOL74 library exports an object with a name that includes a hyphen, an ALGOL client program cannot declare an import object with exactly the same name. Instead, the ALGOL client program can declare the import object with an identifier that is valid in ALGOL, and use an ACTUALNAME clause to specify the name used in the COBOL74 library.

COBOL Programs: Name of Exported PROCEDURE DIVISION

As discussed under "Creating Server Library Programs" earlier in this section, most COBOL74 programs can be called as libraries. However, these programs do not include export lists or declarations of export objects. Instead, the PROCEDURE DIVISION of the program is always the single export object. If the program contains a PROGRAM-ID comment and the CCI option FEDLEVEL is equal to 5, the first word of this comment is used as the name of the library export object. If no PROGRAM-ID comment appears, or if the FEDLEVEL is not equal to 5, the name of the export object is PROCEDUREDIVISION.

In COBOL85 library programs, which export nested programs as library procedures, the export name is specified by the PROGRAM-ID paragraph in the IDENTIFICATION DIVISION of each nested program.

Type Matching

When a process invokes an imported procedure, the system compares several aspects of the import and export procedures to ensure that they match. The factors considered are discussed in the following pages under the following headings:

- Matching Procedure Types
- Matching Parameter Types
- Matching Array Lower Bounds
- Matching Parameter-Passing Mode

Similarly, when a process uses an imported data item, the system compares the data types of the exported data item and imported data item, as described under "Matching Data Types" later in this section.

Matching Procedure Types

ALGOL Procedure Types

Procedures in ALGOL and some other languages can be invoked as functions that return values. Such procedures are referred to as *typed procedures*. For example, an ALGOL procedure can be of any of the following types:

- Untyped
- ASCII STRING
- BOOLEAN
- COMPLEX
- DOUBLE
- EBCDIC STRING
- HEX STRING
- INTEGER
- REAL

COBOL Procedure Types

A client program written in COBOL74 or COBOL85 can use the GIVING clause of the CALL statement to receive the value returned by a typed library procedure.

FORTRAN77 Procedure Types

A FORTRAN77 library procedure can be any of the following:

- SUBROUTINE
- REAL FUNCTION
- INTEGER FUNCTION

- DOUBLE PRECISION FUNCTION
- LOGICAL FUNCTION
- COMPLEX FUNCTION
- CHARACTER FUNCTION (FORTRAN77 only)
- COMMON
- FILE

Matching the Procedure Types

If an export procedure is typed, the matching import procedure must be of the same type or a compatible type. For information about the compatibility of data types in different languages, refer to "Matching Parameter Types".

Matching Parameter Types

For an import procedure to match the corresponding export procedure successfully, both procedures must specify the same number of parameters. The parameters must be specified in the same order in both procedures. Further, each parameter specified by the import procedure must be of a type compatible with the equivalent parameter to the export procedure.

Parameters in Different Programming Languages

Because the system permits client programs to be written in different languages than the libraries they use, there are times when the actual and formal parameters to a library procedure are specified in different programming languages. Each programming language provides different names for the same or similar types of data.

Listings by ALGOL Equivalents

The following subsections list the parameter types that are available for library procedures in each of the various programming languages. For each parameter type, the equivalent ALGOL parameter type is listed. You can also use this information to deduce which parameter types in two non-ALGOL languages are equivalent. In general, two parameters can match only if they are equivalent to the same ALGOL parameter type.

Example of Using the Tables

For example, you will find that an *int* parameter in C is equivalent to an ALGOL integer variable, and that a BINARY, level 77 1-11 digits parameter in COBOL74 is also equivalent to an ALGOL integer variable. It follows that the C parameter type can also match the COBOL74 parameter type.

Matches between Different ALGOL Types

The system permits one exception to the rule that parameters must match the same ALGOL type. A call-by-value INTEGER parameter of an imported procedure can match a REAL parameter of an exported procedure.

ALGOL Parameter Types

The following are the parameter types that can be specified in an ALGOL library procedure:

- ANYTYPE (The ANYTYPE parameter has special properties that are discussed under "Using the ANYTYPE Parameter" later in this section.)
- ASCII ARRAY or DIRECT ASCII ARRAY
- ASCII STRING or ASCII STRING ARRAY
- BOOLEAN, BOOLEAN ARRAY, or DIRECT BOOLEAN ARRAY
- COMPLEX or COMPLEX ARRAY
- DOUBLE, DOUBLE ARRAY, or DIRECT DOUBLE ARRAY
- EBCDIC ARRAY or DIRECT EBCDIC ARRAY
- EBCDIC STRING or EBCDIC STRING ARRAY
- EVENT or EVENT ARRAY
- FILE or DIRECT FILE
- HEX ARRAY or DIRECT HEX ARRAY
- HEX STRING or HEX ARRAY
- INTEGER, INTEGER ARRAY, or DIRECT INTEGER ARRAY
- INTERLOCK or INTERLOCK ARRAY
- POINTER
- PROCEDURE, declared using the FORMAL clause. (If the procedure has parameters, they must each be one of the types listed previously. The procedure itself must be untyped or else of one of the ALGOL procedure types listed under "Matching Procedure Types".)
- QUEUE
- QUEUE ARRAY
- QUEUE ARRAY REFERENCE
- REAL, REAL ARRAY, or DIRECT REAL ARRAY
- TASK or TASK ARRAY
- TRANSACTION RECORD or TRANSACTION RECORD ARRAY

C Parameter Types

All of the data types supported in C can be passed between a C client program and a C library, with a few exceptions that are documented in the *C Programming Reference Manual, Volume 1: Basic Implementation.*

The data types that can be passed between C libraries and client programs in other languages, or between C client programs and libraries in other languages, are much more limited. Table 18–1, "C Parameters," lists the C parameter types available for interlanguage library calls, and the ALGOL equivalents of these C parameter types.

Table 18–1. C Parameters

ALGOL Parameter	Corresponding C Parameters
BOOLEAN	int (both signed and unsigned types)
INTEGER	char int (both signed and unsigned types) short (both signed and unsigned types) long (both signed and unsigned types) pointers (all types)
REAL	float double
DOUBLE	long double
INTEGER ARRAY	int [ ] short [ ] long [ ]
EBCDIC ARRAY	char [ ] __heap_t
REAL ARRAY	float [ ] double [ ] long double [ ] struct union
FILE	_file_t
INTEGER PROCEDURE	char () int () short () long ()

Table 18–1. C Parameters

ALGOL Parameter	Corresponding C Parameters
REAL PROCEDURE	float () double ()
DOUBLE PROCEDURE	long double ()
PROCEDURE	void ()

The *__heap_t* parameter passes the value of the *heap*, which is a memory area where a C program stores arrays, structures, addressed objects, and dynamically allocated objects.

Pointers in C are integer types that indicate the location of an item within the heap. The exact meaning of C pointers varies according to the memory model used for the C program; the programmer can request a particular memory model with the MEMORY_MODEL compiler control option. If the heap is implemented with a MEMORY_MODEL value of TINY or SMALL, then a non-C program that calls a C library can use the C pointer as an indicator of the offset of the item within a heap. If the heap is implemented with a different MEMORY_MODEL value, then the non-C program must use the *__heap_to_ptr_t* procedure to convert the C pointer into a conventional pointer.

The *__heap_to_ptr_t* procedure is one of several export procedures that the C compiler automatically creates in each C library. Other procedures that aid in array handling include *__copy_to_ptr_t*, *__copy_from_ptr_t*, *__free_t*, and *__malloc_t*. These procedures can be accessed only by programs written in ALGOL (including any of the extended forms of ALGOL), NEWP, or Pascal. For examples of the use of some of these procedures, refer to "Library Examples."

COBOL74 Parameter Types

Table 18–2, "COBOL74 Parameters," lists the allowable parameters to a COBOL74 library and the corresponding ALGOL parameters. For further information about COBOL74 parameters, refer to the *COBOL ANSI-74 Programming Reference Manual, Volume 1: Basic Implementation*.

Table 18–2. COBOL74 Parameters

ALGOL Parameter	Corresponding COBOL74 Parameters
DOUBLE	BINARY, 77 12-23 digits
	DOUBLE, 77
DOUBLE ARRAY [0]	DOUBLE, 01
EBCDIC ARRAY [0]	COMP, 01 group item
	DISPLAY, 01
	INDEX, 01 group item
EBCDIC STRING (nonresizable)	DISPLAY item in STRING clause
EVENT	EVENT, 77
EVENT ARRAY [0]	EVENT, 01
FILE	FILE
HEX ARRAY [0]	COMP, elementary 01 and 77
	INDEX, 01 elementary item
INTEGER	BINARY, 77 1-11 digits
	COMP item in INTEGER clause
INTEGER ARRAY [0]	BINARY, 01 (If $INTEGERBNRY = TRUE, the default)
PROCEDURE	TRANSACTION PROCEDURE
REAL	REAL, 77
REAL ARRAY [0]	BINARY, 01 (If $INTEGERBNRY = FALSE)
	REAL, 01
TRANSACTION RECORD	TRANSACTION RECORD
TRANSACTION RECORD ARRAY [0]	TRANSACTION RECORD ARRAY

In COBOL74 client programs, the GIVING clause of a CALL statement specifies a variable to receive the value returned by a typed procedure. If the item in the GIVING clause is a level 77 REAL, the procedure must be of type real. If the item in the GIVING clause is a level 77 DOUBLE, the procedure must be of type double. If the item in the GIVING clause is of any other type, the procedure must be of type integer, and the system converts the integer value returned by the procedure to the data type specified in the GIVING clause. If there is no GIVING clause, the procedure must be untyped.

COBOL85 Parameter Types

Table 18–3, "COBOL85 Parameters," lists the allowable parameters to a COBOL85 library and the corresponding ALGOL parameters. For further information about COBOL85 parameters, refer to the *COBOL ANSI-85 Programming Reference Manual, Volume 1: Basic Implementation*.

Table 18–3. COBOL85 Parameters

ALGOL Parameter	Corresponding COBOL85 Parameters
DOUBLE	DOUBLE level 77
DOUBLE ARRAY	DOUBLE level 01
(Double Integer)	BINARY level 77, 12 to 23 digits
Note: *This data type is not implemented in ALGOL.*	Integer (COMPUTATIONAL) 12 to 23 digits
(Double Integer Array)	BINARY level 01, 12 to 23 digits
Note: *This data type is not implemented in ALGOL.*	
EBCDIC ARRAY	DISPLAY
EBCDIC STRING	String (DISPLAY)
HEX ARRAY	COMPUTATIONAL and INDEX
INTEGER	BINARY level 77, 1 to 11 digits
	Integer (COMPUTATIONAL) 1 to 11 digits
INTEGER ARRAY	BINARY level 01, 1 to 11 digits
REAL	REAL level 77
REAL ARRAY	REAL level 01

The BINARY and INTEGER items with 12-23 digits correspond to a data type that is not implemented in ALGOL, but which would be a double-precision integer. These parameter types can be passed only between a COBOL85 client program and a COBOL85 library.

In COBOL85 client programs, the GIVING clause of a CALL statement can specify a variable to receive the value returned by a typed procedure. If the item in the GIVING clause is a level 77 numeric, you can use Table 18–5 to determine the corresponding procedure type. If the item in the GIVING clause is a level 01 numeric item, the item can match only a procedure of type Integer.

FORTRAN77 Parameter Types

Table 18–4, "FORTRAN77 Parameters," lists the allowable parameters to a FORTRAN77 library and their corresponding ALGOL equivalents. For further information about FORTRAN77, refer to the *FORTRAN77 Programming Reference Manual.*

Table 18–4. FORTRAN77 Parameters

ALGOL Parameter	Corresponding FORTRAN77 Parameter
BOOLEAN	LOGICAL
BOOLEAN ARRAY [*]	LOGICAL array
COMPLEX	COMPLEX
COMPLEX ARRAY [*]	COMPLEX array
DOUBLE	DOUBLE PRECISION
DOUBLE ARRAY [*]	DOUBLE PRECISION array
EBCDIC ARRAY [*]	CHARACTER
	CHARACTER array
INTEGER	INTEGER
INTEGER ARRAY [*]	INTEGER array
REAL	REAL
REAL ARRAY [*]	REAL array

Notes:

- *For the parameters COMPLEX array and DOUBLE PRECISION array, the DOUBLEARRAYS option must be set in the FORTRAN77 source.*
- *The EBCDIC ARRAY [*] parameter must be followed by an INTEGER parameter. The INTEGER parameter matches the hidden lower-bounds parameter that FORTRAN77 generates for a CHARACTER or CHARACTER ARRAY.*

A FORTRAN77 array with a lower bound of 1 is equivalent to an ALGOL array with a lower bound of 0.

NEWP Parameter Types

Table 18–5, "NEWP Parameters," lists the allowable parameters to a NEWP library. The corresponding ALGOL parameters are identical. For further information about NEWP, refer to the *NEWP Programming Reference Manual*.

Table 18–5. NEWP Parameters

ALGOL Parameter	Corresponding NEWP Parameter
ANYTYPE	ANYTYPE
ASCII ARRAY [0]	ASCII ARRAY [0]
BOOLEAN	BOOLEAN
BOOLEAN PROCEDURE	BOOLEAN PROCEDURE
(None)	CONNECTION
DIRECT ARRAY [0]	DIRECT ARRAY [0]
DIRECT EBCDIC ARRAY [0]	DIRECT EBCDIC ARRAY [0]
DIRECT FILE	DIRECT FILE
DIRECT HEX ARRAY [0]	DIRECT HEX ARRAY [0]
DIRECT REAL ARRAY [0]	DIRECT REAL ARRAY [0]
DOUBLE	DOUBLE
DOUBLE ARRAY [0]	DOUBLE ARRAY [0]
DOUBLE PROCEDURE	DOUBLE PROCEDURE
EBCDIC ARRAY [0]	EBCDIC ARRAY [0]
EVENT	EVENT
EVENT ARRAY [0]	EVENT ARRAY [0]
FILE	FILE
HEX ARRAY [0]	HEX ARRAY [0]
INTEGER	INTEGER
INTEGER ARRAY [0]	INTEGER ARRAY [0]
INTEGER PROCEDURE	INTEGER PROCEDURE
INTERLOCK	INTERLOCK
INTERLOCK ARRAY	INTERLOCK ARRAY
POINTER	POINTER
PROCEDURE (Untyped)	PROCEDURE (Untyped)
QUEUE	WORD

Table 18–5. NEWP Parameters

ALGOL Parameter	Corresponding NEWP Parameter
QUEUE ARRAY	WORD ARRAY
QUEUE ARRAY REFERENCE	WORD ARRAY
REAL	REAL
REAL	SHORT SET (MAX VALUE <= 47)
REAL ARRAY	LONG SET (MAX VALUE > 47)
REAL ARRAY [0]	REAL ARRAY [0]
REAL PROCEDURE	REAL PROCEDURE
TASK VARIABLE or ARRAY [0]	TASK VARIABLE or ARRAY [0]
TRANSLATE TABLE	TRANSLATE TABLE
TRUTHSET	TRUTHSET

Note: *The ANYTYPE parameter has special properties that are discussed under "Using the ANYTYPE Parameter."*

Pascal Parameter Types

Table 18–6, "Pascal Parameters," lists the allowable parameters to a Pascal library and their corresponding ALGOL equivalents. For further information about Pascal, refer to the *Pascal Programming Reference Manual, Volume 1: Basic Implementation*.

Table 18–6. Pascal Parameters

ALGOL Parameter	Corresponding Pascal Parameters
BOOLEAN	Boolean Boolean subrange
BOOLEAN ARRAY [*]	Array of Boolean
BOOLEAN PROCEDURE	Function: Boolean Function: Boolean subrange
DOUBLE	Fixed (n > 11) Sfixed (n > 11)
DOUBLE ARRAY [*]	Array of fixed (n > 11) Array of sfixed (n > 11) Packed array of fixed (n > 11) Packed array of sfixed (n > 11)
DOUBLE PROCEDURE	Function: fixed (n > 11) Function: sfixed (n > 11)
EBCDIC ARRAY [*]	Bits (n) Binary (n) U_display (n) Z_display (n) Display_z (n) S_display (n) Display_s (n) Word48 (n) Word96 (n) Integer48 Integer96 Real48 Explicit record (var) Packed array of char

Table 18–6. Pascal Parameters

ALGOL Parameter	Corresponding Pascal Parameters
EBCDIC ARRAY [*] *(cont.)*	Packed array of subrange (17-256 elements in subrange) Packed array of enumeration (17-256 elements in enumeration)
FILE	Systemfile
HEX ARRAY [*]	Hex (n) Digits (n) S_digits (n) Digits_s (n) Boolean1 Boolean4 Packed array of Boolean Packed array of subrange (0-16 elements in subrange) Packed array of enumeration (0-16 elements in enumeration)
INTEGER	Integer Char Enumeration Fixed (n < 12) Sfixed (n < 12) Integer subrange Char subrange Enumeration subrange
INTEGER ARRAY [*]	Array of integer Array of char Array of enumeration Array of fixed (n < 12) Array of sfixed (n < 12) Array of integer subrange Array of char subrange Array of enumeration subrange Packed array of integer Packed array of fixed (n < 12) Packed array of sfixed (n < 12);

Table 18–6. Pascal Parameters

ALGOL Parameter	**Corresponding Pascal Parameters**
INTEGER ARRAY [*] *(cont.)*	Packed array of subrange (> 256 elements in subrange) Packed array of enumeration (> 256 elements in enumeration)
INTEGER PROCEDURE	Function: integer Function: char Function: enumeration Function: fixed (n < 12) Function: sfixed (n < 12) Function: integer subrange Function: char subrange Function: enumeration subrange
PROCEDURE	Procedure
REAL	Real Short set (max value <= 47)
REAL ARRAY [*]	Array of real Array of record Array of set Array of vlstring Array of packed array Array of explicit type Long set (max value > 47) Record Vlstring Explicit record (by-value) Packed array of real Packed array of set Packed array of record Packed array of vlstring
REAL PROCEDURE	Function: real

Some types of Pascal parameters can cause extra parameters to be passed if a variable of the parameter is declared as a parameter for a procedure or function. The Pascal parameters affected by this are string schema, fixed length string schema, and any other schema.

Refer to the *Pascal Programming Reference Manual, Volume 1: Basic Implementation* for information about the Pascal parameters.

Subranges of types integer, Boolean, char, or enumeration are mapped as their host type, except when the subranges or the types are components of packed arrays, which are described below.

Each user-defined type identifier is resolved to one of the Pascal parameter types shown in Table 18–8 according to its general type characteristics. For example, type *color*, as it is usually defined, would be considered an *enumerated type* and would be mapped to the generic type *integer*. The following two array declarations are equivalent:

```
array [index-type1] of array [index-type2] of...

array [index-type1, index-type2] of...
```

Packed arrays of integers, reals, sets, records, variable-length strings, or other arrays are mapped as unpacked arrays of the same type. For packed arrays of Boolean, char, subrange, or enumeration types, the mapping depends on the number of bits it takes to represent the range of the type. If four bits or fewer are required, the mapping is to a hexadecimal array with lower bound. If five to eight bits are required, the mapping is to an EBCDIC array with lower bound. If nine bits or more are required, the mapping is to an integer array with lower bound. More details appear in the data representation discussion in the *Pascal Programming Reference Manual, Volume 1: Basic Implementation*.

If a Pascal library program declares a parameter to be received as a read-only parameter, the client program is allowed to pass the corresponding actual parameter either by call-by-name, call-by-reference, call-by-value, or read-only. This is allowed because the Pascal library program ensures that the parameter's value remains unchanged. A client program can only pass a read-only parameter to a library program that receives the parameter as a read-only parameter, to ensure that the value of the actual parameter is not changed.

Using the ANYTYPE Parameter

The ANYTYPE parameter enables an import procedure to accept any kind of data item from the statement that invokes the procedure. ANYTYPE parameters are supported only in ALGOL and NEWP.

The matching export procedure must occur in a NEWP program, in a block marked as UNSAFE, with the parameter declared as type WORD. Otherwise, when the import procedure is invoked, an error occurs and the system displays the following message:

```
Object <procedure name>: Type or parameter mismatch in interface <client
library identifier> to library <library name>
```

The export procedure can examine the tag bits in the parameter to determine the actual data type that was passed.

The ANYTYPE parameter is typically used for accessing certain operating system procedures that accept multiple data types. An example of such a procedure is the MCPSUPPORT procedure CHECKPOINTARRAY, which is described in Section 11, “Restarting Jobs and Tasks.”

The following is an example of a client program with an import procedure that uses the ANYTYPE parameter:

```
BEGIN

LIBRARY L (LIBACCESS = BYFUNCTION, FUNCTIONNAME = "TESTSUPPORT.");

PROCEDURE TESTPROC (W);
   ANYTYPE W;
   LIBRARY L;

INTEGER I;
EBCDIC ARRAY EBTEST[0:99];
REAL ARRAY RTEST[0:99,0:99];

TESTPROC(I);
TESTPROC(EBTEST);
TESTPROC(RTEST);

END.
```

In this example, import procedure TESTPROC is declared with parameter W of type ANYTYPE. The program invokes procedure TESTPROC three times, passing a different type of data item to parameter W each time.

Matching Array Lower Bounds

Array parameters can be declared either with an undeclared lower-bound specification or with a formal lower-bound specification. Arrays with undeclared lower bounds are hereafter referred to as *unbounded arrays*. Arrays with formal lower bounds are hereafter referred to as *simple arrays*.

In the case of unbounded arrays, the lower-bound value of the array is provided during execution by the program that calls the procedure. In the case of simple arrays, the lower-bound value of the array is fixed during compilation, typically to a value of zero. The actual value of the lower-bound parameter for the simple array is ignored during execution.

If an unbounded array appears as a parameter to an imported or exported procedure, the system generates one or more hidden parameters that pass the actual lower bound for each dimension of the array. These hidden parameters are integer parameters that follow the array parameter and are passed by value.

The syntax for array specifications in the various programming languages is described in Table 18–7.

Table 18–7. Unbounded and Simple Array Declarations

Language	Unbounded Array	Simple Array
ALGOL	ARRAY A[*];	ARRAY A[0];
C	float (*) [ ]	None
COBOL74	01 A BINARY.	01 A BINARY.
	77 B PIC S9(11) BINARY.	
COBOL85	01 A BINARY WITH LOWER-BOUNDS.	01 A BINARY.
FORTRAN77	REAL A(*).	None
NEWP	ARRAY A[*];	ARRAY A[0];
Pascal	var a: ARRAYTYPE;	None

Libraries and client programs written in COBOL85 can receive unbounded array parameters by including a LOWER-BOUNDS clause in the formal array declaration. A COBOL85 library always treats such an array parameter as if it had a lower bound of zero, regardless of the actual lower bound passed by the client program.

COBOL74 libraries and client programs can specify unbounded array parameters by adding a numeric parameter that receives the lower bound. In a COBOL74 client program, this extra parameter must occur immediately after the unbounded array in the CALL statement parameter list. In a COBOL74 library program, this extra parameter must occur immediately after the unbounded array in the USING clause of the PROCEDURE DIVISION. COBOL74 libraries always treat array parameters as if they had a zero lower bound, regardless of the value passed in the extra parameter.

Pascal array parameters can reference space in a data pool or in the heap. As a result, if a Pascal client program passes an array parameter to a library, some constructs in the library might result in incorrect references or cause overwrite corruption of other arrays stored in the same data pool or heap.

For example, suppose a Pascal client program passes an array A to an ALGOL library. In the exported ALGOL procedure, the expression *POINTER(A)* references the first element in the data pool or heap. By contrast, the expression *POINTER(A[0])* correctly references the actual first element of the Pascal array A. Other constructs that can cause similar problems, if not carefully used, include the SIZE and REMAININGCHARS functions and the REPLACE, SCAN, and RESIZE statements. Note that no system function can return the size of a Pascal array.

Additional problems can arise if a Pascal client program passes an array parameter to a library written in a COBOL language. Like ALGOL libraries, COBOL libraries have no way of determining the upper bound of an array in the heap. However, COBOL libraries have the additional limitation that the lower bound of the array is always treated as zero, regardless of where the array starts in the data pool.

Matching Parameter-Passing Mode

Library parameters can be passed by value, by reference, by name, or as read-only. The read-only property causes the compiler to select the most efficient parameter-passing mode, and prevents the receiving procedure from modifying the value of the parameter. For an introduction to these parameter-passing modes and read-only parameters, refer to "Parameter Passing Modes" in Section 17, "Using Parameters."

If the library program declares a parameter to be received as a read-only parameter, the client program can pass the parameter as call-by-name, call-by-reference, call-by-value, or read-only. If a library program declares a parameter to be received by name or by reference, the client program can pass the parameter by name, by reference or by value. If the library program declares a parameter to be received by value, the client program can only pass the parameter by value. For pointer parameters, the parameter-passing mode of the library program and the client program must match.

Table 18–8, “Parameter-Passing Modes,” illustrates the parameter-passing rules. In the table, the legal combinations of parameter-passing modes are marked with a P for pointer parameters and an X for all other parameter types.

Table 18–8. Parameter-Passing Modes

	Client Program			
Library Program	**Read-Only**	**Name**	**Reference**	**Value**
Read-Only	X	X	X	X
Name		X	X	X
Reference		X	P/X	X
Value				P/X

In ALGOL programs, parameters are declared with or without a VALUE clause. In ALGOL library programs, parameters declared with a VALUE clause are received by value. Parameters declared without the VALUE clause are received by reference, except for formal procedures and integer, real, double, Boolean, and complex variables, which are received by name. In ALGOL client programs, library procedure parameters declared with a VALUE clause are passed by value. Parameters declared without a VALUE clause are passed by reference, except for integer, real, double, Boolean, and complex variables, which are passed by name.

In COBOL74, parameters are passed by reference; therefore, a COBOL74 client program cannot call a library that has declared its parameters by value. An ALGOL library must declare its parameters to be by name or by reference if the ALGOL library is to be called by a COBOL74 program.

A COBOL74 library must declare its 77-level BINARY parameters to be received by reference. If any BINARY parameter is received by content, the program is not library-capable. Due to the parameter-passing rules and the ALGOL and COBOL74 language restrictions, an ALGOL program that calls a COBOL74 library must declare its integer, real, or double parameters to be by value.

In COBOL85, parameters can be passed by reference or by value. If a COBOL85 program passes a group item of other than 01 level by reference, then the item is passed as call-by-value-result. In other words, the value of the item in the calling program is updated when the library procedure terminates.

In FORTRAN77, variable parameters are passed by name; array parameters are passed by reference.

In NEWP, parameters are declared to be by value or by reference. Integers, real, double, and Boolean variables that are not declared to be by value are passed by reference.

In Pascal, parameters are passed by reference, read-only, or value. Parameters that are not passed by reference (“VAR” parameters) or by read-only (“CONST” parameters) are passed by value.

Matching Data Types

ALGOL and NEWP provide the ability to export and import data objects, as discussed earlier in this section under "Exported Data in Server Libraries," "Importing Data to Client Programs," and "Exported and Imported Data in Connection Libraries."

ALGOL Capabilities

ALGOL libraries can export or import simple variables of the following data types:

- Simple variables of type BOOLEAN, COMPLEX, DOUBLE, EVENT, INTEGER, and REAL.
- Arrays of any of these data types, as well as ASCII, EBCDIC, and HEX.

  Note that if an exported array is multidimensional, it must have an access mode of READWRITE.

ALGOL cannot export VALUE arrays, but does permit array references to be used to export an array that is assigned the VALUE array.

NEWP Capabilities

Currently, the data exporting and importing capabilities of NEWP are much more restricted than those of ALGOL. In NEWP, the full ability to both import and export data, optionally specifying an access mode, exists only for events and event arrays. NEWP can export other types of arrays, but cannot import them and cannot specify an access mode for them.

FORTRAN77 Capabilities

This section does not discuss the exporting of data in FORTRAN77. Instead, refer to the *FORTRAN77 Programming Reference Manual*.

Matching Data Names

The names of the exported and imported data objects must match, unless an ACTUALNAME assignment is used to establish an alias used for matching, as discussed under "Matching the Object Name" earlier in this section.

Matching Data Types

The type of data that is imported must match the exported data type exactly, except that an INTEGER export can match a REAL import if the access mode is read-only.

Determining Which Clients Are Linked to a Library

Y (Status Interrogate) System Command

You can use the Y (Status Interrogate) system command to display the client processes that are linked to a server library process. The following is an example of this display:

```
Status of Job 2375/2375 at 01:14:11
Program name: *SYSTEM/GENERALSUPPORT
   Codefile created: Monday, July 16, 2001 (20011197) at 14:55:19
Priority: 50
Origination: Unit 0
Stack State: Frozen, Permanent

This library is being used by 8 programs:
   The MCP
   9345: *SYSTEM/SDASUPPORT ON SYS38
   4972: *SYSTEM/TCPHOSTSERVICES
   4876: *SYSTEM/TCPIP/BNAV2/MANAGERS/12152
   6551: *OBJECT/MAIL
   2953: (SWDUNCAN)MCP/39/TRAP/MULTICOMP/AMLIP
   2915: (JASMITH)MCP/MM17A ON MCPS
   4983: *SYSTEM/PRINT/REMOTE/SERVER
```

Equivalent GETSTATUS Call

A DCALGOL program can obtain the same information by using the GETSTATUS call with type 6 (Mix Entries), subclass 1, and mask bits 18 and 34 set. Mask bit 18 returns the number of client processes linked to the library. Mask bit 34 returns a list of the client processes.

Understanding Library Process Structure

In most respects, an imported object can be used just as if it were declared by the importing process rather than the server library or connection library process. The following pages explore the effects of library process structure on scope of declarations, task attribute usage, and error handling.

In this discussion, the process that imports an object is referred to as the *importing process.* The importing process might be a client process (which imports an object through a client library declaration), or a connection library process (which imports an object through a connection library declaration).

Process Stacks

A process can either enter or initiate an imported procedure.

If a process enters an imported procedure, the procedure is executed on the process stack of the importing process, rather than the process stack of the library that exported the procedure.

If a process initiates an imported procedure, the procedure must run as a dependent process. The system creates a new process stack to execute the procedure. The resulting process is considered an external process. (For information about external processes, refer to Section 2, "Understanding Interprocess Relationships.")

Using Isolated Procedures in Libraries

The ISOLATED procedure attribute specifies that when an isolated procedure runs on a stack other than the stack that declared the procedure, it is protected from operator DS (Discontinue) commands and operator ST (Stop) commands that are applied to the stack on which the isolated procedure is running.

An operator-entered DS command applied to the stack that declared an isolated procedure results in the MCP discontinuing all stacks running its procedures, including its isolated procedures.

The stack that declared a procedure is the task or library that was started when the code file containing the procedure code was initiated. For library procedures, for example, this stack is called the library stack.

When an isolated procedure is running on a stack other than its own stack, and that stack is discontinued or stopped by an operator, the stack is discontinued or stopped below the isolated stack frames.

- If those frames continue to run for 5 seconds of processor time after an operator initiates a DS command, a message is displayed that indicates the operator-entered DS command has been delayed while running isolated code from a specified task.

- If those frames continue to run for 5 seconds of processor time on a stack stopped by an operator, the stop action takes effect as if the ISOLATED procedure attribute had not been specified, and a message is displayed describing the situation.
- If those frames continue to run for 5 seconds of processor time on a resource-discontinued stack, an RSVP message enables the operator to override the resource limit. The RSVP message also indicates that the alternative is to terminate the task that owns the "isolated" environment.

Thus a library or other IPC program that uses the ISOLATED procedure attribute can assume that the isolated procedures are not interrupted by operator-entered DS commands when running on other stacks. In addition, operator-entered ST commands cannot delay the isolated procedures if those procedures do not consume more than 5 seconds of processor time before exiting.

If an isolated procedure fails to exit cleanly while running on a stack other than the stack that declared it, the stack that declared it is discontinued by the MCP. Failure to exit cleanly includes things such as fatal errors in the isolated procedure or in procedures invoked above the isolated procedure, or a GO TO statement that cuts back the stack environment of the isolated procedure.

When a stack is running isolated code from another stack, asynchronous actions such as software interrupts, signals, approval and change procedures, and so on are delayed until the isolated code exits. (If this restriction were not enforced, a software interrupt or other asynchronous action that makes the isolated procedure work correctly could be violated. For example, a software interrupt or other asynchronous action could initiate a GO TO statement that would cut back the stack environment of the isolated procedure. Other actions could also take place that the lock or locks held by the isolated procedure were designed to prevent.)

Note: *Software interrupts declared within isolated procedures or declared within procedures invoked above isolated procedures never run.*

When a stack is running isolated code from another stack, stack stretch is permitted up to the limits imposed by hardware, and the discontinuance of stack overflow is postponed until the isolated code exits.

A library or IPC program that uses isolated procedures might use the DSABLE option of the WAIT or LOCK command so that the wait or lock attempt terminates and returns a value indicating that the client stack has been discontinued. If one of these commands is used with the DSABLE option, there should be code to cleanly terminate the operation and exit the procedure when the operator issues a DS command.

When running in an isolated procedure (or in a nonisolated procedure invoked above an isolated procedure) on an operator-discontinued stack (different than the stack which declared the isolated procedure), most system interfaces continue to function as if the stack had not been discontinued. For example, files can be opened, read, written, and so on. Some exceptions are as follows:

- Any RSVP situation that typically lists DS as an acceptable reply takes the DS command. For example, an attempt to open a file that does not exist produces an open result of DSEDINFINDV, which means an open attempt by a nondiscontinuable task was discontinued by a DS system command while the task was trying to find the file.
- DSABLE WAIT and LOCK statements terminate immediately and return a value indicating that the client stack has been discontinued. A Wait on Time operation that does not also use events (WHEN in ALGOL; WAIT (<arithmetic expression>) in NEWP) is interrupted when an operator initiates a DS command, but is fully performed if the operation is begun after the operator initiates a DS command. This is true whether or not the DSABLE option was specified. (To get a fully DSABLE or non-DSABLE Wait on Time operation, use WAIT[DSABLE](<arithmetic expression>,E), where E is an event that is never caused.)

When a stack is running an isolated procedure on a stack other than the stack that declared the procedure, the isolated semantics apply to that stack until that procedure exits or the task or library that declared the procedure is discontinued. Thus, when such an isolated procedure invokes a nonisolated procedure, the nonisolated procedure experiences isolated semantics. This is true regardless of how many isolated or nonisolated procedures are invoked above the first isolated procedure.

When a stack is running multiple isolated procedures declared by multiple different stacks, the isolated semantics apply down to the lowest of the stack frames of an isolated procedure from a task or library that has not been discontinued.

An isolated procedure can cause some system operations (for example, operator-entered DS and ST commands) to be delayed.

Tip: Code an isolated procedure so that the procedure runs quickly and efficiently, and does not consume excessive system resources.

The ISOLATED procedure attribute is usually used in the context of shared libraries that have global data protected by locking mechanisms. However, isolated procedures could be used in any situation where a procedure or procedures from one stack run on another stack or stacks.

Example of an Isolated Procedure

The following library stores configuration data, updating it and searching it in response to invocations of its exported procedures. An interlock and two isolated procedures are used to ensure that updates and searches do not overlap, that partial updates to the configuration data never occur, and that the interlock is never left locked, even if client tasks are discontinued by an operator while running procedures from the library.

```
 $ SHARING = SHAREBYALL
BEGIN
   <global configuration data declarations>
   INTERLOCK CONFIG_LOCK;

   PROCEDURE UPDATE_CONFIG(<parameters>);
   BEGIN

      <local configuration data declarations>

      PROCEDURE UPDATE_GLOBAL;

      BEGIN [ISOLATED]

         LOCK(CONFIG_LOCK);

         <copy from local configuration data to global
         configuration data>

         UNLOCK(CONFIG_LOCK);
      END UPDATE_GLOBAL;

      <build local configuration, using parameters, data from
      files, etc.>
      UPDATE_GLOBAL;

   END UPDATE_CONFIG;

   PROCEDURE SEARCH_CONFIG(<parameters>);
   BEGIN [ISOLATED]
      LOCK(CONFIG_LOCK);
      <using data from parameters, search global configuration
      data>
      <store results of search in other parameters>
      UNLOCK(CONFIG_LOCK);
   END SEARCH_CONFIG;

   EXPORT UPDATE_CONFIG, SEARCH_CONFIG;

   FREEZE(PERMANENT);

END
```

Library Task Attributes

Task Attributes for Entered Procedures

If a process enters an imported procedure, the task attributes of the importing process govern the execution of the procedure. If the MYSELF predeclared task variable is used in the imported procedure, MYSELF refers to the importing process. If the MYJOB task variable is used in the imported procedure, MYJOB refers to the job of the importing process (that is, the eldest ancestor of the importing process).

Task Attributes for Initiated Procedures

If a process initiates an imported procedure, the resulting process receives its own set of task attributes. In this case, the MYSELF task variable refers to the new process. The MYJOB task variable refers to the job of the importing process and the new process. (Because the new process must be dependent, the client process and the new process always have the same job.)

Access to Task Attributes of the Library

Thus, the MYSELF task variable, when referenced in an imported procedure, never refers to the task attributes of the library process that exported the procedure. There is no direct way for the importing process to access the task attributes of the library process it is linked to.

On the other hand, you can design a library to provide importing processes with indirect access to the task variable of the library process. The following ALGOL server library program provides indirect access to its own task variable:

```
BEGIN

PROCEDURE VIRTUAL_OB (LIBTASK);
   TASK LIBTASK;
   BEGIN
   PROCEDURE X (I);
      INTEGER I;
      BEGIN
      I := LIBTASK.TASKVALUE;
      END;
   EXPORT X;
   FREEZE(TEMPORARY);
   END;

VIRTUAL_OB (MYSELF);

END.
```

In the preceding example, the exported procedure X indirectly interrogates a task attribute of the library's task variable. This indirect access is possible because the library program passes MYSELF as an actual parameter to the formal parameter LIBTASK of procedure VIRTUAL_OB. Because exported procedure X is declared in VIRTUAL_OB, procedure X has access to the LIBTASK variable.

APPROVAL and CHANGE Procedures

The MYSELF task variable should not be used inside APPROVAL procedures or CHANGE procedures. For further information, refer to the "APPROVAL" and "CHANGE" headings earlier in this section.

Task Attributes Useful for Libraries

Certain task attributes are particularly useful in the implementation of library applications. These include the following:

- LIBRARY

  This attribute passes library equations to a client process at run time. A library equation modifies the library attributes of libraries declared in the client process. Each library equation is applied to the library declaration with the corresponding internal name, as discussed under "INTNAME" earlier in this section.

- LIBRARYSTATE

  This attribute reports several types of information about a library process, including the sharing value, the type of freeze, and the linkage class. A library process also uses this attribute to determine whether the library process was initiated through the library linkage mechanism.

- LIBRARYUSERS

  For a server library process, this attribute returns the total number of client processes or connection libraries that are linked to this server library.

- STATUS

  A frozen server library process can be thawed by assigning this task attribute the value of GOINGAWAY, as discussed in Section 6, "Monitoring and Controlling Process Status." In addition, the GOINGAWAY assignment prevents further client processes from linking to this library instance.

For complete descriptions of these task attributes, refer to the *Task Attributes Programming Reference Manual.*

Error Handling

Faults in Imported Procedures

If an importing process encounters a fault while executing an imported procedure, the system treats this as a fault in the importing process rather than in the process that exported the procedure. If neither the imported procedure nor the importing program incorporates fault-handling code, the fault causes the importing process to terminate. The fault has no effect on the status of the exporting library process.

Conditions That Can Discontinue the Library Process

A server library with a permanent or temporary freeze cannot incur any faults while frozen, because it is not executing any statements. However, an operator can terminate the process with a DS (Discontinue) system command. Further, a server library with a control freeze can incur faults while executing the control procedure, even while the library is frozen.

Effects of Library Termination

If a library process is terminated by a fault or operator action while client processes are linked to it, the system discontinues all processes currently executing procedures imported from that library, and it delinks all other libraries linked to that library.

APPROVAL Procedure Errors

If an error or fault occurs in an APPROVAL procedure, the system prevents the linkage from completing. However, neither the linking process nor the library being linked to incurs an error.

CHANGE Procedure Errors

If an error or fault occurs in a CHANGE procedure, the system prevents the change in connection state from completing. Further, the error or fault is propagated to the requesting process (the process that initiated the linkage or delinkage that is in progress). If the requesting process does not include code to handle the fault, the system discontinues the requesting process.

Errors Linking to Server Libraries

For information about errors that can occur when a process links to a server library or uses an object imported from a server library, refer to “Linking to Server Libraries”.

Errors Linking to Connection Libraries

For information about errors that can occur when a process links to a connection library or uses an object imported from a connection library, refer to “Matching Connection Library Objects”.

Design Strategies for Linking Libraries

Technical Architectures

Any program that uses library linkages is part of a distributed application. To get the most out of using libraries, it is essential that such applications are designed with this distributed nature in mind. Almost all architectures used to create distributed applications follow one of the two following models.

Two-Tier Model

The two-tier model is the more traditional approach to application distribution. Figure 18–7 illustrates the two-tier model.

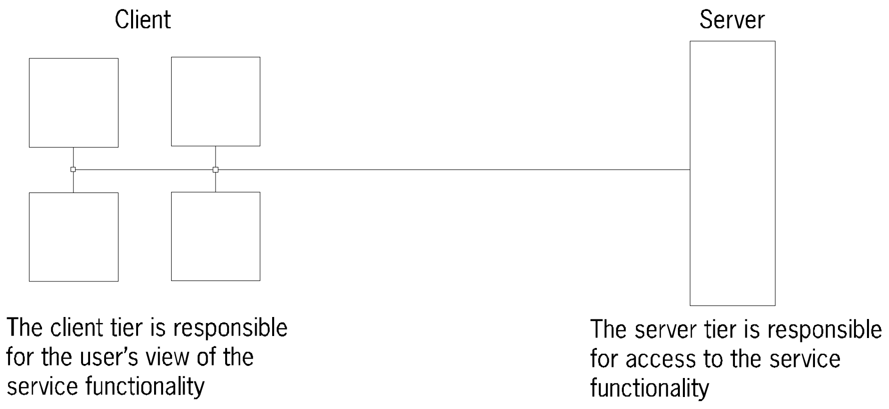
Client
Server
The client tier is responsible for the user's view of the service functionality
The server tier is responsible for access to the service functionality

Figure 18–7. Two-Tier Model

Three-Tier Model

The three-tier model is the newer approach to application distribution. Figure 18–8 illustrates the three-tier model.

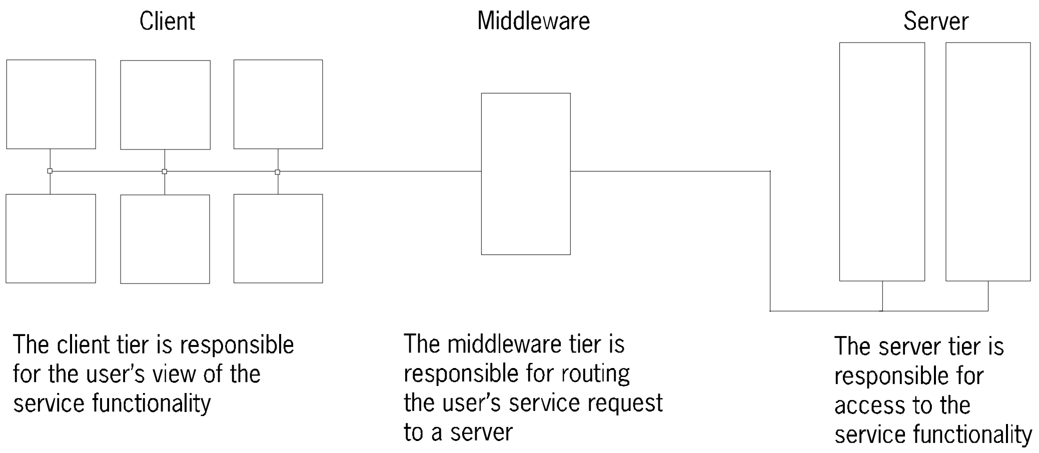
Client
Middleware
Server
The client tier is responsible for the user's view of the service functionality
The middleware tier is responsible for routing the user's service request to a server
The server tier is responsible for access to the service functionality

Figure 18–8. Three-Tier Model

The three-tier model is more complex than the two-tier model, but it is also more flexible and robust. There are many ways the models can be mapped to application architectures. In simple cases, one program can contain the whole model. In the more complex cases, each tier is represented by a separate program and the application uses different services so a program is part of several distribution models. Libraries can be used as the interaction mechanism, but they are not the only interaction mechanism in situations where programs interact. In designing such a system, it is important to recognize which role each side of a library linkage has to fulfill, and what the implementation choices are. An implementation should avoid combining these roles whenever possible. If a library implements two services, there should be separate interfaces to each of the services.

Client Strategies

Clients are typically simple in structure and many in number. They usually need a single access to a service and they initiate the link to the service. There are usually no critical sections or other locking issues. Abnormal termination of a client, if a service terminates, is the acceptable or desired action. For more information refer to “Using Global Objects” and “Using Events and Interlocks” in this book.

Client Libraries

This is the simplest form of a client. The program requires a LIBRARY declaration, which defines where the service provider is located, and a collection of library objects, which define the interface to the service. This implementation strategy is supported by all languages that implement libraries.

Examples

The following is an example of an ALGOL LIBRARY declaration:

```
LIBRARY SERVICE (LIBACCESS = BYFUNCTION,
                 FUNCTIONNAME = "SERVICESUPPORT.");
PROCEDURE REQUEST (BUF);
   ARRAY BUF [*];
   LIBRARY SERVICE;

REQUEST (BUF);
```

The following is an example of a COBOL LIBRARY declaration:

```
CHANGE ATTRIBUTE LIBACCESS OF "SERVICE" TO BYFUNCTION.
CHANGE ATTRIBUTE FUNCTIONNAME OF "SERVICE"
   TO "SERVICESUPPORT.".
CALL "REQUEST IN SERVICE" USING BUF
```

Connection Libraries

This advanced client is only available in ALGOL and NEWP. The program requires a connection block TYPE declaration, which holds declarations for library objects making up the interface, and a SINGLE LIBRARY instance declaration of the connection library type.

Example

An example of an ALGOL SINGLE LIBRARY declaration of the connection library type:

```
TYPE CONNECTION BLOCK SERVICETYPE;
   BEGIN
   PROCEDURE REQUEST (BUF);
      ARRAY BUF [*];
      IMPORTED;
   END SERVICETYPE;

SERVICETYPE SINGLE IMPORTING LIBRARY SERVICE
   (LIBACCESS = BYFUNCTION,
    FUNCTIONNAME = "SERVICESUPPORT.");

SERVICE.REQUEST (BUF);
```

The reasons for choosing a connection library over a client library are

- Program modularity

  The interface declaration is contained inside the connection library type declaration.
- Robustness

  A connection library is delinked instead of terminated when a service terminates.
- Exported functionality

  The interface definition specifies the export of library objects and the imported objects. These implement advanced concepts such as callback functions and client handlers for service events. However, using two-way connections can cause performance problems. For more information about these problems, refer to "Hazards of Circular Connections."

Server Strategies

The main reason servers are typically more complex is that they serve multiple clients. Servers usually are initiated upon the first linkage request from a client. There are usually critical sections and other locking issues to consider. Servers must provide graceful termination of clients if a service is terminated. Some simple services are still implemented using a server library.

Server Libraries

This is the simplest form of a server. The program requires an EXPORT list, which defines the collection of available library objects, and a FREEZE statement, which defines the point in the execution logic that the service is available. This implementation strategy is supported by most languages implementing libraries, although in many cases, the FREEZE statement is performed implicitly.

Examples

The following is an example of an ALGOL EXPORT list:

```
PROCEDURE REQUEST (BUF);
   ARRAY BUF [*];
   BEGIN
   . . .
   END REQUEST;
EXPORT REQUEST;

FREEZE (TEMPORARY);
```

The following is an example of an COBOL EXPORT list:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.
       REQUEST.
DATA DIVISION.
WORKING STORAGE SECTION.
   77 BUF PIC X(80).
PROCEDURE DIVISION USING BUF.
. . .
```

Connection Libraries

This is an advanced server, only available in ALGOL and NEWP. The program requires a connection block TYPE declaration, which holds declarations for the LIBRARY OBJECTS that make up the interface, a LIBRARY interface declaration of the connection library type, and a READYCL statement for that library interface.

Example

The following is an example of an ALGOL connection block TYPE declaration:

```
TYPE CONNECTION BLOCK SERVICETYPE;
   BEGIN
   PROCEDURE REQUEST (BUF);
      ARRAY BUF [*];
      BEGIN
      . . .
      END REQUEST;
   EXPORT REQUEST;
   END SERVICETYPE;

SERVICETYPE LIBRARY SERVICE;

READYCL (SERVICE);
```

The reasons for choosing a connection library over a server library are

- Program modularity

  The interface declaration is contained inside the connection library type declaration.
- Exported functionality

  The interface definition specifies the export of library objects in addition to the imported objects. These implement advanced concepts such as callback functions and client handlers for service events. However, using two-way connections can cause performance problems. For more information, refer to "Hazards of Circular Connections."

- Client state information

  Declarations local to the connection block TYPE declaration are instantaneous for each connection, providing an efficient mechanism to track and save client information. To achieve similar function with server libraries, either a mechanism detecting client identity and locating state information is invoked on every client call referencing local state, or the server is implemented as a private library. This assures a server instance per client, but possibly strains system resources and management.

Middleware Strategies

Middleware, the most complex tier, serves multiple servers and clients. A middleware application usually is initiated upon the first linkage request from a client, but can be available constantly and “listening”. There are critical sections and other locking issues to consider, and these applications must provide full client protection if a service terminates. A middleware application is implemented using connection libraries, as there is no solution for implementing middleware using client and server libraries combined.

Connection Libraries

The program requires two connection block TYPE declarations: one for library objects making up the client interface and one for library objects making up the server interface. The middleware application should include a LIBRARY interface declaration of each connection library type and a READYCL statement for the library interface to clients. How client requests are routed to servers depends on the type of the service. Routing is determined on a session basis or on a per-call basis. For session-based connections, the change procedure for the client interface initiates the server connection.

Example

The following is an example of two ALGOL connection block TYPE declarations:

```
TYPE CONNECTION BLOCK SERVICETYPE;
   BEGIN
   INTEGER MY_SERVICE_INDEX; %% Set up in CHANGE
   PROCEDURE REQUEST (BUF);
      ARRAY BUF [*];
      PENDING;
   EXPORT REQUEST;
   END SERVICETYPE;

TYPE CONNECTION BLOCK SERVERTYPE;
   BEGIN
   PROCEDURE REQUEST (BUF);
      ARRAY BUF [*];
      IMPORTED;
   END SERVICETYPE;

SERVICETYPE SINGLE LIBRARY SERVICE
   (LIBACCES = BYFUNCTION,
   FUNCTIONNAME = "SERVICESUPPORT.");
```

```
SERVICETYPE LIBRARY SERVERS;

PROCEDURE SERVICETYPE.REQUEST (BUF);
   ARRAY BUF [*];
   BEGIN
   SERVERS [MY_SERVICE_INDEX].REQUEST (BUF);
   END REQUEST;

READYCL (SERVICE);
```

Linkage Status Control

A CHANGE procedure is required for each library to allow applications to terminate a delinked connection properly. There is no other way to be notified of clients linking to or delinking from your program as it happens or of servers that have delinked and thus become unavailable while your program is using them. Any exported functions that require guaranteed cleanup should use EXCEPTION procedures. This is typical for any locking protocol. No other safe mechanism is guaranteed to be invoked in all cases. The TRY mechanism cannot be invoked for a termination related to a delinkage of the service that is being protected by a TRY statement. If absolute guarantees of cleanup are necessary, an unsafe mechanism should be used, such as a protected EXCEPTION procedure.

Factors Affecting System Performance

Libraries can be complex and difficult to use correctly and, when used incorrectly, can cause a negative effect on system performance. Delinkage and circular connections should be considered carefully when using libraries.

Delinkage Overuse

Dynamic delinkage of libraries is a complex and costly operation. Excessive use of this capability can lead to severe, system-wide performance problems. The more services a program uses simultaneously (including open subports and open databases), the costlier delinking gets. In addition, the larger a process family is, the more costly library delinkages are. Exiting an environment containing and using a library declaration causes a dynamic delink. Application designers should ensure all libraries are declared globally, and that dynamic delinkage is avoided when unnecessary to the application design.

Hazards of Circular Connections

Circular connections are library connections that provide the ability to both import and export objects. They are created either by circularly linked server libraries or by two-way connection libraries. Circular connections provide complex and powerful library structures that provide benefits and disadvantages to both server and client libraries.

The benefits of circular connections include the ability to implement advanced concepts such as callback functions and client handlers for service events.

Circular connections have significant limitations that degrade system performance. Performance degradation results from the increased time required to search through the associated process families to ensure that references and environments belonging to the delinking connections do not persist beyond delinkage. When two-way connections are used, the processor overhead of delinkage is proportionally greater.

Limited Scaling

Library connections create graph structures used by the MCP to denote which stacks or processes need to be searched when each connection is delinked. When applications use circular linkages, they can require that every stack be searched every time one of the connections is delinked. In a large application, this could result in the system seeming to pause and perform no work for several seconds to several minutes. For this reason, avoid circular connections unless they are essential to the design of the application. If circular connections are used, isolate other applications from the environment to reduce the size of the process environment that is searched at delink.

Indirect or Implicit Connections

Two-way connections can be indirect or implicit; that is, two processes can use two separate connection libraries to create separate import and export paths between each other. This creates the same set of interconnected processes that is created if there is only one two-way connection library.

Existing Circular Libraries

Existing circular applications based on the standard client/server library model can be converted to use connection libraries to gain the robustness that they provide. However, the redesign of such applications should attempt to eliminate circularity to avoid the potential performance degradation while incorporating the advantages of connection libraries.

One-Way Connection Libraries

Use one-way connection libraries whenever possible. You can explicitly define a connection library as a one-way connection library using the IMPORTING or EXPORTING modifier to the Connection Library Declaration Statement in both ALGOL and NEWP. This form of library enables a design to take advantage of most of the features of connection libraries without encountering the performance problems that two-way connections and circular linkages can cause. The compilers and the MCP ensure that a linkage operation does not create a two-way import/export relationship with current library and IPC relationships.

Use of either the IMPORTING or the EXPORTING modifier instructs the compiler to issue a syntax error if an importing connection library contains exports or an exporting connection library contains imports. The MCP returns error –26 at library linkage if an importing or exporting connection library has both imports and exports. This error is also returned by the MCP if an explicit one-way connection library had no imports or exports, or if the importing side of the connection is a server library.

When linking importing and exporting connection libraries, the MCP ensures that only a one-way import or export relationship is established; the context of all of the existing library and IPC relationships for the processes being linked are included in this check. If this one-way relationship check fails, a link error of –7 is returned.

A library link error of –27 is returned if an attempt is made to connect either two importing or two exporting connection libraries.

Application design must still avoid linking one-way connection libraries first and then creating loops by linking to either server libraries or connection libraries not specified as IMPORTING or EXPORTING. Loops are checked for only when connections are made that are explicitly marked as IMPORTING or EXPORTING. This kind of inadvertent circularity can be avoided by only using importing or exporting connection libraries within an application. Such an application will avoid the performance drawbacks inherent in two-way connections and circular libraries.

Examples

The use of the IMPORTING and EXPORTING modifiers is illustrated as follows:

Client

```
TYPE CONNECTION BLOCK SERVICETYPE;
   BEGIN
   PROCEDURE REQUEST (BUF);
      ARRAY BUF [*];
      IMPORTED;
   END SERVICETYPE;

SERVICETYPE SINGLE IMPORTING LIBRARY SERVICE
   (LIBACCESS = BYFUNCTION,
    FUNCTIONNAME = "SERVICESUPPORT.");

SERVICE.REQUEST (BUF);
```

Server

```
 TYPE CONNECTION BLOCK SERVICETYPE;
   BEGIN
   PROCEDURE REQUEST (BUF);
      ARRAY BUF [*];
      BEGIN
      . . .
      END REQUEST;
   EXPORT REQUEST;
   END SERVICETYPE;

SERVICETYPE EXPORTING LIBRARY SERVICE;

READYCL (SERVICE);
```

Security Considerations for Libraries

The same sorts of security considerations apply to both server libraries and connection libraries.

Privileges of the Importing Process

Usercode Privileges

When a process enters an imported procedure, the procedure is executed under the usercode of the importing process, with whatever privileges are defined for the usercode.

Inheriting Privileges from Library Code File

A process can also temporarily assume additional privileges while executing an imported procedure. This is the case because the MP (Mark Program) system command can be used to assign options to a library object code file. These options can confer compiler status, control program status, privileged status, security administrator status, or tasking status. The importing process benefits from these added privileges only while executing procedures imported from that library; it is not enough simply to be linked to the library.

Inheriting Privileges from Importing Process Code File

Alternatively, the MP command can assign privileged transparent status, security administrator transparent status, or tasking transparent status to the library object code file. In this case, library procedures inherit privileges from the object code file of the process that invokes the procedures.

For more information about privileges inherited from usercodes and from object code files, refer to Section 5, “Establishing Process Identity and Privileges.”

Support Library Attributes

You can use the SL (Support Library) system command to assign any of several security-related attributes to an SL function name. These attributes affect any library that is initiated by function with the specified function name.

These attributes should not be confused with the general library attributes discussed under “Using Library Attributes”.

Attribute Descriptions

The following are the support library attributes and their meanings:

- LINKCLASS

  Specifies the linkage class for the library. The linkage class restricts the classes of programs allowed to import objects from that library. Refer to "Using Linkage Classes."

- MCPINIT

  If set, then only operating system processes can initiate a library by linking to the library by function.

  Nonoperating system processes can still cause library initiation by linking to the library by title, or can initiate the library program with a task initiation statement such as CALL, PROCESS, or RUN. However, when a system library is initiated in any of these ways, the system library does not receive the privileges normally granted to system libraries.

  MCPINIT affects only library initiation. If the library has already been initiated by function, then the LINKCLASS and TRUSTED attributes determine whether a given client can link to the library instance by function.

- ONEONLY

  If set, only one version of the library code file is frozen as a BYFUNCTION library at any one time.

- SYSTEMFILE

  If set, the library code file is made a nonremovable system file when initiated or marked as a support library by an SL command.

- TRUSTED

  If set, the library can specify a different linkage class for each export object. Refer to "Using Linkage Classes."

Attributes Preset by the Operating System

Note that some system libraries have selected library attributes assigned to them automatically by the operating system. You can use the SL command to set any Boolean attribute of a system library. However, if the operating system has already set a Boolean attribute, you cannot use the SL command to reset that attribute. The Boolean attributes are MCPINIT, ONEONLY, SYSTEMFILE, and TRUSTED.

Changing a Preset LINKCLASS

If the LINKCLASS value of a system library is set by the operating system, then you can use an SL command to change the LINKCLASS only to values that are able to link to the system-defined LINKCLASS. For example, you can use an SL command to change the LINKCLASS from 2 to 1, but not from 2 to 3.

Error for Incorrect Attribute Assignments

If an SL command assigns attributes to a system library that violate the preceding rules, the system displays the error message "LIBRARY ATTRIBUTES NOT CHANGEABLE."

Nonresumable Libraries

The operating system treats certain system libraries as *nonresumable*. Such libraries cannot be thawed by THAW (Thaw Frozen Library) commands or discontinued by DS (Discontinue) commands. Most nonresumable libraries support specialized commands for the operator to thaw and resume the library. For example, an operator can use the PS QUIT system command to thaw and resume the PRINTSUPPORT system library.

Restricting Access to a Library Code File

Users of a library can be restricted using the SECURITYTYPE file attribute of the library's object code file. If the SECURITYTYPE value is PRIVATE, then any nonprivileged process that uses the library must have the same usercode as the library. If the SECURITYTYPE value is GUARDED or CONTROLLED, then a guard file is used to restrict the nonprivileged library users.

Using Linkage Classes

Linkage Classes

Every process and every exported library object has a linkage class. Whenever a process links to an exported library object the system compares the linkage classes of the importing process and the exporting library object. If the linkage classes are not compatible, the system prevents the object from being imported and returns a linkage error.

Library linkage classes provide a level of protection that goes beyond that provided by the SECURITYTYPE file attribute. Even if the SECURITYTYPE value is PUBLIC, the linkage class still prevents certain processes from importing objects from the library. Linkage classes control several layers of system software services. By applying linkage classes, less privileged applications and system software processes are prevented from linking to more privileged system library processes.

Trusted Libraries

A system library process can have a trusted status. This status allows the library to export library objects with a linkage class different from its own linkage class. Typically, this differing linkage class is less privileged and incompatible with its own linkage class. Only trusted library processes can export library objects into an incompatible linkage class, thus enabling them to offer their services to processes in otherwise incompatible linkage classes.

Determining the Importing Linkage Class

The system determines an importing linkage class for a program when the program is first initiated. The importing linkage class remains the same until the program terminates.

The system uses the following factors to determine the importing linkage class of a process:

- LINKCLASS support library attribute

  This attribute affects only library programs that are initiated by function. If an existing library instance is linked to by function, and the linkage class of the library instance is less privileged than the LINKCLASS of the function name, then the system displays the warning message "LIBRARY INITIATED IN INCOMPATIBLE LINKAGE CLASS."

- MCS status, tasking status, privileged program status, and compiler status

  For information about how a process receives these types of security status, refer to Section 5, "Establishing Process Identity and Privileges."

- Multiple linkage classes

  If a process qualifies for two or more different linkage classes, the system grants the linkage class that affords the greatest privileges, except for compilers which only have compiler linkage class even if marked Tasking or PU. A process can inquire on its own importing linkage class through field [19:04] in the LIBRARYSTATE task attribute.

Determining the Trusted Status

The system determines the trusted status for each library program on the system when the program is first initiated. The trusted status remains the same until the program terminates.

The system uses the following factors to determine the trusted status of a library process:

- The TRUSTED support library attribute. This attribute affects only library programs that are initiated by function.
- MCS status, tasking status, privileged program status, or compiler status. These types of security status give trusted status to a process. Refer to Section 5, "Establishing Process Identity and Privileges" for more information about how a process receives these types of security status.

A process can inquire on its own trusted status through field [26:01] in the LIBRARYSTATE task attribute.

Determining the Exporting Linkage Class

The system determines an exporting linkage class for an exported library object of a program when the program makes the library objects available for linkage. The exporting linkage class remains the same until the library object is no longer available. The system uses the following factors to determine the exporting linkage class of an exported library object:

- The trusted status of the library process. If a library process does not have trusted status, the system treats all the library objects as having the same linkage class as the importing linkage class of the library process.
- The LINKCLASS export list modifier. In ALGOL and NEWP libraries you can use the export list to specify the linkage class for each library object. If the export list does not specify a linkage class for a particular object, the object defaults to zero.
- The importing linkage class of a program. As a performance optimization, if a process has an importing linkage class of zero, the process is implicitly trusted, since there is no security breach possible.

Programs That Contain Multiple Libraries

The linkage class value evaluations apply to all libraries in that program. For example, suppose a program is a server library program that also contains two connection libraries and a client library. Then,

- If the program has no trusted status, all exported objects in these libraries receive the importing linkage class of the program.
- If the program does have trusted status, all exported objects in the libraries have the same default linkage class value of zero, which can be overridden for individual exported library objects.

If a separate function name is associated with each library in the program, then you could assign a different LINKCLASS value to each function name. However, the only LINKCLASS value that has effect is the one used at initiation time. Therefore, the actual linkage class for the program will depend on which library is linked to first.

Descriptions of the Linkage Classes

Table 18–9 lists the linkage classes and indicates which linkage classes are compatible.

Table 18–9. Linkage Classes

Importing Linkage Class	**Type of Process**	**Exporting Linkage Class**
0	Default	0
1	The following processes belong to this class: • Master control program (MCP) • Libraries initiated BYFUNCTION, using a function name that has LINKCLASS = 1. This category includes some system libraries such as COMSSUPPORT, DATACOMSUPPORT, and PRINTSUPPORT	Any
2	The following processes belong to this class: • Libraries initiated BYFUNCTION, using a function name that has LINKCLASS = 2 • Message control systems (MCSs) • Tasking programs	0, 2, 3, 4
3	This class consists of libraries initiated BYFUNCTION, using a function name that has LINKCLASS = 3	0, 3, 4
4	The following processes belong to this class: • Libraries initiated BYFUNCTION, using a function name that has LINKCLASS = 4 • Programs marked as privileged by the PU option of the MP (Mark Program) system command	0, 4
5	The following processes belong to this class: • Libraries initiated BYFUNCTION, using a function name that has LINKCLASS = 5 • Programs marked as compilers by the COMPILER option of the MP (Mark Program) system command.	0, 5
6-7	Reserved for use by system software.	
8-15	Free for site-dependent definition and use.	Any other site-defined linkage class of equal or lesser value

Linkage Class Errors

A process might be able to link to a library even if that library is of an incompatible linkage class. The system evaluates the linkage classes only when the importing process attempts to use an import object. At this point, if the linkage classes are incompatible, the system discontinues the importing process and displays the following error message:

```
OBJECT <name> LINKAGE CLASS VIOLATION IN LIBRARY <name>
```

The importing process could avoid this error by checking the ISVALID function before using the import object. Refer to "Matching Client and Server Library Objects" and "Matching Connection Library Objects" earlier in this section.

For implicit linkages, the above error can occur at linkage time rather than later. This is possible because implicit linkages result when a process uses an import object from a library that is not yet linked. If this import is of a linkage class that is incompatible with the matching export object, then linkage fails.

System Libraries

Libraries with a linkage class other than 0 are referred to as *system libraries*. Much of the system software is provided in the form of system libraries. Examples of system libraries are GENERALSUPPORT, which provides intrinsic functions; MARCSUPPORT, which provides Menu-Assisted Resource Control (MARC); and DSSSUPPORT, which supports distributed system services (DSSs).

Library Debugging

LIBRARIES Option for Program Dumps

If you are debugging a library application, and you are not sure whether an observed bug originates in an exported procedure or in the process that invoked a library procedure, then it can be helpful to set the LIBRARIES option of the OPTION task attribute in the importing process. The LIBRARIES option causes information related to libraries to be included in any program dumps generated by the importing process. This information includes

- The contents of all library process stacks to which the importing process is linked. For a server library, the library process stack is dumped from the base of the FREEZE environment down to the base of the stack. For a connection library, the library process stack is dumped from the highest environment containing a ready connection library interface down to the base of the stack. If no interfaces have been readied from the program, then the process stack will not be dumped even if there is a linked connection library element.
- The contents of the *library directory* in each library process stack. There is one library directory for each export list in a library. For server libraries, only one library directory is in effect at the time a library freezes. For connection libraries, multiple library directories can be in effect at the same time. For each export object, the library directory stores the name, the type of object, and the type of provision used (direct, indirect, or dynamic). For exported procedures, the library directory also stores a description of the parameters of the procedure.
- The contents of all *library templates* in the importing process stack. One library template exists for each library declaration executed by the importing process. A library template stores information about the library attributes. A library template also stores descriptions of all the objects imported from a particular library, including the name, type of object, and parameters.

LIBTRACE System Command

If you encounter a system software bug related to libraries, it might be a good idea to use the LIBTRACE (Library Trace) system command to enable the STANDARD library trace option. The STANDARD option causes the system to write a useful subset of library trace information to the system log. If you succeed in reproducing the bug, submit the system log containing the trace information along with the User Communication Form (UCF). The LIBTRACE system command is available only on systems running diagnostic versions of the MCP.

After you finish reproducing the bug and submitting your UCF, you can reduce system overhead by disabling the STANDARD library trace option.

Server Library Examples

The following subsections give examples of server libraries and client programs that import objects from these libraries.

ALGOL Library: OBJECT/FILEMANAGER/LIB

The following library, called OBJECT/FILEMANAGER/LIB, uses dynamic provision to export a set of file management routines. This library demonstrates features of dynamic provision, but does not necessarily represent efficient programming.

```
$SHARING = PRIVATE
  BEGIN                            % FILEMANAGER/LIB.
  TASK ARRAY LIBTASKS [0:10];      % PROVIDE UP TO 11 DIFFERENT LIBRARY
                                   % PROCESSES.
  STRING ARRAY FILETITLES [0:10];  % LIBPARAMETER FOR EACH LIB PROCESS

  PROCEDURE FILEMANAGER (TASKINDEX);
    VALUE   TASKINDEX;
    INTEGER TASKINDEX;

  BEGIN
    PROCEDURE READFILE;
    BEGIN
    .
    .
    .
    END READFILE;

    PROCEDURE WRITEFILE;
    BEGIN
    .
    .
    .
    END WRITEFILE;

    EXPORT READFILE, WRITEFILE;

    FREEZE (TEMPORARY);
    FILETITLES [TASKINDEX]:= ".";
  END FILEMANAGER;

  PROCEDURE SELECTION (USERSFILE, MCPCHECK);
    VALUE  USERSFILE;
    EBCDIC STRING USERSFILE;
    PROCEDURE MCPCHECK (T); TASK T; FORMAL;
  BEGIN
    INTEGER TASKINDEX;
    BOOLEAN FOUND;
```

```
LABEL SEARCH;

  % LOOK AT ALL THE FILETITLES CHECKING TO SEE IF A LIBRARY PROCESS
  % HAS ALREADY BEEN INITIATED FOR FILE TITLE USERSFILE.

SEARCH:

  WHILE NOT FOUND AND (TASKINDEX LEQ 10) DO
  BEGIN
    IF FILETITLES [TASKINDEX] = USERSFILE THEN
       FOUND:= TRUE
    ELSE
       TASKINDEX:= * + 1;
  END;

  % IF NO LIBRARY PROCESS EXISTS FOR THIS FILE TITLE, THEN CREATE ONE
  IF NOT FOUND THEN
  BEGIN
      TASKINDEX:= 0;
      WHILE NOT FOUND AND (TASKINDEX LEQ 10) DO
        IF LIBTASKS [TASKINDEX].STATUS LEQ 0 THEN
          FOUND:= TRUE
        ELSE
          TASKINDEX:= * + 1;
      IF NOT FOUND THEN
      BEGIN

        % WAIT A SECOND AND MAYBE
        % A LIBRARY PROCESS WILL GO TO EOT.
        WAIT ((1));
        GO SEARCH; %
      END;

    PROCESS FILEMANAGER (TASKINDEX) [LIBTASKS [TASKINDEX]];
    WHILE LIBTASKS [TASKINDEX].STATUS NEQ VALUE (FROZEN) DO
      WAIT ((1));
    FILETITLES [TASKINDEX]:=USERSFILE;
  END;

  MCPCHECK (LIBTASKS [TASKINDEX]);
END SELECTION;

PROCEDURE READFILE;
  BY CALLING SELECTION;
PROCEDURE WRITEFILE;
  BY CALLING SELECTION;

EXPORT READFILE, WRITEFILE;
FREEZE (TEMPORARY);
END LIBRARY.
```

Before attempting to understand this example, you should be familiar with the concepts discussed under “Dynamic Provision” in this section.

OBJECT/FILEMANAGER/LIB exports two procedures: READFILE and WRITEFILE. OBJECT/FILEMANAGER/LIB provides these procedures dynamically. The procedures are ultimately provided by various library processes that are initiated by OBJECT/FILEMANAGER/LIB. Each of these offspring library processes is an instance of the procedure FILEMANAGER. Each library process is intended to provide read and write access to a different data file. Each client process is expected to use the LIBPARAMETER library attribute to indicate the name of the file to be read or written.

The system automatically invokes the SELECTION procedure whenever a client process first links to OBJECT/FILEMANAGER/LIB. The system passes an MCP procedure to the MCPCHECK parameter of the SELECTION procedure. The system also passes the LIBPARAMETER attribute specified by the client process to the USERSFILE parameter of the SELECTION procedure.

The SELECTION procedure searches the string array FILETITLES to see if a library process with the name specified by USERSFILE is already running. If so, the SELECTION procedure selects the task variable of the requested library process. If no library process with the requested name is yet running, SELECTION initiates a new library process and stores the name of the process in the FILETITLES array.

After SELECTION has selected a task variable, it invokes the procedure MCPCHECK, passing the selected task variable as a parameter. The MCPCHECK procedure informs the system to link the client process to the library process with the specified task variable. The actual library linkage is not performed until SELECTION has been exited.

ALGOL Client Program #1

The following ALGOL client program invokes the ALGOL dynamic library in the previous example, OBJECT/FILEMANAGER/LIB. This client program reads information from the file MYFILE by assigning a value of "MYFILE" to the LIBPARAMETER library attribute and then invoking the procedure READFILE. The client program then writes information to a file called OTHERFILE by canceling the library, changing the LIBPARAMETER library attribute to "OTHERFILE", and invoking the WRITEFILE procedure.

```
BEGIN
LIBRARY L (TITLE = "OBJECT/FILEMANAGER/LIB.");
PROCEDURE READFILE;
  LIBRARY L;
PROCEDURE WRITEFILE;
  LIBRARY L;

L.LIBPARAMETER:= "MYFILE";

READFILE;

CANCEL (L);

L.LIBPARAMETER:="OTHERFILE"; % LIBPARAMETER CAN BE CHANGED
```

```
                                         % BECAUSE THE LIBRARY HAS BEEN
                                         % CANCELED.

WRITEFILE;
END PROGRAM.
```

ALGOL Library: OBJECT/SAMPLE/LIBRARY

The following ALGOL library, compiled as OBJECT/SAMPLE/LIBRARY, uses direct library linkage:

```
BEGIN
 ARRAY MSG[0:120];

 INTEGER PROCEDURE FACT(N);
      INTEGER N;

      BEGIN
      IF N LSS 1 THEN
           FACT:= 1
      ELSE
           FACT:= N * FACT(N - 1);
      END;  % OF FACT.


 PROCEDURE DATEANDTIME(TOARRAY, WHERE);
      ARRAY TOARRAY[*];
      INTEGER WHERE;

      BEGIN
      REAL T;
      POINTER PTR;

      T:= TIME(7);
      PTR:= POINTER(TOARRAY, 8) + WHERE;
      CASE T.[5:6] OF

           BEGIN
            0: REPLACE PTR:PTR BY "SUNDAY, ";
            1: REPLACE PTR:PTR BY "MONDAY, ";
            2: REPLACE PTR:PTR BY "TUESDAY, ";
            3: REPLACE PTR:PTR BY "WEDNESDAY, ";
            4: REPLACE PTR:PTR BY "THURSDAY, ";
            5: REPLACE PTR:PTR BY "FRIDAY, ";
            6: REPLACE PTR:PTR BY "SATURDAY, ";
           END;
      REPLACE PTR BY T.[35:6] FOR 2 DIGITS, "-",
                     T.[29:6] FOR 2 DIGITS, "-",
                     T.[47:12] FOR 4 DIGITS, ", ",
                     T.[23:6] FOR 2 DIGITS, ":",
                     T.[17:6] FOR 2 DIGITS, ":",
                    T.[11:6] FOR 2 DIGITS;
          END; % OF DATEANDTIME.

     EXPORT FACT, DATEANDTIME AS "DAYTIME";
     REPLACE POINTER(MSG, 8) BY
          "  - SAMPLE LIBRARY STARTED",
          " " FOR 94;
     DATEANDTIME(MSG, 60);
     DISPLAY(MSG);
     FREEZE(TEMPORARY);
     REPLACE POINTER(MSG, 8)+ 19 BY "ENDED  ";
     DATEANDTIME(MSG, 60);
     DISPLAY(MSG);
     END.
```

ALGOL Library: OBJECT/SAMPLE/DYNAMICLIB

The following ALGOL library, compiled as OBJECT/SAMPLE/DYNAMICLIB, uses dynamic and indirect provision. This library references the library OBJECT/SAMPLE/LIBRARY in the preceding example:

```
    BEGIN
     TASK LIB1TASK, LIB2TASK;
     LIBRARY SAMLIB(TITLE= "OBJECT/SAMPLE/LIBRARY.");
     INTEGER PROCEDURE FACT(N);
          INTEGER N;
          LIBRARY SAMLIB;

     PROCEDURE DYNLIB1;
     % LIBRARY PROVIDED DYNAMICALLY AND INDIRECTLY
          BEGIN % PRINTS DATE WITH TIME.
          LIBRARY SAMLIB (TITLE="OBJECT/SAMPLE/LIBRARY.");
          PROCEDURE DAYTIME(TOARRAY, WHERE);
               ARRAY TOARRAY[*];
               INTEGER WHERE;
               LIBRARY SAMLIB;
          EXPORT DAYTIME;
          FREEZE(TEMPORARY);
          END; % OF DYNLIB1.

     PROCEDURE DYNLIB2;
     % LIBRARY PROVIDED DYNAMICALLY
          BEGIN % PRINTS DATE WITHOUT TIME.
          PROCEDURE DAYTIME(TOARRAY, WHERE);
               ARRAY TOARRAY[*];
               INTEGER WHERE;

               BEGIN
               REAL T;
               T:= TIME(7);
               REPLACE POINTER(TOARRAY, 8) + WHERE
               BY T.[35:6] FOR 2 DIGITS, "-",
                  T.[29:6] FOR 2 DIGITS, "-",
                  T.[47:12] FOR 4 DIGITS;
          END; % OF DAYTIME.
     EXPORT DAYTIME;
     FREEZE(TEMPORARY);
     END; % OF DYNLIB2

% THE SELECTION PROCEDURE.
PROCEDURE THESELECTIONPROC(LIBPARSTR, NAMINGPROC);
     VALUE LIBPARSTR;
     EBCDIC STRING LIBPARSTR;

     PROCEDURE NAMINGPROC(LIBTASK);
          TASK LIBTASK; FORMAL;

     BEGIN
     IF LIBPARSTR EQL "WITH TIME" THEN
          BEGIN
          IF LIB1TASK.STATUS NEQ VALUE(FROZEN) THEN
               PROCESS DYNLIB1 [LIB1TASK];
          NAMINGPROC(LIB1TASK);
          DISPLAY(" *** CALLING DYNLIB1 ");
          END
     ELSE
          BEGIN
          IF LIB2TASK.STATUS NEQ VALUE(FROZEN) THEN
               PROCESS DYNLIB2 [LIB2TASK];
          NAMINGPROC(LIB2TASK);
          DISPLAY(" *** CALLING DYNLIB2 ");
          END;
     END; % OF THE SELECTION PROCEDURE

PROCEDURE DAYTIME(TOARRAY, WHERE);
     ARRAY TOARRAY[*];
     INTEGER WHERE;
     BY CALLING THESELECTIONPROC;

EXPORT  FACT     % PROVIDED INDIRECTLY.
      ,DAYTIME; % PROVIDED DYNAMICALLY
FREEZE(TEMPORARY);
END.
```

ALGOL Client Program #2

The following ALGOL program invokes the library OBJECT/SAMPLE/DYNAMICLIB described previously.

```
BEGIN
 LIBRARY MYLIB(TITLE= "OBJECT/SAMPLE/DYNAMICLIB.");
 INTEGER PROCEDURE FAKTORIAL(N);
      INTEGER  N;
      LIBRARY MYLIB(ACTUALNAME= "FACT");

 PROCEDURE DAYTIME(A, W);
      ARRAY A[*];
      INTEGER W;
      LIBRARY MYLIB;

 REAL T;
 ARRAY DATIME[0:120];

 MYLIB.LIBPARAMETER:= "WITH TIME";
 REPLACE POINTER(DATIME[0], 8) BY
      " 13 FACTORIAL IS ",
      FAKTORIAL(13) FOR 12 DIGITS,
      "     -     ";
 DAYTIME(DATIME[*], 40);
 DISPLAY(DATIME[0]);
 END.
```

ALGOL Circular Client Programs

The following ALGOL programs invoke each other. The first program is the client program.

```
BEGIN
LIBRARY L(TITLE="OBJECT/MYLIB1");
PROCEDURE PLIB1_A (R);
   VALUE R;
   REAL R;
   LIBRARY L;
REAL X;
.......
PLIB1_A (X);
.......
END.
```

The second program (MYLIB1) is a library that uses another library.

```
BEGIN
LIBRARY L(TITLE="OBJECT/MYLIB2");
REAL PROCEDURE PLIB2_A (R);
   VALUE R;
```

```
      REAL R;
      LIBRARY L;

   REAL PROCEDURE PLIB1_A (R);
      VALUE R;
      REAL R;
      BEGIN
    .......
      PLIB1_A:= PLIB2_A (R);    % Invokes a procedure in MYLIB2.
      END;

   PROCEDURE PLIB1_B (P1, P2);
      VALUE P1;
      REAL P1, P2;
      BEGIN
    .......
      END;

   EXPORT PLIB1_A, PLIB1_B;
   FREEZE(TEMPORARY);
   END.
```

The third program (MYLIB2) is a library that uses another library (MYLIB1).

```
   BEGIN
   LIBRARY L(TITLE="OBJECT/MYLIB1");
   REAL PROCEDURE PLIB1_B (A, B);
      VALUE A;
      REAL A, B;
      LIBRARY L;

   REAL PROCEDURE PLIB2_A (R);
      VALUE R;
      REAL R;
      BEGIN
      REAL X, Y;
    .......
      PLIB1_B (X, Y)            % Procedure in MYLIB1; circular linkage
    .......                    % is allowed, because MYLIB1 is frozen.
      PLIB2_A:=Y;
      END;

   BOOLEAN PROCEDURE PLIB2_B (X);
      VALUE X;
      REAL X;
      BEGIN
    .......
      END;

   EXPORT PLIB2_A, PLIB2_B;
   FREEZE(TEMPORARY);
   END.
```

ALGOL Incorrect Circular Libraries

The following are examples of libraries and client programs that use circular linkage incorrectly.

Example 1: Indirect Self Referencing

The following is the client program OBJECT/INDIRECT/CALL. This program invokes procedure X in the library OBJECT/INDIRECT/LIB1.

```
BEGIN
    LIBRARY L(TITLE="OBJECT/INDIRECT/LIB1");
    PROCEDURE X;
      LIBRARY L;
    X;
  END.
```

The following is the library OBJECT/INDIRECT/LIB1. This library provides procedure X indirectly by importing it from another library, OBJECT/INDIRECT/LIB2.

```
$SHARING = SHAREDBYALL
  BEGIN
    LIBRARY L(TITLE="OBJECT/INDIRECT/LIB2");
    PROCEDURE X;
      LIBRARY L;
    EXPORT X;
    FREEZE(PERMANENT);
  END.
```

The following is the library OBJECT/INDIRECT/LIB2. This library also provides procedure X indirectly, in this case by importing procedure X from OBJECT/INDIRECT/LIB1.

```
$SHARING = SHAREDBYALL
  BEGIN
    LIBRARY L(TITLE="OBJECT/INDIRECT/LIB1");
    PROCEDURE X;
      LIBRARY L;
    EXPORT X;
    FREEZE(PERMANENT);
  END.
```

This chain of linkages is completely circular. That is, the chain leads back not just to the original library, but also to the original procedure. When the client program invokes procedure X, the system discontinues the program and displays the message "CURRENT CIRCULAR LIBRARY REFERENCE STRUCTURE IS NOT ALLOWED."

Example 2: Direct Self Referencing

The following ALGOL library, called OBJECT/ALGOL/SELF/LIB, attempts to provide a procedure by importing it from the same procedure in the same library. When a client process attempts to invoke procedure X in this library, the client process hangs. The Y (Status Interrogate) system command shows a STACK STATE of WAITING ON AN EVENT, but the process does not appear in the W (Waiting Mix Entries) system command display. This situation continues until an operator enters a DS (Discontinue) system command or until the system is halt/loaded.

```
$SHARING = SHAREDBYALL
  BEGIN
    LIBRARY L(TITLE="OBJECT/ALGOL/SELF/LIB");
    PROCEDURE X;
       LIBRARY L;
    EXPORT X;
    FREEZE(TEMPORARY);
  END.
```

Example 3: Libraries That Wait on Each Other

The following client program invokes a procedure in the library OBJECT/LIB/WAIT1.

```
BEGIN
    LIBRARY L(TITLE="OBJECT/LIB/WAIT1.");
    PROCEDURE X;
       LIBRARY L;
    X;
  END.
```

The following is the library OBJECT/LIB/WAIT1. Before freezing, this library invokes a procedure in the library OBJECT/LIB/WAIT2.

```
$SHARING = SHAREDBYALL
  BEGIN
    LIBRARY L(TITLE="OBJECT/LIB/WAIT2.");
    PROCEDURE X;
       LIBRARY L;
    PROCEDURE Y;
       DISPLAY ("Y");
    EXPORT X, Y;
    X;
    FREEZE(TEMPORARY);
  END.
```

The following is the library OBJECT/LIB/WAIT2. Before freezing, this library invokes a procedure in the library OBJECT/LIB/WAIT1.

```
$SHARING = SHAREDBYALL
  BEGIN
    LIBRARY L(TITLE="OBJECT/LIB/WAIT1.");
    PROCEDURE Y;
       LIBRARY L;
    PROCEDURE X;
       DISPLAY ("X");
    EXPORT X, Y;
    Y;
    FREEZE(TEMPORARY);
  END.
```

Because OBJECT/LIB/WAIT1 was initiated through the library linkage mechanism, is SHAREDBYALL, and has not yet frozen, OBJECT/LIB/WAIT2 waits for OBJECT/LIB/WAIT1 to freeze. Both libraries are then waiting for each other to freeze. The client process hangs indefinitely. The Y (Status Interrogate) system command shows the client process to have a STACK STATE of WAITING ON AN EVENT, but the client process does not appear in the W (Waiting Mix Entries) system command display. This situation continues until an operator enters a DS (Discontinue) system command or until the system is halt/loaded.

C Library and ALGOL Client Program

The following pair of examples illustrate the ability of an ALGOL calling program to indirectly access character data in a C library.

The following C library exports a procedure named WRITELINE. This procedure accepts a parameter that is a pointer to a string of characters. The procedure writes the string to a file named TEST.

```
#include <stdio.h>
#include <stdlib.h>
FILE * pf;

asm WRITELINE(char * pc) {
        fputs(pc, pf);
        fputc('\n',pf);
}

void cleanup(void) {
        /* called after main exits, i.e., after thaw */
        fputs("all done\n", pf);
        fclose(pf);
}
```

```
main() {
        pf = fopen("TEST", "w");
        atexit(cleanup);
}
```

The C library is called by the following ALGOL client program:

```
BEGIN

LIBRARY CLIB(LIBACCESS = BYTITLE, TITLE="OBJECT/STREAM/C.");
INTEGER PROCEDURE MALLOC(BYTES);
   VALUE                BYTES;
   INTEGER              BYTES;
   LIBRARY CLIB;
INTEGER PROCEDURE FREE(CPTR);
   VALUE              CPTR;
   INTEGER            CPTR;
   LIBRARY CLIB;
INTEGER PROCEDURE HEAPTOPTR(CPTR, APTR);
   VALUE                   CPTR     ;
   INTEGER                 CPTR     ;
   POINTER                       APTR;
   LIBRARY CLIB;
INTEGER PROCEDURE WRITELINE(CPTR);
   VALUE                   CPTR;
   INTEGER                 CPTR;
   LIBRARY CLIB;

PROCEDURE XFER(S);
   VALUE      S;
   STRING     S;
BEGIN
   POINTER APTR;
   INTEGER CPTR;
   CPTR:=MALLOC (LENGTH(S) + 1);
   HEAPTOPTR(CPTR, APTR);
   REPLACE APTR BY S, 48 "00";
   WRITELINE(CPTR);
   FREE     (CPTR);
END XFER;

XFER("HELLO WORLD");
XFER("THIS IS AN EXAMPLE");

END.
```

In addition to importing WRITELINE from the C library, this program also imports the procedures MALLOC, HEAPTOPTR, and FREE. These procedures are implicitly created by the *#include <stdlib.h>* statement in the C library, and are referred to in C as *__malloc_t*, *__heap_to_ptr_t*, and *__free_t*.

The ALGOL program includes the XFER procedure, which accepts a string parameter and writes it to the file TEST by making appropriate calls on the C library. First, XFER invokes the MALLOC procedure, which allocates memory space for the string. MALLOC returns an integer, CPTR, which indicates the position of the string in memory.

CPTR can be used as a pointer only within the C library itself. To write data into the memory area allocated by MALLOC, the ALGOL program must first assign an ALGOL-style pointer to that memory location. The program does this with the call on HEAPTOPTR. The ALGOL program then uses the REPLACE statement to write the string to the area allocated for it.

The ALGOL program then invokes the WRITELINE procedure, passing CPTR as a parameter. The WRITELINE procedure in the C library uses CPTR to locate the string that is to be written to the file.

C Client Program Passing Array to ALGOL Library

The following examples illustrate the C syntax for calling a library. The examples also illustrate the ability to pass arrays between C and ALGOL.

The following is a file named FOO, which is specified by a #include statement in the C client program. This file contains the import declaration for a procedure called LC.

```
extern "ALGOL" void LC (char*, char (&)[ ], int, int&, __heap_t,
                        __errno_t);
```

In the FOO file, the use of the string "ALGOL" has the following effects on the declaration:

- The return type of void causes the declaration to be interpreted as an untyped procedure rather than an integer procedure.
- Normally, C passes all parameters by value. The string "ALGOL" identifies LC as a non-C procedure, thereby allowing parameters to be passed by reference. Call-by-reference parameters are denoted by the ampersand (&) operator. For example, *int&* matches a call-by-reference integer, and *char (&) []* matches a call-by-reference unbounded EBCDIC array.
- The string "ALGOL" also allows some hidden parameter types to be included in the import procedure declaration. In FOO, the hidden parameters are *__heap_t* and *__errno_t*. These parameters are supplied by the compiler, and are not mentioned in the statement in the C program that invokes the imported procedure. The *__heap_t* parameter passes the C program's heap as an EBCDIC array with 0 lower bound. The *__errno_t* parameter passes the predeclared global variable *errno* as a call-by-name integer.

Notice that the import procedure name, LC, is given in all uppercase. This is because many languages, including ALGOL, do not allow library objects to be exported with names containing lowercase letters.

The following is the C client program FOO/C.

```
#include <stdio.h>
#include "foo" (bytitle="OBJECT/FOO/A", intname="FOO")

main (int argc, char *argv []) {
   char buf [10];
   int  i, len;

   for (i=1; i<argc; i++) {
      errno = 0;
      LC (argv [i], buf, sizeof (buf), len);
      if (errno!= 0) {
         printf ("string too long: %d > %d\n", len-1, sizeof (buf)-1);
         errno = 0;
      } else {
         printf ("(%2d) \"%s\" -> \"%s\"\n", len-1, argv [i], buf);
      }
   }
}
```

The *#include "foo"* line in this client program serves as the library declaration. This program calls the imported procedure LC to convert some text to lowercase letters. Notice that the invocation of LC has only four parameters supplied. The compiler automatically supplies parameters for *__heap_t* and *__errno_t* as described previously.

After invoking LC, the program uses the variable *errno* to read and update the error value returned by the library in the *__errno_t* parameter. Notice that the C program initializes *errno* to 0 before calling LC. The library functions responsible for assigning *errno* only modify the *errno* value if an error occurs. These functions do not automatically assign a value of 0 to *errno* when a valid result occurs, as the 0 would overwrite any value left to record a previous error.

The following is the ALGOL library FOO/A, which is used by the C program FOO/C.

```
BEGIN
   PROCEDURE LC (PTR, BUF, BUFSIZE, LEN, HEAP, ERRNO);
      VALUE PTR, BUFSIZE;
      INTEGER PTR, BUFSIZE;
      EBCDIC ARRAY BUF [*], HEAP[0];
      INTEGER LEN, ERRNO;
    BEGIN
      DEFINE MAX_LEN = 65536 #;
      POINTER P;
      INTEGER L;
      TRANSLATETABLE DOWNCASE(EBCDIC TO EBCDIC,
         "ABCDEFGHIJKLMNOPQRSTUVWXYZ" TO "abcdefghijklmnopqrstuvwxyz");
      SCAN P:HEAP [PTR] FOR L:MAX_LEN UNTIL = 48"00";
      LEN:= MAX_LEN - L + 1; % +1 for trailing null
      IF LEN > BUFSIZE THEN BEGIN
         ERRNO:= 1;
      END ELSE BEGIN
```

```
          REPLACE BUF [0] BY HEAP [PTR] FOR LEN WITH DOWNCASE;
        END IF;
      END;

      EXPORT LC;
      FREEZE(TEMPORARY);
  END.
```

In this example, the BUF parameter is specified as unbounded in order to match the *char (&) []* parameter in the FOO file. The HEAP parameter is specified with a lower bound of 0 because C passes the *__heap_t* parameter this way.

Note that the PTR parameter is declared as an integer. The program is able to use PTR as a pointer by including it in the pointer expression HEAP [PTR]. This technique works reliably only if the MEMORY_MODEL compiler control option in the C program specifies a one-dimensional memory model. If the C program had specified a two-dimensional memory model, this ALGOL program would have to use the HEAPTOPTR function to convert the integer value to a pointer.

The SCAN statement and the LEN assignment statement both rely on the fact that C terminates each string with a null character.

As noted in the description of the C program, this procedure assigns a value to ERRNO only if an error is encountered.

C Client Program Passing File to ALGOL Library

The following is an ALGOL library called FOO/FILE/A:

```
  BEGIN
     INTEGER PROCEDURE STATIONNAME (F, MALLOC, COPYTOPTR);
        FILE F;
        INTEGER PROCEDURE MALLOC (SIZE);
           VALUE SIZE;
           INTEGER SIZE;
           FORMAL;
        PROCEDURE COPYTOPTR (LEN, BUF, OFF, PTR);
           VALUE LEN, OFF, PTR;
           INTEGER LEN, OFF, PTR; EBCDIC ARRAY BUF [*];
           FORMAL;
     BEGIN
        EBCDIC ARRAY T [0:300];
        POINTER P;
        INTEGER PTR, LEN;
        REPLACE P:T BY F(1).STATIONNAME;
        IF F.ATTERR THEN BEGIN
           REPLACE P:T BY "<attribute error>.";
        END IF;
        REPLACE P-1 BY 48"00";
        LEN:= OFFSET (P);
        PTR:= MALLOC (LEN);
        COPYTOPTR (LEN, T, 0, PTR);
```

```
         STATIONNAME:= PTR;
      END;
      EXPORT STATIONNAME;
      FREEZE (TEMPORARY);
   END.
```

FOO/FILE/A exports a single procedure called STATIONNAME. The STATIONNAME procedure accepts three parameters: a remote file called F, the procedure MALLOC, and the procedure COPYTOPTR. The STATIONNAME procedure reads the STATIONNAME attribute of the remote file into array T. The STATIONNAME procedure uses the MALLOC function to allocate space in the C client program heap. The STATIONNAME procedure then uses the COPYTOPTR function to copy the contents of array T into the C client program heap. The STATIONNAME procedure return value stores the length of the STATIONNAME file attribute value.

The following is the file FOO/FILE/H, which is a header file used by the C client program to declare the imported procedure STATIONNAME:

```
   extern "ALGOL" char *STATIONNAME (__file_t, __malloc_t,
                                     __copy_to_ptr_t);
```

Of the items in this header file, *__file_t* corresponds to parameter F in the ALGOL library; *__malloc_t* corresponds to MALLOC; and *__copy_to_ptr_t* corresponds to COPYTOPTR.

The following is the C client program, FOO/FILE/C:

```
   #include <stdio.h>
   #include "foo.file.h" (bytitle="OBJECT/FOO/FILE/A", intname="LIB")

   main () {
      printf ("this printf implicitly opens stdout.\n");
      printf ("title=\"%s\"\n", STATIONNAME (stdout->_file_no));
   }
```

The first *printf* statement in the C client program implicitly opens the file *stdout* as a remote file. The second *printf* statement implicitly invokes the STATIONNAME procedure and displays the STATIONNAME file attribute value stored in the C client program heap.

Note that the STATIONNAME invocation does not mention the *__malloc_t* and *__copy_to_ptr_t* parameters, because these are automatically passed by the C compiler. However, the STATIONNAME invocation explicitly passes *__file_t* a parameter of type *int*, which the C compiler changes into a parameter of type file, as required by the ALGOL library. The *int* value that is passed should always be extracted from a *FILE** pointer, as shown by the *->_file_no* clause in this example.

COBOL74 Library: OBJECT/SAMPLE4

The following COBOL74 library compiled as OBJECT/SAMPLE4 is referenced in various examples in this section. The entry point to this library is named PROCEDUREDIVISION. This library is PERMANENT by default.

```
IDENTIFICATION DIVISION.
  ENVIRONMENT DIVISION.
  DATA DIVISION.
  WORKING-STORAGE SECTION.
  77 PARAM PIC 9(11) BINARY.
  PROCEDURE DIVISION USING PARAM.
  P1.
       DISPLAY "I AM SAMPLE4".
       DISPLAY "MY PARAMETER IS " PARAM.
       EXIT PROGRAM.
```

COBOL74 Library: OBJECT/SAMPLE5

The following COBOL74 library is referenced in various examples in this section. This library is TEMPORARY; therefore, when it is no longer in use it unfreezes and resumes running as a regular program. The FEDLEVEL option is set to 5 to allow the PROGRAM-ID to be used as the entry point name.

```
$ SET FEDLEVEL = 5
  $ SET TEMPORARY
   IDENTIFICATION DIVISION.
   PROGRAM-ID. ENTRYPOINT.
   ENVIRONMENT DIVISION.
   DATA DIVISION.
   WORKING-STORAGE SECTION.
   77 PARAM PIC 9(11) COMP.
   PROCEDURE DIVISION USING INTEGER (PARAM).
   P1.
        DISPLAY "I AM SAMPLE5".
        DISPLAY "MY PARAMETER IS " PARAM.
        EXIT PROGRAM.
```

COBOL74 Client Program

The following COBOL74 program uses various ALGOL and COBOL74 libraries described in this section:

```
IDENTIFICATION DIVISION.
  ENVIRONMENT DIVISION.
  DATA DIVISION.
  WORKING-STORAGE SECTION.
  77  PARAM1     PIC 9(11) BINARY.
  77  PARAM2     PIC 9(11) COMP.
  77  RETURNVAL1 PIC 9(11) BINARY.
  77  RETURNVAL2 PIC 9(11) COMP.
  01  TOARRAY    BINARY WITH LOWER-BOUNDS.
      03 ELEMENT PIC 9(6)  BINARY OCCURS 13.
  77  WH         PIC 9(11) BINARY.
  PROCEDURE DIVISION.
  P1.

 * CALL COBOL74 LIBRARY NAMED "OBJECT/SAMPLE4"

      CALL "PROCEDUREDIVISION OF OBJECT/SAMPLE4" USING PARAM1.

 * CALL COBOL74  LIBRARY NAMED "OBJECT/SAMPLE4" USING TITLE ATTRIBUTE

      CHANGE ATTRIBUTE TITLE OF "OBJECT/SAMPLE6" TO
           "OBJECT/SAMPLE4.".
      CALL "PROCEDUREDIVISION IN OBJECT/SAMPLE6"
           USING INTEGER (PARAM2).

 * CALL COBOL74 LIBRARY NAMED "OBJECT/SAMPLE5" WHOSE ENTRYPOINT IS
 * NAMED ENTRYPOINT

      CALL "ENTRYPOINT OF OBJECT/SAMPLE5" USING PARAM1.

 * CALL COBOL74 LIBRARY NAMED "OBJECT/SAMPLE4"
 * USING ANSI74 IPC SYNTAX; CANCEL THAT COBOL74 LIBRARY

      CALL "OBJECT/SAMPLE4" USING INTEGER (PARAM2).
      CANCEL "OBJECT/SAMPLE4".

 * CALL DIRECT ALGOL LIBRARY
 * INTERNAL NAME IS "INTLIB"; TITLE IS "OBJECT/SAMPLE/LIBRARY"

      CHANGE ATTRIBUTE TITLE OF "INTLIB"

         TO "OBJECT/SAMPLE/LIBRARY.".

      CALL "FACT OF INTLIB"
         USING PARAM1 GIVING RETURNVAL1.

 * CALL DYNAMIC ALGOL LIBRARY
 * TITLE IS "OBJECT/SAMPLE/DYNAMICLIB"
* SELECTION PROCEDURE PARAMETER IS "WITH TIME"

     CHANGE ATTRIBUTE LIBPARAMETER OF "OBJECT/SAMPLE/DYNAMICLIB"
         TO "WITH NAME".
     CALL "DAYTIME IN OBJECT/SAMPLE/DYNAMICLIB"
        USING TOARRAY, WH.

     STOP RUN.
```

COBOL85 Libraries and Client Program

The following is a COBOL85 library named TASKM/COBOL85/LIBRARY:

```
000100$ RESET LIST SET ERRORLIST LINEINFO
000200 IDENTIFICATION DIVISION.
000300  PROGRAM-ID. EXPLICIT-LIBRARY LIBRARY.
000400  ENVIRONMENT DIVISION.
000500  INPUT-OUTPUT SECTION.
000600  FILE-CONTROL.
000700       SELECT FL-1 ASSIGN TO DISK.
000800  DATA DIVISION.
000900  FILE SECTION.
001000  FD FL-1 GLOBAL.
001100  01 FL-1-REC PIC X(80) GLOBAL.
001200  WORKING-STORAGE SECTION.
001300  PROGRAM-LIBRARY SECTION.
001400  LB EXPLICIT-LIBRARY EXPORT
001500                      ATTRIBUTE SHARING IS SHAREDBYRUNUNIT.
001600  ENTRY PROCEDURE USERCODE.
001700  ENTRY PROCEDURE WRITER.
001800  PROCEDURE DIVISION.
001900  P-1.
002000      OPEN OUTPUT FL-1.
002100      CALL "DISPLAYER".
002200      CALL SYSTEM FREEZE TEMPORARY.
002300      CLOSE       FL-1 SAVE.
002400      EXIT PROGRAM.
002500  IDENTIFICATION DIVISION.
002600  PROGRAM-ID. WRITER.
002700  DATA DIVISION.
002800  PROCEDURE DIVISION.
002900  P-1.
003000       MOVE "DATA WRITTEN FROM LIBRARY" TO FL-1-REC.
003100       WRITE FL-1-REC.
003200       EXIT PROGRAM.
003300  END PROGRAM WRITER.
003400  IDENTIFICATION DIVISION.
003500  PROGRAM-ID. DISPLAYER.
003600 PROCEDURE DIVISION.
003700  P-1.
003800       DISPLAY "THE EXPLICIT LIBRARY HAS BEEN ENTERED."
003900       EXIT PROGRAM.
004000  END PROGRAM DISPLAYER.
004100  IDENTIFICATION DIVISION.
004200  PROGRAM-ID. USERCODE.
004300  DATA DIVISION.
004400  WORKING-STORAGE SECTION.
004500  01 REC-1.
004600     02 BUFFER      PIC X(20).
004700     02 USERNAME    PIC X(20).
```

```
004800     02 DIRECTORIES PIC 9(4) COMP.
004900  01 MAX-DIRECTORIES PIC 9(4) COMP VALUE 17 GLOBAL.
005000  LINKAGE SECTION.
005100  01 STR       PIC X(80).
005200  01 USER-CODE PIC X(20).
005300  PROCEDURE DIVISION USING STR USER-CODE.
005400  P-1.
005500       UNSTRING STR DELIMITED BY "/" OR "(" OR ")"
005600       INTO BUFFER USERNAME TALLYING IN DIRECTORIES.
005700       MOVE USERNAME TO USER-CODE.
005800       IF DIRECTORIES IS GREATER THAN MAX-DIRECTORIES
005900         CALL "ERROR-MESSAGE" USING DIRECTORIES.
006000       EXIT PROGRAM.
006100  IDENTIFICATION DIVISION.
006200  PROGRAM-ID. ERROR-MESSAGE.
006300  DATA DIVISION.
006400  WORKING-STORAGE SECTION.
006500  77 EXTRA-DIRECTORIES PIC 9(4) COMP.
006600  LINKAGE SECTION.
006700  77 TOTAL-DIRECTORIES PIC 9(4) COMP.
006800  PROCEDURE DIVISION USING TOTAL-DIRECTORIES.
006900  P-1.
007000       SUBTRACT MAX-DIRECTORIES FROM   TOTAL-DIRECTORIES
007100                                 GIVING EXTRA-DIRECTORIES.
007200       DISPLAY "THERE WERE " EXTRA-DIRECTORIES
007300                             "EXTRA DIRECTORIES".
007400       EXIT PROGRAM.
007500  END PROGRAM ERROR-MESSAGE.
007600  END PROGRAM USERCODE.
007700  END PROGRAM EXPLICIT-LIBRARY.
```

The program TASKM/COBOL85/LIBRARY illustrates several library features that distinguish COBOL85 libraries from libraries in earlier COBOL implementations. These features include:

- An explicit FREEZE statement at line 2200, which includes the freeze duration option of TEMPORARY.
- The PROGRAM-LIBRARY SECTION, which includes an explicit export declaration at lines 1400-1700. The declaration specifies a sharing option of SHAREDBYRUNUNIT and lists USERCODE and WRITER as the names of nested programs to be exported.
- Three nested programs, including the two specified in the export declaration: USERCODE, at lines 4100-7600; and WRITER, at lines 2500-3300.
- Local variables. The exported nested program USERCODE includes declarations of the data items REC-1 at line 4500 and MAX-DIRECTORIES at 4900. Note that these local variables are reinitialized each time the USERCODE nested program is invoked. You can cause the values of these variables to be preserved by adding an *IS INITIAL* clause to the PROGRAM-ID paragraph at line 4200.

The following is another COBOL85 library called TASKM/COBOL85/PROCEDURE:

```
000100$RESET LIST SET ERRORLIST LINEINFO
000200$SHARING = SHAREDBYRUNUNIT
000300$LIBRARYPROG
000400  IDENTIFICATION DIVISION.
000500  PROGRAM-ID. IMPLICIT-LIBRARY.
000600  ENVIRONMENT DIVISION.
000700  DATA DIVISION.
000800  WORKING-STORAGE SECTION.
000900  77 SWAP PIC X(10).
001000  LINKAGE SECTION.
001100  01 A-REC.
001200     02 FLD-1 PIC X(10).
001300     02 FLD-2 PIC X(10).
001400  PROCEDURE DIVISION USING A-REC.
001500  P-1.
001600        MOVE FLD-1 TO SWAP.
001700        MOVE FLD-2 TO FLD-1.
001800        MOVE SWAP  TO FLD-2.
001900        EXIT PROGRAM.
```

The library TASKM/COBOL85/PROCEDURE is designed to function much as a COBOL74 library. There is no explicit export declaration or freeze statement, and the entire PROCEDURE DIVISION is exported. However, unlike COBOL74, COBOL85 requires the LIBRARYPROG compiler option to be set in order to indicate that the program will function as a library.

The following is a COBOL85 client program called TASKM/COBOL85/PROGRAM. This program calls the two COBOL85 libraries described previously:

```
000100$ RESET LIST SET ERRORLIST LINEINFO
000200  IDENTIFICATION DIVISION.
000300  PROGRAM-ID. DRIVER.
000400  ENVIRONMENT DIVISION.
000900  DATA DIVISION.
001500  WORKING-STORAGE SECTION.
001600  01 THE-TITLE PIC X(80).
001700  01 USERCODES.
001800     02 USERCODE-1 PIC X(20).
001900     02 USERCODE-2 PIC X(20).
002000  LOCAL-STORAGE SECTION.
002100  LD PARAM-1.
002200  01 P-REC-1.
002300     02 A-FLD-1 PIC X(10).
002400     02 A-FLD-2 PIC X(10).
002500  LD PARAM-2.
002600  01 P-REC-2 PIC X(80).
002700  01 P-REC-3 PIC X(20).
003100  PROGRAM-LIBRARY SECTION.
003200  LB LIB-ONE IMPORT
003300            ATTRIBUTE FUNCTIONNAME IS "THELIBRARY"
```

```
003400                       LIBACCESS    IS BYTITLE.
003500  ENTRY PROCEDURE WRITER.
003600  ENTRY PROCEDURE USERCODE WITH  PARAM-2
003700                           USING P-REC-2 P-REC-3.
003800  LB LIB-TWO IMPORT
003900             ATTRIBUTE TITLE IS "OBJECT/TASKM/COBOL85/PROCEDURE".
004000  ENTRY PROCEDURE PROCEDUREDIVISION WITH  PARAM-1
004100                                    USING P-REC-1.
004500  PROCEDURE DIVISION.
004600  P-1.
004700      CHANGE ATTRIBUTE TITLE OF LIB-ONE
004800                             TO "OBJECT/TASKM/COBOL85/LIBRARY".
005000      CALL WRITER.
005100      CALL WRITER OF LIB-ONE.
005200      MOVE ATTRIBUTE TITLE OF LIB-TWO TO THE-TITLE.
005300      CALL USERCODE
005400           USING THE-TITLE USERCODE-1.
005500      MOVE ATTRIBUTE TITLE OF LIB-ONE TO THE-TITLE.
005600      CALL "USERCODE IN OBJECT/TASKM/COBOL85/LIBRARY"
005700           USING THE-TITLE USERCODE-2.
005800      CALL PROCEDUREDIVISION OF LIB-TWO
005900           USING USERCODES.
006000      CALL "OBJECT/TASKM/COBOL85/PROCEDURE"
006100           USING USERCODES.
006200      DISPLAY USERCODES.
006300      STOP RUN.
```

The program TASKM/COBOL85/PROGRAM illustrates the explicit library declarations and import declarations provided by COBOL85. Thus, the PROGRAM-LIBRARY SECTION at lines 3100-4400 includes declarations of the libraries LIB-ONE and LIB-TWO. These library declarations include library attribute assignments as well as import declarations for WRITER, USERCODE, and PROCEDUREDIVISION.

TASKM/COBOL85/PROGRAM includes examples of the following types of CALL statements:

- A CALL statement that invokes an explicitly declared import object. The statements at lines 5000 and 5300-5400 are examples that invoke import objects declared in the PROGRAM-LIBRARY SECTION.
- A CALL statement that invokes an explicitly declared import object in an explicitly specified library. The statements at lines 5100 and 5800 refer to the LIB-ONE and LIB-TWO declarations in the PROGRAM-LIBRARY SECTION.
- A CALL statement that uses a string literal to specify the library object code file title. The statement at line 6000 invokes the library TASKM/COBOL85/PROCEDURE. The CALL statement does not need to specify the name of a particular import object, as the library in question exports only the PROCEDUREDIVISION.
- A CALL statement that uses a string literal to specify both the library object code file title and the import object name. The statement at line 5600 invokes the object USERCODE in the library OBJECT/TASKM/COBOL85/LIBRARY.

FORTRAN77 Library and Client Program

The following examples present the FORTRAN77 versions of the library program and client program.

The following FORTRAN77 program, compiled as MATHINTRINSICS, creates a library.

```
    BLOCK GLOBALS
FILE  6(KIND="PRINTER")
         EXPORT (SINE="SIN",COSINE)
      END
      REAL FUNCTION SINE(X)
C*         PERFORM SINE CALCULATION...
           SINE=SIN(X)
      END
      REAL FUNCTION COSINE(X)
C*         PERFORM COSINE CALCULATION...
           COSINE=COS(X)
      END
C*    MAIN PROGRAM
           WRITE(6,*)SINE(X),COSINE(X)
           CALL FREEZE("TEMPORARY")
      END
```

The following FORTRAN77 program invokes the FORTRAN77 library MATHINTRINSICS previously described:

```
    BLOCK GLOBALS
FILE  5(KIND="REMOTE")
FILE  6(KIND="PRINTER")
           LIBRARY LIB1(TITLE="MATHINTRINSICS",
*                        INTNAME="MATHINTRINSICS")
      END
      REAL FUNCTION SIN(X)
           REAL X
           IN LIBRARY LIB1
      END
      REAL FUNCTION COS(X)
           REAL X
           IN LIBRARY LIB1(ACTUALNAME="COSINE")
      END
      PROGRAM MAIN_PROGRAM
           EXTERNAL SIN, COS
           READ(5,*)X
           WRITE(6,*)SIN(X),COS(X)
      END
```

Pascal Library

The following is an example of a Pascal library.

```
library lib; usage(sharing = sharedbyall);
     interface
        type vect = array [1..30] of integer;
        procedure sum (vector1, vector2: vect);
        function fact (n: integer): integer;
        function sin (r: real): real;
     end;

     library mylib (title = 'OBJECT/ARITHLIB');
     procedure sum;
        var i: integer;
        begin
        for i:= 1 to 30 do
           vector[i]:= vector1[i] + vector2[i];
        end;
     function sin; mylib;
     function fact;
        begin
        if n < 1 then
           fact:= 1
        else
           fact:= n * fact(n-1);
        end;

     begin
     freeze;
     end.
```

This library exports the procedure *sum* and the functions *fact* and *sin*, all of which appear in the interface part. *Sum* and *fact* are completely declared in the library block. *Sin* is imported from another library.

When an ALGOL library procedure uses an EBCDIC array that was passed from a Pascal client program, the library procedure should specify the starting index of the array. This precaution is necessary because ALGOL processes the following statements slightly differently:

```
REPLACE A BY "HI";
REPLACE A[0] BY "HI";
```

The following statement causes the value "HI" to be correctly assigned to element 0 of array A:

```
REPLACE A[0] BY "HI";
```

The following ALGOL library illustrates the use of these types of statements:

```
$SHARING = PRIVATE
BEGIN

PROCEDURE ALGOLDISPLAY(S1, S2);
  EBCDIC ARRAY S1[*], S2[*]
  BEGIN
  REPLACE S1[0] BY "HI"; % THESE TWO STATEMENTS ARE CORRECT
  DISPLAY(S1[0]);

  REPLACE S2 BY "HI"     % THESE TWO STATEMENTS ARE ALLOWED,
  DISPLAY(S2);          % BUT WILL NOT GIVE THE EXPECTED RESULTS.

  END;

  EXPORT ALGOLDISPLAY;
  FREEZE(TEMPORARY);
END.
```

The following example illustrates how an ALGOL client program should pass an unbounded array to a Pascal library. The ALGOL client program declares the imported procedure PASCPROC with an unbounded array parameter A. The ALGOL client program declares another array, called MYARRAY, for use as the actual parameter. MYARRAY is declared with a lower bound of 0 because a Pascal library always assumes the first array element to be at index 0.

```
BEGIN
   LIBRARY MYLIB;
   PROCEDURE PASCPROC(A);
      EBCDIC ARRAY A[*];
      LIBRARY MYLIB;
   EBCDIC ARRAY MYARRAY [0:5];

   REPLACE MYARRAY BY "DATA";
   PASCPROC(MYARRAY);

END.
```

The following Pascal program invokes the procedure ALGOLDISPLAY in an ALGOL library called OBJECT/ALGOLLIB.

```
program p;
  type
    stringbyte = packed array [1..20] of char;

  library mylib(title= 'OBJECT/ALGOLLIB');
  procedure algoldisplay(s1, s2: stringtype); mylib;

  var
    s1, s2: stringtype;

  begin
  s1:= 'abcdefghijklmnopqrst';
  s2:= '12345678901234567890';

  algoldisplay(s1, s2);
  end.
```

Connection Library Example

The following connection library program serves as a switch, connecting client libraries to any of several similar server libraries. These server libraries each export a procedure called A2.

Note that this example includes only the connection library program; the server libraries and client libraries would be in separate programs, which are not shown here.

```
TYPE CONNECTION BLOCK SERVER_CL_TYPE;
  BEGIN
  PROCEDURE A2;
    IMPORTED;
  END OF SERVER_CL_TYPE;

SERVER_CL_TYPE LIBRARY SERVER_CL(CONNECTIONS = 10,
                                 LIBACCESS = BYFUNCTION,
                                 INTERFACENAME = "SECONDLINK.");

TYPE CONNECTION BLOCK CLIENT_CL_TYPE;
  BEGIN
  INTEGER SERVER_ID,
          SERVER_CL_INDEX;

  PROCEDURE LINK_ME (ID);
    VALUE ID; INTEGER ID;
    BEGIN
    EBCDIC ARRAY A[0:15];
    INTEGER I;

    SERVER_ID:= ID;

    % FIND A VALID CONNECTION
    I:= 1;
    WHILE I LSS LIBRARY(SERVER_CL).CONNECTIONS AND
          SERVER_CL_INDEX = 0         DO
      IF LIBRARY(SERVER_CL[I]).STATE = VALUE(NOTLINKED) THEN
        SERVER_CL_INDEX:= I
      ELSE
        I:= * + 1;
    IF SERVER_CL_INDEX = 0 THEN
      BEGIN
      LIBRARY(SERVER_CL).CONNECTIONS:=
        LIBRARY(SERVER_CL).CONNECTIONS + 1;
      SERVER_CL_INDEX:= LIBRARY(SERVER_CL).CONNECTIONS;
      END;

    REPLACE A BY "SERVERNAME", ID FOR * NUMERIC;
    LINKLIBRARY(SERVER_CL[SERVER_CL_INDEX], FUNCTIONNAME = A);
    END OF LINK_ME;
  PROCEDURE A1;
```

```
      BEGIN
      SERVER_CL[SERVER_CL_INDEX].A2;
      END OF A1;


    EXPORT LINK_ME, A1;
    END OF CLIENT_CL_TYPE;

  CLIENT_CL_TYPE LIBRARY CLIENT_CL(CONNECTIONS = 10,
                                   INTERFACENAME = "FIRSTLINK.");

  RSLT:= READYCL(CLIENT_CL);
```

In this example, the client program submits a linkage request by invoking the procedure LINK_ME in connection library CLIENT_CL. The client program uses parameter ID to procedure LINK_ME to specify which server library to link to. LINK_ME performs the following actions:

- Searches for a CLIENT_CL connection that is not in use, and increases the CONNECTIONS value if necessary.
- Builds a FUNCTIONNAME value out of the string SERVERNAME and the value of parameter ID.
- Links through the selected connection using the selected FUNCTIONNAME value.

Once procedure LINK_ME is finished, the client program can invoke the procedure A1 of connection library CLIENT_CL. Procedure A1 in turn executes a call on procedure A2 of the server library that was linked to previously.

Section 19
Using Shared Files

Files are relevant to interprocess communication (IPC) in two ways:

- Certain kinds of files are intended specifically for use in IPC, and present unique advantages when compared to other IPC techniques.
- Processes can share the same file, even if the file is not used as a medium for IPC. For example, two processes might have a shared responsibility for updating a single file. Even though the file is not used for IPC, the processes must use IPC techniques to ensure that their updates do not conflict.

This section gives an overview of both of these aspects of files and IPC. For more detailed information on many of the topics discussed in this section, refer to the *I/O Subsystem Programming Guide*.

Sharing Communications Files

IPC files enable processes to communicate with other processes in a manner similar to reading or writing a file on a physical device. This makes IPC files an ideal method for transmitting large quantities of textual information between processes.

The processes that communicate through an IPC file do not have to belong to the same process family. The processes specify various file attributes that allow the system to establish a link between the correct pair of processes.

The system supports three types of files for use in IPC: port files, host control (HC) files, and HYPERchannel (HY) files. The following subsections briefly outline the capabilities of each of these types of IPC files.

Using Port Files

Port files enable communication between processes regardless of whether those processes reside on a single host or on separate hosts in a local area network or wide area network. There are several different environments, known as *port providers*, available on your system, which provide port file capabilities. Examples of port providers are BNA Version 2, Open Systems Interconnection (OSI), and Transmission Control Protocol/Internet Protocol (TCP/IP). However, even if none of these port providers are installed on a host, applications on the host can still use port files to communicate with applications on the same host.

Port files have been implemented to provide the applications programmer with a single interface that can be used to communicate within a single host or across all the types of multihost networks supported by A Series systems. However, the programmer can choose among any of several different port services with varying functionality. Each type of port provider supports one or more of these port services. A port service that is available from most port providers is BASICSERVICE. Port file applications using BASICSERVICE can run with little or no modification on BNA Version 2, OSI networks, or a single host that is not part of any network.

The following languages support port file functionality: ALGOL, COBOL74, COBOL85 FORTRAN77, and Pascal. Other languages that support files also access port files; however, for these languages, full support of port file statements might not be available.

A port file consists of one or more distinct communication paths, or *subfiles,* that are grouped under a common name. Individual subfiles are identified by way of the port file name and a number called the *subfile index*. You can specify the number of subfiles associated with a port file by using the MAXSUBFILES file attribute. On systems running BNA Version 2 or TCP/IP, you can establish differing priorities for the subfiles using the DIALOGPRIORITY file attribute.

Before opening a subfile, the application can assign several file attributes that help the system to identify the matching port subfile. These file attributes include FILENAME, which specifies the name of the port file; MYNAME, which must match the YOURNAME file attribute of the matching port subfile; and YOURHOST, which specifies the host where the matching process is running. Other attributes can be used to restrict access to processes having specified usercodes.

An application can use any of several OPEN statement options to specify whether the process waits for a matching process to appear or continues execution immediately. If the process continues execution, it can either abandon the open operation or leave the subfile in an offered state, ready for the matching process to link to it.

Each subfile consists of an *input queue*, from which the process reads, and an *output queue*, to which the process writes. Messages are processed through each queue on a first-in, first-out basis, so they always reflect the chronological order in which they were transmitted. The system provides the event-valued file attributes INPUTEVENT and OUTPUTEVENT to inform the process of activity in the input and output queues.

The process can write messages to individual subfiles, or can use a *broadcast write* statement, which sends the same message to all the subfiles in a port file. Similarly, a process can read messages from a specific subfile, or use a *nonselective read* statement, which reads a message from any one of the subfiles with waiting input.

For a complete explanation of how to use port files, refer to the *I/O Subsystem Programming Guide*.

The following subsections provide simple examples of port file programs written in COBOL74 and ALGOL.

COBOL74 Port File Example

The following COBOL74 program declares a port file called MSSR with three subfiles. This program runs on host SFA15CD and opens the port file with MYNAME = MASTER and YOURHOST = SF59D. The program opens the port and broadcasts a message to all subfiles. The program then performs a nonselective operation read to determine which of the remote processes responded first. It sends a message to the remote process that responded first and a different message to the other remote processes. A “USE AFTER ERROR” procedure causes the program to display a message after any I/O error. Note that this program should be initiated after the three matching processes have been initiated and have attempted to open their subfiles.

```
*COBOL74 READ/WRITE PROGRAM USING BNA OPTIONS.
  *THIS PROGRAM RUNS ON HOST SFA15CD.
  IDENTIFICATION DIVISION.
  PROGRAM-ID. COBOL74-PORTFILE-DEMO2.
  ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
  FILE-CONTROL.
      SELECT MSSR ASSIGN TO PORT
             ACTUAL KEY IS MSSR-KEY
             FILE STATUS IS MSSR-STATUS.
  DATA DIVISION.
  FILE SECTION.
  FD MSSR.
  01 MSSR-REC              PIC X(17).
  WORKING-STORAGE SECTION.
  01 MSSR-WS.
     05 MSSR-STATUS        PIC XX.
     05 MSSR-KEY           PIC 9 COMPUTATIONAL.
  PROCEDURE DIVISION.
  DECLARATIVES.
  IO-ERROR SECTION.
  USE AFTER ERROR PROCEDURE ON I-O.
  IO-ERROR-HANDLER.
       DISPLAY "I/O ERROR ENCOUNTERED".
       DISPLAY "STATUS IS ", MSSR-STATUS.
  END DECLARATIVES.
  THE-PROGRAM SECTION.
  INITIALIZATION.
       CHANGE ATTRIBUTE MAXSUBFILES OF MSSR TO 3.
       CHANGE ATTRIBUTE MYNAME OF MSSR TO "MASTER.".
  *COBOL74 APPLIES THE FOLLOWING ASSIGNMENTS TO ALL SUBFILES.
       CHANGE ATTRIBUTE YOURNAME OF MSSR(0) TO "SERVANT.".
       CHANGE ATTRIBUTE YOURHOST OF MSSR(0) TO "SF59D.".
  OPEN-THE-PORT.
       MOVE 0 TO MSSR-KEY.
       OPEN I-O MSSR.
  BROADCAST-THE-MESSAGE.
       MOVE 0 TO MSSR-KEY.
       MOVE "SEND ME A MESSAGE" TO MSSR-REC.
       WRITE MSSR-REC.
```

```
GET-A-MESSAGE.
     MOVE 0 TO MSSR-KEY.
     READ MSSR.
     DISPLAY "MESSAGE RECEIVED FROM SUBFILE ", MSSR-KEY.
     DISPLAY MSSR-REC.
SEND-A-MESSAGE.
* THE ACTUAL KEY AT THIS POINT CONTAINS THE SUBFILE
* FROM WHICH THE MESSAGE WAS REMOVED IN THE LAST
* READ STATEMENT.
     MOVE "YOU WIN" TO MSSR-REC.
     WRITE MSSR-REC.
CLOSE-THE-WINNING-SUBFILE.
     CLOSE MSSR.
* READ THE MESSAGES FROM THE LOSING SUBFILES.
PERFORM GET-A-MESSAGE.
PERFORM GET-A-MESSAGE.
BROADCAST-A-MESSAGE-TO-LOSERS.
     MOVE "YOU LOSE" TO MSSR-REC.
     MOVE 0 TO MSSR-KEY.
     WRITE MSSR-REC.
CLOSE-THE-LOSERS.
     CLOSE MSSR.
     STOP RUN.
```

ALGOL Port File Example

The following ALGOL program is designed to communicate with the preceding COBOL74 program. If three instances of this program are initiated, they will each communicate with one of the subfiles declared in the COBOL74 program. This ALGOL program runs on host SF59D and opens a port file called MSSR with a single subfile, and with MYNAME = MASTER and YOURHOST = SFA15CD. The program reads the message broadcast from the COBOL74 program, sends a reply, and then reads another message from the COBOL74 program. The read and write statements are handled by the procedures READIT and WRITEIT, which use complex wait statements to monitor the status of the subfiles used.

```
% COMPLEMENTARY ALGOL PROGRAM FOR READ/WRITE TO A PORT FILE.
% THIS PROGRAM IS TO BE EXECUTED ON HOST "SF59D".
BEGIN
FILE MSSR (KIND = PORT, MAXSUBFILES = 1, MAXRECSIZE = 17,
           MYUSE = IO, UNITS = CHARACTERS, MYNAME = "SERVANT.",
           YOURNAME="MASTER.", YOURHOST = "SFA15CD.");
EBCDIC ARRAY INMSS[1:17],OUTMSS[1:14];
BOOLEAN RSLT;
INTEGER INT;
PROCEDURE ABORT(REASON);
STRING REASON;
BEGIN
  DISPLAY (REASON);
  MYSELF.STATUS:= VALUE(TERMINATED);
END;
```

```
PROCEDURE READIT;
BEGIN
   INTEGER EVNT;
   EVNT:= WAIT ((120),MSSR.CHANGEEVENT,MSSR.INPUTEVENT);
   CASE EVNT OF
   BEGIN
      1: ABORT ("TIME LIMIT ELAPSED - NO MESSAGE RECEIVED");
      2: CASE MSSR.FILESTATE OF
         BEGIN
            VALUE(BLOCKED):VALUE(DEACTIVATIONPENDING):
            VALUE(OPENED): VALUE(SHUTTINGDOWN):
                IF HAPPENED (MSSR.INPUTEVENT) THEN
                   RSLT:= READ (MSSR,17,INMSS)
                ELSE ABORT ("NO MESSAGE RECEIVED");
            ELSE: ABORT ("BAD FILESTATE");
         END;
      3: RSLT:= READ (MSSR,17,INMSS);
   END;
   IF RSLT THEN ABORT ("MESSAGE MISCARRIED")
           ELSE DISPLAY (INMSS);
END READIT;

PROCEDURE WRITEIT;
BEGIN
   INTEGER EVNT;
   EVNT:= WAIT ((120),MSSR.CHANGEEVENT,MSSR.OUTPUTEVENT);
   CASE EVNT OF
   BEGIN
      1: ABORT ("TIME LIMIT ELAPSED - NO ROOM TO WRITE");
      2: CASE MSSR.FILESTATE OF
         BEGIN
            VALUE(BLOCKED): VALUE(OFFERED): WRITEIT;
            VALUE(OPENED): IF HAPPENED (MSSR.OUTPUTEVENT) THEN
                           RSLT:= WRITE(MSSR,14,OUTMSS);
            ELSE: ABORT ("BAD FILESTATE");
         END;
      3: RSLT:= WRITE (MSSR,14,OUTMSS);
   END;
   IF RSLT THEN ABORT ("WRITE MISCARRIED");
END WRITEIT;

IF (INT:= OPEN (MSSR)) NEQ 1 THEN
    ABORT("UNABLE TO OPEN SUBFILE");
READIT;
REPLACE OUTMSS BY "REMOTE MESSAGE";
WRITEIT;
READIT;
CLOSE (MSSR);
END.
```

Using Host Control (HC) Files

Host control (HC) files provide a simple, high-speed method for transferring data between processes running on two different hosts in a local area network.

HC file communication takes place through intersystem control (ISC) hardware. Each ISC link consists of an ISC hub, which is connected by cables to a host control data link processor (HCDLP) at each host system.

The ISC link must be dedicated to the exclusive use of a pair of application processes, one on each host. Each process opens a file with a KIND file attribute value of HC and a FILENAME value equal to the ISC hub name. The processes must communicate using direct I/O. Therefore, the application must be written in one of the languages that support HC files and direct I/O. The only languages that support both these features are ALGOL and NEWP.

Compared to port files, HC files have the potential for offering faster data transfer. However, unlike port files, HC files can operate only across a single type of dedicated hardware link, and only in a local area network. Further, HC files lack helpful port file features such as multiple subfiles, subfile matching based on user-supplied names, message rerouting, and data compression.

For additional information, refer to the *I/O Subsystem Programming Guide*.

Using HYPERchannel (HY) Files

HYPERchannel (HY) files are similar to HC files in that HY files allow application processes to communicate in a local area network over dedicated hardware links. In the case of HY files, the links are HYPERchannel coaxial trunks that are connected to a host through an A223 adapter and a HYPERchannel data link processor (HYDLP). An application declares an HY file by specifying a KIND file attribute value of HY. Applications that communicate over the HYPERchannel link must use direct I/O. Therefore, the applications must be written in one of the languages that support HY files and direct I/O. The only languages that support both these features are ALGOL and NEWP.

Compared to port files, HY files offer much the same advantages and liabilities as HC files. That is, HY files can offer a faster data transfer rate than port files, but lack helpful port file features such as multiple subfiles, subfile matching based on user-supplied names, message rerouting, and data compression.

However, the underlying HYPERchannel hardware gives HY files the following advantages over HC files:

- HYPERchannel interfaces are supported by many non-MCP Series systems, including 1100 and 2200 series systems as well as IBM and DEC systems. Thus, HY files enable you to implement applications that communicate across networks of multivendor systems.
- HYPERchannel links can connect systems spread over a distance of several kilometers. By contrast, the ISC links that support HC files are limited to a few hundred feet in length. Thus, HY files can provide communication across a larger distance than HC files can.

For further information about HY files, including examples of programs that use HY files, refer to the *I/O Subsystem Programming Guide*.

Sharing Other Kinds of Files

Port files, HC files, and HY files were designed specifically for use in IPC. The system also supports a number of other file types that are associated with permanent storage media, such as disk or tape. These file types are not well suited for use in transmitting information between processes, for two reasons. First, an I/O operation to a peripheral device takes greater elapsed time than operations on port files, which take place largely within main memory. Second, these file types do not provide some of the convenient features of port files, such as separate input and output queues, or event-valued file attributes that notify processes when records have been received.

However, there are cases where it can useful for two or more processes to share the same file, even if the file itself is not being used for IPC. These are cases where several different processes running at the same time are responsible for reading and updating the same file. The file is being used as a permanent storage medium, rather than a method of passing control information between processes.

In this situation, you can use IPC techniques to accomplish two of the goals of file sharing:

- To provide two or more processes with access to the same file
- To regulate timing to prevent these processes from accidentally overwriting each other's changes to the file

For most file types, the only way to achieve these goals is to design the processes so they communicate with the file through the same logical file. The *logical file* is an access structure, created by a file declaration in the program that exists in system memory. By contrast, the *physical file* is the file that exists on a peripheral storage device, such as a disk or tape drive. Opening a file causes the logical file to be linked to a physical file so that the process can read or write information in the file. The following subsections discuss the concept of the logical file and considerations that arise from sharing logical files.

For disk files, it is also possible for processes to use the same physical file without using the same logical file. Considerations for doing this are discussed under "Accessing Disk Files Through Separate Logical Files" later in this section.

Using Shared Logical Files

Processes can use any of the following methods to share logical files:

- An internal task can use logical files declared globally in the parent program, as discussed in Section 15, "Using Global Objects."
- An initiating process can pass a file as a parameter to a process that it initiates, as discussed in Section 17, "Using Parameters."
- A WFL job can declare a file and use a global file equation to cause a task to use this file in place of one declared in the task itself. An example of global file equation is given under "File Sharing Examples" later in this section.
- A shared library can contain a file declaration that is global to the exported library procedures. You can design the exported procedures to allow user processes to access the file declared globally in the library. Refer to "Global Objects in Server Libraries" and "Global Objects in Connection Libraries" in Section 18, "Using Libraries."

If all the processes that use the file belong to the same process family, then you can use any of these sharing methods. If the processes belong to different process families, then using shared libraries is the only method that can enable the logical file to be shared.

Specifying the File Location

For files with KIND = DISK and NEWFILE = FALSE, if the TITLE file attribute of the logical file was not assigned a usercode, then at file-open time the file is searched for under the usercode of the process that declared the file. If the file is not found under that usercode, it is searched for as a nonusercoded file. This search pattern is the same even if the file-open statement is executed by a process different from the process that declared the file.

Similarly, the family where the system searches for the file is affected by the FAMILY task attribute of the process that declares the file, rather than by that of the process that opens the file. If the TITLE file attribute specifies a family, the FAMILY task attribute can specify a substitute family. If the TITLE file attribute does not include a family, the default is DISK, and the FAMILY task attribute can specify a substitute family for DISK.

Synchronizing Access with Shared Logical Files

The design of the I/O hardware prevents any two I/O operations from accessing the same record at the same time. For example, there is no way for a record to be read when a write operation involving the same record is half completed. The read operation is delayed until the write operation finishes.

However, you can prevent other types of synchronization problems only through careful program design. Note that these synchronization concerns arise only in cases where at least one of the processes writes to the shared file. If all the processes simply read from the file, then the order in which they execute their read operations makes no difference. The following are the basic goals of synchronization:

- For cases where one process writes to a particular record and a second process is to read the updated record, the second process should wait until the record is updated before reading it.
- For cases where two processes read and update the same record, the processes should be prevented from accidentally overwriting each other's updates. Such overwriting can occur if both processes read from the record and then both processes write to the record. The second write operation erases the effects of the first, and information can therefore be lost. A mechanism must be established to ensure that the first process completes its update of the record before the second process reads it.

These types of synchronization can be achieved conveniently through the use of events. Events can be shared between processes in the same way that logical files are shared: they can be accessed as global objects by internal tasks, passed as call-by-reference parameters, or accessed through a SHAREDBYALL library.

If it is only necessary to ensure that different processes do not update the file at the same time, the available state of an event can be used. Each process can be designed to procure the event before using the file, and liberate the event afterward.

If it is necessary to ensure that processes access a file in a certain order, the happened state of an event can be used. The second process that is to use the file can wait on an event. When the first process is finished using the file, the process can cause the event.

Establishing Access Rights

Some special security considerations arise from the ability of different processes to share a logical file. The use of a shared logical file can override the file access privileges of some processes. The use of a shared logical file can make a physical file available to processes that would otherwise not have been able to access it. A shared logical file can also prevent a process from using a physical file that it would normally have had access to. You can control the effects of the shared logical file through careful program design.

Before reading the following discussion, you should be familiar with the concepts introduced in Section 5, "Establishing Process Identity and Privileges."

The access rights allowed for a logical file are determined at file-open time. Once the logical file is opened, all the processes that share the logical file have equal access rights to the physical file. This is true even if the processes have different usercodes, accesscodes, names, and security statuses.

When a process attempts to open a file, the system examines the physical file to determine the access rights that can be granted to the logical file. The system evaluates these rights according to one of the following two rules:

- Actor rule

  File access rights are based on the security status and task attributes of the process that opens the file.
- Declarer rule

  File access rights are based on the security status and task attributes of the process that declares the file.

The actor rule was formerly the only method used for file security checking. The declarer rule was introduced to provide an alternative that gives more predictable behavior. By default, the declarer rule is now used to determine file access rights.

You can override the default method of file access by assigning the FILEACCESSRULE task attribute of the process that opens the file. The default value of DECLARER results in file access rights being based on the declarer rule. A value of ACTOR causes file access rights to be based strictly on the actor rule. The value of ACTOR can be assigned only by a privileged process.

The following subsections give examples of how the actor and declarer rules apply to file access through libraries and file access restricted by guard files.

Example: Nonprivileged Library Program

Suppose there is a SHAREDBYALL library program. The library object code file is nonprivileged and has a SECURITYTYPE of PUBLIC. An instance of the library is running under a privileged usercode called U1. This library declares a file with usercode U2. The library also exports procedures that can be used to open, close, read from, and write to the file. It happens that a physical file with the specified title already exists and has a SECURITYTYPE of PRIVATE. There are also a number of user processes, including one called A that has usercode U3 and runs with nonprivileged status, and another user process B that has usercode U4 and runs with privileged status.

Under the actor rule, if the library process attempts to open the file before freezing, then the file open operation is successful and all user processes are able to access the file by way of the exported procedures. The file open operation succeeds because the library runs under a privileged usercode and therefore has the right to access a private file stored under a different usercode.

On the other hand, if the library process freezes before it opens the file, and user process A enters a library procedure that opens the file, then the file open operation fails. This is because user process A is nonprivileged and, therefore, does not have the right to access private files stored under different usercodes. However, if user process B enters the library procedure that opens the file, the file open operation succeeds. Once the file is open, all user processes, including user process A, are able to access the file by way of library procedures.

Under the declarer rule, the file open operation will be successful regardless of whether the library process opens the file directly or exports a procedure to a user process that opens the file. In either case, file access is evaluated based on the declaring process, which is the library. The library, because it is privileged, has the ability to access a file stored under a different usercode.

Example: Privileged Transparent Library Program

The concept of privileged transparent status was introduced under "Transparent Object Code File Privileges" in Section 5, "Establishing Process Identity and Privileges." For library procedures that open files, the effects of privileged transparent status vary, depending on whether the file to be opened is globally declared, locally declared, or passed as a parameter. The following ALGOL library, named FILELIB, illustrates these three possibilities.

```
$ SHARING = SHAREDBYALL
BEGIN
FILE GLOBALFILE;

PROCEDURE GLOBAL_OPEN;
  OPEN(GLOBALFILE);

PROCEDURE LOCAL_OPEN;
BEGIN
  FILE LOCALFILE;
  OPEN(LOCALFILE);
END;

PROCEDURE USER_OPEN(PASSEDFILE);
  FILE PASSEDFILE;
BEGIN
  OPEN(PASSEDFILE);
END;

EXPORT GLOBAL_OPEN, LOCAL_OPEN, USER_OPEN;
FREEZE(PERMANENT);
END.
```

FILELIB is a permanent, SHAREDBYALL library. The object code file is marked with privileged transparent status. FILELIB exports three procedures: GLOBAL_OPEN, which opens a file declared globally in the library; LOCAL_OPEN, which declares and opens a file; and USER_OPEN, which opens a file received as a parameter from the user process.

FILELIB is used by the following user program, called USERPROC.

```
BEGIN
FILE USERFILE;
LIBRARY L(LIBACCESS=BYTITLE,TITLE="OBJECT/TEST/ALGOL/LIB.");

PROCEDURE GLOBAL_OPEN;
  LIBRARY L;

PROCEDURE LOCAL_OPEN;
  LIBRARY L;

PROCEDURE USER_OPEN(PASSEDFILE);
  FILE PASSEDFILE;
  LIBRARY L;

GLOBAL_OPEN;
LOCAL_OPEN;
USER_OPEN(USERFILE);
END.
```

USERPROC invokes all of the library procedures: GLOBAL_OPEN, LOCAL_OPEN, and USER_OPEN.

Suppose that USERPROC runs with a FILEACCESSRULE value of ACTOR. If USERPROC is privileged, then it succeeds in opening all three files: LOCALFILE, GLOBALFILE, and PASSEDFILE. If USERPROC is nonprivileged, then the procedures might or might not succeed in opening the files. The success of each file open operation depends on the TITLE and SECURITYTYPE attributes of the file.

Suppose instead that the user process USERPROC has a FILEACCESSRULE value of DECLARER. In this case, USERPROC has different file access rights with regard to PASSEDFILE than it does with regard to GLOBALFILE and LOCALFILE. The rules are as follows:

- Because PASSEDFILE is ultimately declared by USERPROC, the security status of USERPROC determines whether it has the right to open PASSEDFILE. If USERPROC runs under a privileged usercode, the file open operation is executed with privileged status. Similarly, if the object code file for USERPROC is privileged, the library procedures inherit this status because they are privileged transparent.
- Because GLOBALFILE and LOCALFILE are declared in the library program, the rights to open these files are determined solely by the security status of the library process. Even if the object code file of USERPROC is privileged, the privileges inherited by GLOBAL_OPEN and LOCAL_OPEN do not extend to files declared in the library program. This behavior is a special exception to the rule that privileged transparent procedures inherit the privileged status of the code that invokes them.

There is a good reason for this strict treatment of files declared in privileged transparent libraries. The file access rights for a file declared in a library are permanently established at file-open time. Thus, a privileged user process opening a file in a shared library can have the effect of granting file access to other, nonprivileged user processes. The behavior under the default declarer rule prevents this file access from being granted accidentally.

You can overcome this restriction by assigning a FILEACCESSRULE value of ACTOR to the privileged user process that opens the file.

Example: Parent and Task Accessing a Guarded File

Suppose that a process with a NAME task attribute value of (SMITH)PROC1 declares a file titled (SMITH)FILEA ON DISK. The process (SMITH)PROC1 initiates a second process with a NAME of (SMITH)PROC2 and passes the file as a call-by-reference parameter. Both of these processes are nonprivileged. Suppose, further, that (SMITH)FILEA ON DISK has a SECURITYTYPE of CONTROLLED and a guard file that allows only processes named (SMITH)PROC1 to access this file.

Under the actor rule, if (SMITH)PROC1 opens the file, then both (SMITH)PROC1 and (SMITH)PROC2 are granted access to the file. However, if (SMITH)PROC2 attempts to open the file before (SMITH)PROC1 opens it, the file open operation fails. The process (SMITH)PROC2 cannot use the file until it has been opened by (SMITH)PROC1.

Under the declarer rule, both processes are granted access to the file, regardless of which one opens the file first. This is because (SMITH)PROC1, which declares the file, is allowed access rights by the guard file.

Understanding I/O Accounting

The system maintains several records of the I/O time accumulated by a process. More specifically, these are records of the time I/O devices devoted to executing I/Os for the process. This information is maintained in the ACCUMIOTIME task attribute. This information also appears in the Major Type 1, Minor Type 2 (EOJ) and Minor Type 4 (EOT) log entries.

If you are involved in writing billing systems or in evaluating the system workload, then you should be aware that the system logs all I/O time for shared logical files to the process that declared the file. The process that declared the file is not necessarily the process that is actually executing read and write statements that use the file. Consider the following ALGOL example:

```
BEGIN
FILE DATAFILE(KIND=DISK,NEWFILE=FALSE,DEPENDENTSPECS=TRUE);
TASK T;
PROCEDURE UPDATE;
BEGIN
  ARRAY LINE[0:79];
  WHILE NOT READ(DATAFILE,80,LINE) DO
  BEGIN
    REPLACE LINE BY "NEW DATA";
```

```
      WRITE(DATAFILE,80,LINE);
    END;
  END;
  CALL UPDATE [T];
  END.
```

In this example, the parent process initiates the procedure UPDATE as a synchronous task. UPDATE then reads each line of the file, modifies the data, and writes it back out to the file. Because the parent process was the declarer of DATAFILE, the ACCUMIOTIME attribute of the parent reflects all the I/O time logged by the UPDATE task.

Note that the system handles I/O accounting in a different manner for direct files. Suppose that one process declares a direct file, and that several other processes read from or write to that file using direct arrays. In this situation, the I/O time for each I/O operation is charged to the process that declares the direct array used for the I/O, rather than to the process that declares the direct file.

Understanding File Attribute Access

When multiple processes access a file through the same logical file, file attribute changes made by any one process are visible to all the other processes. This visibility holds true regardless of whether a process assigned a file attribute explicitly, or performed some other action that implicitly caused the file attribute value to change. Therefore, when you design processes to read or write the attributes of a shared file, you must be aware of the timing considerations previously discussed under "Synchronizing Access with Shared Logical Files" in this section.

For example, suppose a process performs a read operation through a shared logical file, and then immediately interrogates attributes such as STATE, LASTRECORD, or CURRENTRECORDLENGTH. These attributes do not necessarily return the values established by the preceding read statement, because another process might have executed an I/O operation in between the read statement and the file attribute interrogations performed by this process. You can prevent this problem in either of the following ways:

- By designing an I/O statement to directly return the information you need, so that the process does not have to interrogate file attributes to determine the result of the statement. For example, you could use a statement like the following:

  ```
  B := READ (PORTF [SUBFILE INX:0], 72, IOBUF)
  ```

  If this statement is used, the process can use the value stored in B instead of interrogating the STATE attribute, and can use the value stored in INX instead of interrogating the LASTSUBFILE attribute. The process could also read the current record length from field [47:20] of B, instead of interrogating the CURRENTRECORDLENGTH attribute.

- By designing your application so that no process performs an I/O operation on a file while another process is interrogating the attributes of that file.

File Sharing Examples

The following is a simple example of a library that allows multiple user processes to access the same disk file. This particular library allows processes to access a file as if it were a stack. In other words, whenever a user process writes to the file, the line pointer is incremented by one. Whenever a user process reads from the file, the line pointer is decremented by one. The PROCURE and LIBERATE statements are used to ensure that only one process accesses the file at a time.

```
$SHARING = SHAREDBYALL
  BEGIN
    FILE STK(KIND=DISK,MAXRECSIZE=12,BLOCKSIZE=1200);
    EVENT STACK_ACCESS;
    INTEGER TOP_OF_STACK;

    BOOLEAN PROCEDURE PUSH_STK(BUF);
    ARRAY BUF[0];
    BEGIN
      PROCURE(STACK_ACCESS);
      TOP_OF_STACK:= * + 1;
      PUSH_STK:= WRITE(STK[TOP_OF_STACK], 12, BUF);
      LIBERATE(STACK_ACCESS);
    END PUSH_STK;

    BOOLEAN PROCEDURE POP_STK(BUF);
    ARRAY BUF[0];
    BEGIN
      PROCURE(STACK_ACCESS);
      POP_STK:= READ(STK[TOP_OF_STACK], 12, BUF);
      TOP_OF_STACK:= * - 1;
      LIBERATE(STACK_ACCESS);
    END POP_STK;

    EXPORT PUSH_STK, POP_STK;
    OPEN (STK);
    TOP_OF_STACK:= -1;
    FREEZE(PERMANENT);
  END.
```

The following is an example of a WFL job that uses a global file equation to cause two tasks to use the same logical file. The logical file is declared in the job at lines 140-150. The global file equations occur at lines 190 and 210.

```
100 ?BEGIN JOB TEST/WFL;
110   JOBSUMMARY = SUPPRESSED;
120   DISPLAYONLYTOMCS = TRUE;
130   CLASS = 0;
140 FILE GBAL(KIND=REMOTE,NEWFILE=TRUE,TITLE="JUNK/ERRORLOG",
150           MAXRECSIZE=15,UNITS=WORDS);
160 MYSELF(STATIONNAME = #MYSELF(SOURCENAME));
170 OPEN(GBAL);
180 PROCESS RUN OBJECT/TEST/ALGOL/TASK;
```

```
190    FILE BALANCES:= GBAL;
200 PROCESS RUN OBJECT/TEST/ALGOL/TASK;
210    FILE BALANCES:= GBAL;
220 LOCK(GBAL);
230 ?END JOB
```

Note that, in the preceding WFL job, it is the colon before the equal sign on lines 190 and 210 that informs WFL that this is a global file equation. If the statement were *FILE BALANCES = GBAL*, then WFL would interpret this as meaning that the file title is GBAL.

The statement at line 160 ensures that the STATIONNAME task attribute of the job reflects the name of the originating station. This STATIONNAME value is inherited by the tasks, and determines the station where the GBAL remote file is opened.

The following is the program that is initiated twice by this WFL job.

```
BEGIN
FILE BALANCES;
PROCEDURE ERRWRITE(ERR_ARRAY,DEPOSIT,SEQ);
  EBCDIC ARRAY ERR_ARRAY[*];
  INTEGER DEPOSIT,SEQ;
BEGIN
  INTEGER CUST_BALANCE;
  MYJOB.LOCKED:= TRUE;
  READ(BALANCES[SEQ],//,ERR_ARRAY);
  CUST_BALANCE:= INTEGER(ERR_ARRAY,8) + DEPOSIT;
  REPLACE ERR_ARRAY BY CUST_BALANCE FOR 8 DIGITS;
  WRITE(BALANCES[SEQ],//,ERR_ARRAY);
  MYJOB.LOCKED:= FALSE;
END;
% The outer block statements are omitted from this example
END.
```

Because events cannot be declared in WFL, this program is designed to make use of the LOCKED task attribute to regulate access to the file. Setting LOCKED to TRUE has the same effect as procuring an event, and setting LOCKED to FALSE has the same effect as liberating an event. The program uses the MYJOB task variable because this task variable has visibility to all the tasks of the WFL job. This mechanism ensures that only one process is actively reading and writing the file at a time, though all processes continue to have the file open.

Accessing Disk Files Through Separate Logical Files

It is possible for multiple processes to use the same physical disk file at the same time without sharing the same logical file. The sharing is accomplished by using identifying file attributes to cause the logical files in each process to link to the same physical file when opened.

The following pages explain how to link separate logical files to the same physical file, and then review the methods of synchronizing access to the physical file.

Entering a File in the Directory

Multiple logical files can link to the same physical file only if the physical file has been entered into the disk directory. The concept of the disk directory is closely related to the concepts of *permanent* and *temporary* files. A permanent file appears in the disk directory, and by default is retained when it is closed. A temporary file does not appear in the disk directory, and by default is removed when it is closed.

A process can cause a file to be entered in the directory by any of the following methods, which are referred to as *directory entrance operations*:

- Opening the file with the NEWFILE file attribute set to TRUE and the PROTECTION file attribute set to SAVE or PROTECTED. Of these two values, SAVE is preferable in most cases because PROTECTED adds overhead. Opening the file creates a permanent file, which is entered immediately in the directory. The file continues to exist after the process terminates, unless the process specifies the PURGE option in the statement that closes the file.
- For a file that was opened with a PROTECTION value of TEMPORARY (the default), closing the file with either the LOCK or CRUNCH option specified in the close statement. In addition to closing the file, this action enters the file in the directory.
- Opening the file with the NEWFILE file attribute and the SENSITIVEDATA file attribute both set to TRUE. The main purpose of using the SENSITIVEDATA attribute is to protect sensitive information, but it also has the side effect of entering a file in the directory.

If two files with the same title exist on the same family, they cannot both be permanent. An option called AUTORM specifies the action to be taken if a process attempts to enter a file in the directory, and a file with same title already exists. If the AUTORM option is set, the attempt to enter the file in the directory causes one of the following actions:

- If the old file is not in use by any process, it is removed and the new file is entered in the directory.
- If the old file is still in use by another process, the old file is removed from the directory and the new file is added to the directory. The old file remains open as a temporary file. Some unexpected effects can arise from this situation. For example, suppose the new file was opened as a temporary file by process B, and the old file was originally opened as a permanent file by process A. When process B closes the new file with LOCK, the new file is changed into a permanent file, and the old file is changed into a temporary file. If process A then closes the old file with LOCK, the

old file becomes permanent again and the new file is changed back into a temporary file.

In general, the last file entered or reentered in the directory is the one that is saved permanently, regardless of whether the directory entry was caused by the file attribute values in force at file open time, or by the option used for the close operation.

If the AUTORM option is reset, and a permanent file with a particular title already exists, then a process that attempts to enter a file with the same title into the directory is suspended with a “DUP LIBRARY” RSVP message. An operator can restart the process by entering the RM (Remove) system command, which deletes the existing duplicate file.

The system treats the AUTORM option as set for a particular process if either or both of the following are true:

- The AUTORM option of the OPTION task attribute is set.
- The AUTORM operating system option is set. This option can be set or reset through the OP (Options) system command.

Matching Physical Files

Once a physical file is entered in the directory, processes can link other logical files to it by specifying appropriate values for the following file attributes:

- KIND

  Specifies the type of storage medium, such as DISK.
- TITLE

  Specifies the usercode, file name, and family.
- NEWFILE

  If set to FALSE, specifies that an existing file should be opened. Note that the process will be suspended with a “NO FILE” message if the file does not exist.
- DEPENDENTSPECS

  Causes the logical file to assume all the file attributes of the physical file. This makes it unnecessary for all processes to repeat the file attribute assignments that determine the structure of the file.

Synchronizing Access with Separate Logical Files

When processes access a single disk file through separate logical files, some additional synchronization problems arise beyond those that occur for shared logical files. These problems arise because read and write operations affect physical files indirectly, through a set of buffers associated with the logical file. The system moves information between the buffers and the physical file only when necessary, and the application typically plays no role in determining when such updates occur.

What is more, the BLOCKSIZE file attribute can cause multiple records to be read into or written from a buffer, so that changes by other processes to nearby records can be accidentally overwritten. (For more information about file blocking and file buffers, refer to the *I/O Subsystem Programming Guide*.)

These synchronization concerns arise only in cases where at least one of the processes writes to the shared physical file. If all the processes simply read from the file, then the order in which they execute their read operations makes no difference.

However, if at least one of the processes writes to the shared physical file, then you must use one of the following methods to ensure that I/O operations are properly synchronized:

- Use the BUFFERSHARING file attribute to cause multiple logical files to share the same set of buffers.
- Use the EXCLUSIVE file attribute to ensure that only one process has the physical file open at any given time.

Using the BUFFERSHARING File Attribute

You can use the BUFFERSHARING file attribute to allow separate logical files to share the same set of buffers. The effect is to make the results of each write statement immediately visible to all processes that use the same buffers, even if the physical file has not yet been updated. The following are the possible mnemonic values for this attribute:

- NONE. The logical file does not share buffers with any other logical file. This value is the default.
- SHARED. The logical file shares buffers with any other logical files that have a BUFFERSHARING value of SHARED or EXCLUSIVELYSHARED. Logical files with a BUFFERSHARING value of NONE can link to the same physical file, but use separate buffers.
- EXCLUSIVELYSHARED. The logical file shares buffers with any other logical files that link to the same physical file. Logical files with a BUFFERSHARING value of NONE are prevented from linking to the same physical file.

Note that several restrictions apply to the use of the BUFFERSHARING file attribute. For example, the physical file must reside on the local host and the BLOCKSTRUCTURE attribute must be FIXED. For a detailed list of the restrictions, refer to the *File Attributes Programming Reference Manual*.

If you use the SHARED or EXCLUSIVELYSHARED value for all the processes that share the same physical file, then the issue of when the physical file is updated becomes irrelevant. However, you must still handle the types of synchronization problems that were previously described under “Synchronizing Access with Shared Logical Files” in this section.

You can handle these synchronization problems in any of the following ways:

- Through the use of event variables, using the techniques described under "Synchronizing Access with Shared Logical Files," earlier in this section.
- By setting the APPEND file attribute to TRUE. The primary effect of this setting is to prevent random writes to a file and to cause all serial writes to begin at the current end-of-file position. However, when BUFFERSHARING has a value of SHARED or EXCLUSIVELYSHARED, an APPEND value of TRUE has the additional effect of causing the system to provide implicit locking for write statements. This implicit locking prevents write statements issued by different processes from conflicting and overwriting each other. Note that this locking is effective only if all the processes that write to the file use a BUFFERSHARING value of SHARED or EXCLUSIVELYSHARED and an APPEND value of TRUE.
- Through the use of a feature called *record locking.* This record locking feature makes it possible for a process to secure exclusive access to one or more file records while the process reads and updates those records. The record locking feature can be used only for files that have BUFFERSHARING = SHARED or EXCLUSIVELYSHARED or that use direct I/O.

Record locking is supported in COBOL85 through the LOCKRECORD and UNLOCKRECORD statements. Record locking is available in other languages through calls on the MCPSUPPORT library procedures RECORDLOCKER, DIRECTRECORDLOCKER, RECORDLOCKTEST, and DIRECTRECORDLOCKTEST.

Record locks are of two types: *shared locks* and *exclusive locks*.

Before performing a read operation, each process should typically secure a shared lock on the file records that are to be read. A shared lock allows other processes to establish shared locks that overlap the same region, but prevents other processes from establishing exclusive locks that overlap that region. After performing the read operation, the process should then remove the shared lock from the region.

Before performing a write operation, each process should typically secure an exclusive lock on the file records that are to be written. An exclusive lock prevents other processes from establishing any shared locks or exclusive locks that overlap the same region. After performing the write operation, the process should then remove the exclusive lock from the region.

Note that it is possible for processes to read or write a record without first locking the record, regardless of the value of the BUFFERSHARING attribute. Locking a record only prevents another process from locking the same record, not from performing I/O on that record. Record locking works well as long as all participating processes follow the convention of locking records before reading or writing them.

If a process locks a record and then terminates before unlocking the record, the system implicitly unlocks the record.

The following paragraphs briefly introduce the MCPSUPPORT procedures and COBOL85 statements that support record locking, and then explain how record locking applies to direct I/O files.

Record Locking with MCPSUPPORT Procedures

A program can lock records by calling the MCPSUPPORT procedure RECORDLOCKER. This procedure includes parameters that allow you to specify

- Whether a shared lock or exclusive lock is to be used.
- The starting position and length of the locked region.
- For regions that already have a conflicting lock, whether the RECORDLOCKER procedure should fail or wait (and if it waits, for how long).

A program can interrogate the availability of a range of records for locking by invoking the procedure RECORDLOCKTEST.

For direct files, the program must use the procedures DIRECTRECORDLOCKER instead of RECORDLOCKER, and DIRECTRECORDLOCKTEST instead of RECORDLOCKTEST.

For detailed descriptions of each of these procedures, refer to the *Master Control Program (MCP) System Interfaces Programming Reference Manual.*

Record Locking with COBOL85 Statements

A COBOL85 program can lock a record with the LOCKRECORD <file name> statement. This statement creates exclusive locks only (never shared locks). The record to be locked is not specified by the LOCKRECORD statement, but rather by the ACTUAL KEY phrase for the file. The statement can optionally include an ON EXCEPTION clause and a NOT ON EXCEPTION clause (specifying actions to be taken if the lock fails or succeeds, respectively). For a file named THETA-DATA and subroutines named BAD-LOCK and GOOD-LOCK, you could use a statement of the following form:

```
LOCKRECORD THETA-DATA
  ON EXCEPTION PERFORM BAD-LOCK
  NOT ON EXCEPTION PERFORM GOOD-LOCK.
```

If a conflicting lock already exists for the requested region, then the LOCKRECORD statement waits for the length of time specified by the FILELOCKTLIMIT (File Lock Time Limit) system command. If the conflicting lock is removed before the time limit expires, the LOCKRECORD statement succeeds, and the NOT ON EXCEPTION clause is invoked (if there is one). If the time limit expires, the LOCKRECORD statement fails, and the ON EXCEPTION clause is invoked (if there is one).

You can unlock a record using the UNLOCKRECORD statement, which has syntax similar to that of the LOCKRECORD statement. The UNLOCKRECORD statement can also include ON EXCEPTION or NOT ON EXCEPTION clauses.

If the compiler control option MUSTLOCK is TRUE and the FD statement for a file assigns a BUFFERSHARING value of SHARED or EXCLUSIVELYSHARED, then the COBOL85 compiler generates extra code for any WRITE statements that write to that file. This extra code checks to see whether the program has previously locked the record being written to. If not, then the program incurs an error. If the program has specified a file status data item, a USE procedure, or an INVALID KEY clause, then the error is nonfatal and control returns to the program. Otherwise the error is fatal and the program is discontinued.

Note that this special feature of the WRITE statement is enabled only if the FD statement includes the BUFFERSHARING assignment. If the BUFFERSHARING value is altered by a CHANGE ATTRIBUTE statement in the program, the behavior of the WRITE statement is not affected by the change.

For detailed explanations of these COBOL85 features, refer to the *COBOL ANSI-85 Programming Reference Manual, Volume 1: Basic Implementation.*

Record Locking for Direct I/O Files

Direct I/O is a technique that allows a program to create and control a private buffer for file I/O. The program creates a buffer by declaring a direct array. The application program is responsible for initiating and monitoring the transfer of information between the physical file and the direct array.

The BUFFERSHARING file attribute must have a value of NONE for direct I/O files. If multiple programs access the same physical disk file through direct I/O, each direct array exists independently. Changes made to the contents of one direct array are not propagated to other direct arrays.

Further, changes made through direct I/O have no effect on the file buffers of processes that access the file through normal I/O. This problem exists even if the processes using normal I/O have set BUFFERSHARING = SHARED.

Note: *If multiple processes update the same disk file through different logical files, then either all the processes should use direct I/O or all the processes should use normal I/O. It is not possible to reliably coordinate the updating of buffers if a file is accessed by both direct I/O and normal I/O.*

The system does support the use of record locking with direct I/O files, by way of the same MCPSUPPORT procedures mentioned previously. You can use this record locking feature effectively for cases where multiple processes all use direct I/O to update the same physical disk file. However, you must take measures to ensure that the contents of any direct arrays used to access a file are kept properly updated.

For example, suppose a certain record in a file stores the current balance in a customer's account. Suppose also that you want a program to use direct I/O to increase the account balance by $10. In order to ensure accurate results, you should design the program to follow these steps:

1. Secure an exclusive lock on the record.
2. Read the latest contents of the record into the direct array. This is necessary because another process might have updated the record since it was last read by this process.
3. Wait for the read operation to complete.
4. Update the data in the direct array to reflect the $10 increase.
5. Write the updated data from the direct array to the physical file. This write is necessary to make the results of the action visible to other processes that do not share this direct array.

6. Wait for the write operation to complete.
7. Remove the exclusive lock from the record.

Using the EXCLUSIVE File Attribute

If a file is used by multiple processes, but the processes access the file only occasionally, then you may find it convenient to simply ensure that only one of those processes has the file open at any given time. When each process closes the file, the system updates the physical file with any outstanding changes that are stored in the file buffers. These changes are therefore visible to the next process that opens the file.

The simplest method of ensuring exclusive access to a physical file is to create the file with the default PROTECTION value, which is TEMPORARY. The file is not entered in the directory and therefore is not visible to other processes. Later, the process can close the file with LOCK, thus entering the file in the directory and making it available to other processes. If another process attempts to access the file before it is locked, the process is suspended with a "NO FILE" condition. When the file is locked, the process resumes.

Another method of securing exclusive access to a file is by setting the EXCLUSIVE file attribute to TRUE before opening the file. The EXCLUSIVE file attribute specifies that no other process can have the physical file open at the same time as this process.

If a process sets EXCLUSIVE to TRUE and then opens a file, then any other process that attempts to open that physical file is suspended until this process closes it. If a process sets EXCLUSIVE and attempts to open a physical file that is already in use by another process, the process is suspended until the other process closes the file. In either case, the RSVP message displayed is "WAITING ON: <file title>."

It is possible for multiple processes to be waiting to open the same physical file with EXCLUSIVE = TRUE. When the file becomes available, one of the waiting processes opens the file and the other processes continue to wait. It is not possible to predict which of the waiting processes will succeed in opening the file first.

If it is not desirable for the program to be suspended until the file becomes available, the process can attempt a conditional open operation instead. This can be achieved by using an open statement with the AVAILABLE option set or by interrogating the AVAILABLE file attribute. If another process is currently using the file with EXCLUSIVE = TRUE, the conditional open operation fails and returns a result reporting the reason for the failure. (The results are documented in the AVAILABLE file attribute description in the *File Attributes Programming Reference Manual*.) The process then continues executing normally.

Exclusive files are best suited to situations where a single body of information is to be transmitted from one process to another. An extended dialogue between processes cannot be implemented efficiently by this method, because it requires repeated file open and close operations. Each file open or close operation is an expensive operation that consumes many times the resources required to access an event or perform a simple read or write operation.

Avoiding Nonpreferred Methods

If you do not use the BUFFERSHARING or EXCLUSIVE file attributes, then any number of logical files with separate buffers can be linked to the same physical disk file at the same time. However, you should be aware that this type of disk file sharing involves complexities of synchronization that are extremely difficult to resolve. When different buffers are used it is not possible to predict the order in which read and write operations submitted by different processes will be executed.

Some programmers have mistakenly believed that each SEEK statement to record number -1 causes the physical file to be updated with the contents of the file buffers. Although this statement sometimes has the desired effect, it is not and never has been a reliable method of flushing the buffers.

A newer feature is the SYNCHRONIZE file attribute. When SYNCHRONIZE is set to OUT, the system updates the physical file before completing any given WRITE statement. This technique can help reduce the data loss caused by a program failure or system halt/load. However, you should be aware that SYNCHRONIZE does not provide any buffer updating for READ statements.

For example, suppose that two separate processes access a physical file through separate buffers with SYNCHRONIZE set to OUT. The first process issues a WRITE statement, during which the physical file is updated. The first process then uses an event variable to signal the second process that it can read the file. The second process issues a READ statement, but the read statement may reflect outdated contents of the file buffers rather than the current contents of the physical file.

Because of these and other problems, the concurrent use of physical disk files through separate buffers is not recommended.

Synchronizing Access on Shared Disk Families

You can use the SHARE (Shared Family) system command to enable multiple hosts to share the same disk family. (A host is either an A Series system or the MCP environment of a ClearPath system.) One of the hosts is designated as the master host, and only programs on the master host can create new files, expand existing files, or modify file attributes of files on the share family. For a full description of the SHARE command and its effects, refer to the *System Commands Operations Reference Manual*.

The shared families feature can be a convenient way of making code files available for execution by multiple hosts or for making files available for reading by programs on multiple hosts.

The shared families feature also allows existing rows of a file to be rewritten by programs on any of the sharing hosts. However, such updating by programs on multiple hosts can result in timing issues that go beyond those previously discussed in this section. For instance:

- The methods of synchronization discussed under “Using Shared Logical Files” apply only to processes that are running on the same host.
- Most of the synchronization methods discussed under “Synchronizing Access with Separate Logical Files” depend on the use of the BUFFERSHARING file attribute or the EXCLUSIVE file attribute. However, programs are not permitted to set either of these attributes for files on shared families.
- It is possible to use direct I/O files on shared families, together with record locking through the MCPSUPPORT procedure RECORDLOCKER. However, the record locking affects only processes on a single host. It does nothing to prevent concurrent access by processes running on different hosts.

Given these limitations, it is generally best to avoid application architectures in which processes running on separate hosts are responsible for updating the same file on a shared family during the same period.

Section 20
Using Core-to-Core (CRCR) and Storage Queue (STOQ)

Core-to-core (CRCR) and storage queue (STOQ) are two methods of transmitting EBCDIC text between processes. The primary difference between these two methods is that CRCR communicates data synchronously, while STOQ communicates data asynchronously. Another difference is that CRCR data is always received in the order that it is sent, while STOQ data can optionally be received in a different order.

COBOL85 programs can use CRCR or STOQ by using special forms of the SEND and RECEIVE statements in that language. Programs written in other languages can use CRCR or STOQ through calls on procedures in the EVASUPPORT library.

This section emphasizes the use of CRCR through EVASUPPORT procedure calls, but also gives examples of the corresponding COBOL85 statements. For complete details on the COBOL85 syntax, refer to the *COBOL ANSI-85 Programming Reference Manual, Volume 1: Basic Implementation*.

CRCR and STOQ are modeled on features that were originally implemented for V Series systems. For information about migrating applications from V Series systems, refer to the *EVA Application Programs Transition Guide.*

This section first describes the CRCR and STOQ features separately. This section then describes procedure result values and operations interfaces that apply to both CRCR and STOQ.

Using CRCR

Core-to-core (CRCR) is a method for synchronously communicating EBCDIC text between two processes. CRCR transfers information directly from a variable in one process to a variable in another process, without using any intermediate storage area. The data transfer occurs only when both processes acknowledge that they are prepared for the exchange.

The processes communicating through CRCR must be located on the same host system.

How CRCR Processes Interlock

In order for information to be transferred through CRCR, the sending process and the receiving process must *interlock*. This interlocking is a temporary linkage established by the operating system. The system interlocks two processes if one is attempting a send operation, the other is attempting a receive operation, and the program name of each process matches that requested by the other process.

Once the interlock occurs, the system transfers the data from one process to the other and then removes the interlock. If the same two processes exchange information repeatedly, the system repeats the steps of matching the processes and establishing an interlock each time.

In practice, the two processes are unlikely to issue the send and receive requests at exactly the same moment. Therefore, the first of the two processes to submit its request enters a waiting state until the matching process submits the matching request. However, you can control which process does the waiting by making one of the two processes do a conditional send or receive request.

If you do not wish for a CRCR process to enter a waiting state, you can design the process to set the No Wait flag in the send or receive request. When you set this flag, the process abandons the send or receive operation and continues executing if the matching process is not ready to interlock.

Alternatively, you can design a CRCR process to execute a WAIT statement that waits on a list of events. The list of events can include the predeclared events CRCR_INPUT and CRCR_OUTPUT, which notify the process when a matching process submits a send or receive request. By looping repeatedly on such a complex wait statement, a process can handle both CRCR communications and a variety of other transactions as they occur.

The No Wait flag or the complex wait strategy should be used for only one of the two matching CRCR processes. One of the two CRCR processes must submit a simple send or receive request and enter a waiting state, or no interlock can ever occur.

Using CRCR Through Library Calls

Programs in languages other than COBOL85 can perform CRCR send and receive operations by calling the CRCR_SEND and CRCR_RECV procedures in the EVASUPPORT system library. These procedures each receive four parameters, which must be declared in the following order: program name, program name length, message data, and No Wait flag. These procedures also return a real value indicating the procedure result.

Program Name (EBCDIC Array)

Stores the name of the matching CRCR program. The value must be a file title of up to 256 characters in length. It is not necessary to terminate the value with a period.

If the program name does not include a usercode, the system prefixes the program name with the same usercode as the process executing the statement.

The system compares the program name with the NAME task attribute of possible matching processes. Any ON <family> clause on either side is ignored in the comparison. Otherwise, the program name and the NAME task attribute must be an exact match in order for interlock to occur.

There is one exception to the requirement for exact matches. In a CRCR_RECV invocation, the program name specified can be all blank characters. This type of invocation is referred to as a *global receive operation.* In such a case, the process interlocks with any other process that is attempting a send operation.

Note that the reverse capability, a global send operation, does not exist. If the program name for a send operation is all blanks, the process waits indefinitely and the send operation is never completed.

Program Name Length (Integer)

The user program must use this parameter to specify the length of the program name, in units of characters.

Message Data (EBCDIC Array)

Stores or receives the message data.

The variable declared in the sending process can be of a different type than the variable declared in the receiving process. However, the system does not perform any data translation. The data received is a bit image of the data that was sent.

The size of the variable is limited only by the amount of memory required by the sending and receiving processes. If the receiving variable cannot hold all the data that is sent, the system truncates the data and sends what can fit. If the receiving variable is larger than the sending variable, the system left-justifies the data and adds blank fill to the right.

No Wait Flag (Real)

A real variable that indicates whether the user program is willing to wait for the interlock to occur.

An interlock can be delayed if the matching process is not currently running or has not executed its send or receive operation yet. By default, the user program becomes suspended if the matching process is not ready, and resumes executing when the interlock occurs.

However, by setting bit [00:01] of the No Wait flag, the user program can indicate that the send or receive operation should be abandoned if the matching process is not ready to interlock. The user program continues running and can attempt the send or receive operation again later.

Procedure Result (Real)

CRCR_SEND and CRCR_RECV each return a real value indicating the result of the requested operation. The format of this value is described under “Handling the CRCR or STOQ Result” later in this section.

Using CRCR in ALGOL

The following is an example of an ALGOL program that invokes the CRCR_RECV and CRCR_SEND procedures.

```
BEGIN
EBCDIC ARRAY CRCR_PGM_NAME [0:255];
INTEGER CRCR_PGM_NAME_LEN;
EBCDIC ARRAY CRCR_DATA     [0:19999];
REAL         CRCR_FLAGS, RESULT_STATUS;

LIBRARY EVASUPPORT (LIBACCESS    =  BYFUNCTION,
                     FUNCTIONNAME = "EVASUPPORT.");
     REAL PROCEDURE CRCR_RECV (CRCR_PGM_NAME,
                                    CRCR_PGM_NAME_LEN,
                                    CRCR_DATA,
                                    CRCR_FLAGS);
        EBCDIC  ARRAY CRCR_PGM_NAME [*];
        INTEGER       CRCR_PGM_NAME_LEN;
        EBCDIC  ARRAY CRCR_DATA [0];
        REAL          CRCR_FLAGS;
        LIBRARY  EVASUPPORT;

   REAL PROCEDURE CRCR_SEND (CRCR_PGM_NAME,
                                    CRCR_PGM_NAME_LEN,
                                    CRCR_DATA,
                                    CRCR_FLAGS);
        EBCDIC  ARRAY CRCR_PGM_NAME [*];
        INTEGER       CRCR_PGM_NAME_LEN;
        EBCDIC  ARRAY CRCR_DATA [0];
        REAL          CRCR_FLAGS;
        LIBRARY  EVASUPPORT;

REPLACE CRCR_DATA[0] BY "DATA MESSAGE";
REPLACE CRCR_PGM_NAME[0] BY "(USER)PROGRAM/NAME";
CRCR_PGM_NAME_LEN:= 18;
CRCR_FLAGS:= 0;   % WAIT FOR HOOKUP

CRCR_SEND(CRCR_PGM_NAME, CRCR_PGM_NAME_LEN,
                CRCR_DATA, CRCR_FLAGS);

REPLACE CRCR_DATA[0] BY " " FOR SIZE(CRCR_DATA);
REPLACE CRCR_PGM_NAME[0] BY "(USER)PROGRAM/NAME";
CRCR_PGM_NAME_LEN:= 18;
CRCR_FLAGS:= 0;   % WAIT FOR HOOKUP

CRCR_RECV(CRCR_PGM_NAME, CRCR_PGM_NAME_LEN,
                CRCR_DATA, CRCR_FLAGS);

END.
```

Using CRCR in COBOL74

The following rules apply to the data items passed as parameters to CRCR_SEND or CRCR_RECV in COBOL74:

- The program name length parameter must be a level 77 item of type PIC S9(11) BINARY.
- The message data parameter must be of either level 77 or level 01. If a level 01 item is used, all subordinate items are passed as part of the message data.
- The flag parameter and the result status must be level 77 items of type REAL.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  RESULT-STATUS         REAL.
77  CRCR-PGM-NAME-LEN     PIC S9(11) BINARY.
77 CRCR-FLAGS            REAL.
01  CRCR-PGM-NAME         PIC X(256) WITH LOWER-BOUNDS.
01  CRCR-DATA             PIC X(20000).

PROCEDURE DIVISION.
BEGIN-PARA.
    CHANGE ATTRIBUTE LIBACCESS OF "EVASUPPORT"
      TO BYFUNCTION.
    MOVE "(EVAENG)OBJECT/TEST/CRCR/ALGOL ON MISC."
      TO CRCR-PGM-NAME.
    MOVE 39 TO CRCR-PGM-NAME-LEN.
    CALL "CRCR_RECV OF EVASUPPORT"
      USING CRCR-PGM-NAME, CRCR-PGM-NAME-LEN,
             CRCR-DATA, CRCR-FLAGS
      GIVING RESULT-STATUS.
    CALL "CRCR_SEND OF EVASUPPORT"
      USING CRCR-PGM-NAME, CRCR-PGM-NAME-LEN,
           CRCR-DATA, CRCR-FLAGS
      GIVING RESULT-STATUS.
STOP RUN.
```

Using CRCR Through COBOL85 Statements

In COBOL85, the SEND statement follows one of the following formats:

```
SEND <program name> FROM <variable name>

SEND <program name> FROM <variable name> ON EXCEPTION <statement>

SEND <program name> FROM <variable name> NOT ON EXCEPTION <statement>
```

Similarly, the RECEIVE statement follows one of the following two formats:

```
RECEIVE <program name> INTO <variable name>

RECEIVE <program name> INTO <variable name> ON EXCEPTION <statement>

RECEIVE <program name> INTO <variable name> NOT ON EXCEPTION <statement>
```

The following are explanations of the various portions of these statements:

- The program name stores the name of the matching program. The program name must be an alphanumeric literal or a variable, such as a DISPLAY item.
- The variable name refers to a variable that stores or receives the message data. The variable specified must be an alphanumeric data item, such as a DISPLAY item.
- The ON EXCEPTION clause specifies an action to be taken if the matching process is not ready to participate in the CRCR send or receive operation. If there is no ON EXCEPTION clause, the process waits until the matching process is ready. Using the ON EXCEPTION clause is equivalent to setting the No Wait flag in a CRCR_SEND or CRCR_RECV procedure invocation.
- The NOT ON EXCEPTION clause specifies an action to be taken if the CRCR statement finishes successfully.

For further information about the contents of program names and variable names, refer to the discussion of the program name and message data parameters under "Using CRCR Through Library Calls" earlier in this section.

Following is a COBOL85 program that uses CRCR constructs.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
 01  CRCR-PGM-NAME          PIC X(256).
 01  CRCR-DATA              PIC X(2000).
PROCEDURE DIVISION.
BEGIN-PARA.
   MOVE "(JASMITH)OBJECT/TEST/CRCR/ALGOL ON MISC."
     TO CRCR-PGM-NAME.
   MOVE "TEST DATA   TEST DATA   TEST DATA  "
     TO CRCR-DATA.
   SEND CRCR-PGM-NAME FROM CRCR-DATA.

   MOVE "(JASMITH)OBJECT/TEST/CRCR/ALGOL ON MISC."
     TO CRCR-PGM-NAME.
   RECEIVE CRCR-PGM-NAME INTO CRCR-DATA.
STOP RUN.
```

Using STOQ

Storage queues (STOQs) are a medium that processes can use to store information in memory for later retrieval by other processes. Any number of processes can share the same STOQ. A single process could also use a STOQ for temporary storage.

Each message sent to a STOQ simply remains there until a process signals its readiness to receive that message. Further, a single STOQ can store multiple messages. These features make it possible for STOQ applications to run in parallel without having to synchronize the sending or receiving of messages.

The applications that use STOQs can specify whether each message should be sent to, or received from, the top or bottom of a STOQ. This flexibility makes it simple for you to implement last in, first out (LIFO) or first in, first out (FIFO) communications mechanisms.

STOQ applications can also subdivide each STOQ into subqueues with distinct names. Applications can then send messages to, or receive messages from, the top or bottom of each subqueue.

An inquiry mechanism provides a rapid means of determining the number of messages in a STOQ or a subqueue without disturbing any elements.

The programs communicating through STOQs must reside on the same host system.

The following pages present an overview of STOQ communications and describe how to perform STOQ communications through the library interface or through COBOL85 statements.

Creating and Using STOQs

To use STOQ communications, a program must first declare a STOQ parameter block. A STOQ parameter block serves as a buffer for the data in a single send operation or receive operation. A program can reuse the same STOQ parameter block for successive operations, provided that the program reinitializes the data in the parameter block appropriately each time. The same STOQ parameter block can be reused even for operations that involve different STOQs.

It is not necessary for a program to explicitly create or link to a STOQ. Before each send or receive operation, the program simply stores the STOQ name in the appropriate portion of the STOQ parameter block. If the STOQ specified by a send operation does not yet exist, the system creates the STOQ at that point. Any process can thereafter access the same STOQ by specifying the same STOQ name for a send or receive operation.

The STOQ grows one message longer after each send operation. Each receive operation removes a message from the STOQ and decreases the length of the STOQ by one.

A STOQ can continue to exist even after the process or processes that sent messages to it have terminated. The messages placed in the STOQ by those processes remain in the STOQ until received by another process or processes. The STOQ itself continues to exist until the next halt/load, even if it no longer contains any messages and is not in use by any process.

An operator can use the WQ (Display STOQ Count) command in MARC to display the number of messages in a STOQ. An operator can also purge messages from a STOQ with the RQ (Remove STOQ Entries) command in MARC. For details, refer to “Operator Interfaces to CRCR and STOQ” later in this section.

While a STOQ can store multiple messages, memory constraints limit the possible size of a STOQ. If a process attempts to send a message to a STOQ that is full, the process is then suspended unless it has set the No Wait flag for the send operation.

Similarly, if the STOQ or subqueue specified for a receive operation is empty, the receiving process is suspended unless it has set the No Wait flag for the receive operation.

A program can use the events STOQ_INPUT or STOQ_OUTPUT to detect when a STOQ is ready for a send or receive operation. For details, refer to “Waiting for CRCR or STOQ Events” later in this section.

A receive operation can specify a partial subqueue name. Such an operation returns the first message whose subqueue name begins with that partial subqueue name. For example, a receive operation that specifies a subqueue named OPS could receive a message from subqueues named OPS, OPSCON, OPSREP, or OPSPOD.

Using STOQ Through Library Calls

Application programs can perform STOQ communications by invoking three procedures in the EVASUPPORT system library: STOQ_SEND, STOQ_RECV, and STOQ_POLL. Before invoking these procedures, each application program must declare and initialize a STOQ parameter block.

Declaring a STOQ Parameter Block

A STOQ parameter block stores the following information, in the following order:

- The name of the STOQ. The STOQ name is always six characters long, and should be left-justified and blank filled if necessary to attain the six-character length. The STOQ name can contain any EBCDIC characters, including embedded blanks and nonprinting values. The name cannot consist entirely of null characters.
- The length of the subqueue name, in characters. This value must be in the range 0 to 99. A length of 0 indicates that there is no subqueue name.
- The subqueue name, if any. The subqueue name can contain any EBCDIC characters, including embedded blanks and nonprinting values. The name cannot consist entirely of null characters.

- The length of the message data, in characters. This value must be in the range 0 to 9999. A length of 0 indicates that there is no message data.
  - For send operations, this value must be stored in advance by the application program.
  - For receive operations, the caller stores a value that determines the maximum message size to be received. After a receive, the system sets the value to the actual size of the message received. If this actual message size value is larger than the maximum message size value stored by the caller, the message will be truncated to the caller’s maximum message size. The caller should restore the maximum message size value prior to each receive.
  - For poll operations, this value is returned by the system and reflects the number of messages in the specified STOQ or subqueue.
- The message data. The maximum length for message data is 9999 bytes. The message data can contain any EBCDIC characters, including embedded blanks and nonprinting values.
  - For send operations, this value must be stored in advance by the application program.
  - For receive operations, this value is returned by the system.
  - For poll operations, this field is not used.

Send, Receive, and Poll Operations

The three EVASUPPORT procedures STOQ_SEND, STOQ_RECV, and STOQ_POLL each receive two parameters: a STOQ parameter block and a flag variable. The STOQ parameter block was described in the previous subsection.

The flag variable is a real variable with the following significant fields:

Field	Value and Meaning
[01:01]	Indicates whether the top or bottom of the STOQ is used for the operation. • 0 The bottom of the STOQ or subqueue is used. • 1 The top of the STOQ or subqueue is used.
[00:01]	The No Wait flag. Indicates whether to wait or to abandon the STOQ operation if a delay is encountered. A delay occurs for a STOQ_SEND if the specified STOQ is full. A delay occurs for a STOQ_RECV if there are no messages in the specified STOQ or subqueue. • 0 The process waits until the send or receive operation can be completed. • 1 The process abandons the operation and proceeds to the next statement.

Additionally, the STOQ_SEND, STOQ_RECV, and STOQ_POLL procedures each return a real value. The application program can inspect this result to determine whether errors occurred in the operation. Refer to "Handling the CRCR or STOQ Result" later in this section.

Using STOQ in ALGOL

For ALGOL programs, the STOQ parameter block can be declared as an EBCDIC array. In this array,

- The first six bytes store the STOQ name.
- The next byte stores the two-digit subqueue name length in packed decimal form.
- The next 0 to 99 bytes store the subqueue name (depending on the value of the subqueue name length specified in the previous byte).
- The next two bytes store the four-digit message data length in packed decimal form.
- The next 0 to 9999 bytes store the message data.

The following ALGOL program includes statements to define a STOQ parameter block and invokes the STOQ_SEND, STOQ_RECV, and STOQ_POLL procedures.

```
BEGIN

EBCDIC ARRAY STOQ_BLOCK [0:9999];

REAL STOQ_FLAGS, RESULT_STATUS;

LIBRARY EVASUPPORT (LIBACCESS     =  BYFUNCTION,
                     FUNCTIONNAME = "EVASUPPORT.");
     REAL PROCEDURE STOQ_RECV (STOQ_BLOCK,  STOQ_FLAGS);
           EBCDIC ARRAY STOQ_BLOCK [0];
           REAL         STOQ_FLAGS;
           LIBRARY EVASUPPORT;

   REAL PROCEDURE STOQ_SEND (STOQ_BLOCK,  STOQ_FLAGS);
           EBCDIC ARRAY STOQ_BLOCK [0];
           REAL         STOQ_FLAGS;
           LIBRARY EVASUPPORT;

   REAL PROCEDURE STOQ_POLL (STOQ_BLOCK,  STOQ_FLAGS);
           EBCDIC ARRAY STOQ_BLOCK [0];
           REAL         STOQ_FLAGS;
           LIBRARY EVASUPPORT;

REPLACE STOQ_BLOCK[0] BY "QUEUE1", % QUEUE NAME
                          48"09",   % SUB QUEUE NAME LEN
                          "SUBQUEUE1", % SUB QUEUE NAME
                          48"0012", % DATA LENGTH
                          "DATA MESSAGE";  % DATA
STOQ_FLAGS.[1:1]:= 0;   % INSERT AT BOTTOM OF STOQ
STOQ_FLAGS.[0:1]:= 0;   % WAIT FOR SPACE
STOQ_SEND(STOQ_BLOCK, STOQ_FLAGS);

REPLACE STOQ_BLOCK[0] BY 48"00" FOR 10000;
REPLACE STOQ_BLOCK[0] BY "QUEUE1", % QUEUE NAME
                         48"09",   % SUB QUEUE NAME LEN
                         "SUBQUEUE1", % SUB QUEUE NAME
                         48"0000"; % CLEAR FOR RECV
STOQ_FLAGS.[1:1]:= 1;   % RECEIVE FROM TOP OF STOQ
STOQ_FLAGS.[0:1]:= 0;   % WAIT FOR MESSAGE
STOQ_RECV(STOQ_BLOCK, STOQ_FLAGS);

REPLACE STOQ_BLOCK[0] BY 48"00" FOR 10000;
REPLACE STOQ_BLOCK[0] BY "QUEUE1", % QUEUE NAME
                         48"00",   % SUB QUEUE NAME LEN
                         48"0000"; % CLEAR FOR POLL
STOQ_FLAGS:= 0;   % CLEAR - NOT  USED FOR POLL
STOQ_POLL(STOQ_BLOCK, STOQ_FLAGS);

END.
```

Using STOQ in COBOL74

For COBOL74 programs, the STOQ parameter block must be a group item. All subordinate items are passed to STOQ.

All subqueue names used with that STOQ parameter block must be the same length as the subordinate item declared to store the subqueue name. One simple way to handle this requirement is by padding all subqueue names with blanks to bring them to the same length. An alternative would be to declare a different STOQ parameter block for use with each subqueue name of a given length.

The subqueue name item should be declared with the number of digits specified by the subqueue name length item. Thus in the following example, STOQ-SUBQ-LENGTH is declared with a value of 6, and STOQ-SUBQ is declared as PIC X(6).

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  RESULT-STATUS          REAL.
77  STOQ-STATUS            REAL.
01  STOQ-BLOCK.
    03  STOQ-QUEUE         PIC X(6).
    03  STOQ-SUBQ-LENGTH   PIC 9(2) COMP VALUE 6.
    03  STOQ-SUBQ          PIC X(6).
    03  STOQ-DATA-LENGTH   PIC 9(4) COMP.
    03  STOQ-DATA          PIC X(9999).

PROCEDURE DIVISION.
BEGIN-PARA.
```

```
          MOVE "QUEUE1" TO STOQ-QUEUE.
          MOVE "SUBQ01" TO STOQ-SUBQ.
          MOVE "HELLO THERE" TO STOQ-DATA.
          MOVE 11 TO STOQ-DATA-LENGTH.
          CHANGE ATTRIBUTE LIBACCESS OF "EVASUPPORT"
            TO BYFUNCTION.

        CALL "STOQ_SEND OF EVASUPPORT"
            USING STOQ-BLOCK, STOQ-STATUS
            GIVING RESULT-STATUS.
          CALL "STOQ_RECV OF EVASUPPORT"
            USING STOQ-BLOCK, STOQ-STATUS
            GIVING RESULT-STATUS.
          CALL "STOQ_POLL OF EVASUPPORT"
            USING STOQ-BLOCK, STOQ-STATUS
            GIVING RESULT-STATUS.
          STOP RUN.
```

Using STOQ Through COBOL85 Statements

COBOL85 supports STOQ communications through various language extensions. It is not necessary for a COBOL85 program to import procedures from the EVASUPPORT library.

STOQ parameter blocks are declared in COBOL85 in the same way as in COBOL74. The STOQ parameter block must be a group item. All subordinate items are passed as part of the STOQ parameter block.

All subqueue names used with that STOQ parameter block must be the same length as the subordinate item declared to store the subqueue name. One simple way to handle this requirement is by padding all subqueue names with blanks to bring them to the same length. For example, all subqueue names used with the following STOQ parameter block should be padded with blanks to bring them to 10 characters in length:

```
01  STOQ-BLOCK.
    03  STOQ-QUEUE         PIC X(6).
    03  STOQ-SUBQ-LENGTH   PIC 9(2) COMP VALUE 10.
    03  STOQ-SUBQ          PIC X(10).
    03  STOQ-DATA-LENGTH   PIC 9(4) COMP.
    03  STOQ-DATA          PIC X(9999).
```

The subqueue name item should be declared with the number of digits specified by the subqueue name length item. Thus in the preceding example, STOQ-SUBQ-LENGTH is declared with a value of 10, and STOQ-SUBQ is declared as PIC X(10).

In COBOL85, the SEND statement for a STOQ follows one of the following two formats:

```
SEND TO <top or bottom> <STOQ parameter block>.

SEND TO <top or bottom> <STOQ parameter block> ON EXCEPTION <statement>.
```

In COBOL85, the RECEIVE statement for a STOQ follows one of the following two formats:

```
RECEIVE FROM <top or bottom> <STOQ parameter block>.

RECEIVE FROM <top or bottom> <STOQ parameter block>
   ON EXCEPTION <statement>.

RECEIVE FROM <top or bottom> <STOQ parameter block>
   NOT ON EXCEPTION <statement>.
```

In the preceding statements,

- The <top or bottom> clause consists of the keyword TOP or BOTTOM and specifies the end of the STOQ or subqueue at which the message should be inserted.
- The <STOQ parameter block> clause is the identifier of a STOQ parameter block declared in the program.
- The ON EXCEPTION clause specifies a statement to be executed if the SEND or RECEIVE statement cannot be completed immediately. A SEND statement incurs a delay if the specified STOQ is full. A RECEIVE statement incurs a delay if there are no messages in the specified STOQ or subqueue.

  If there is no ON EXCEPTION clause and a delay occurs, the process waits until the SEND or RECEIVE statement can be completed.
- The NOT ON EXCEPTION clause specifies an action to be taken if the SEND or RECEIVE statement finishes successfully.

To poll the message count in a STOQ, the syntax is:

```
ACCEPT <STOQ parameter block> MESSAGE COUNT.
```

The following COBOL85 program declares a STOQ parameter block and uses the SEND, RECEIVE, and ACCEPT statements.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  STOQ-BLOCK.
    03  STOQ-QUEUE         PIC X(6).
    03  STOQ-SUBQ-LENGTH   PIC 9(2) COMP.
    03  STOQ-SUBQ          PIC X(6).
    03  STOQ-DATA-LENGTH   PIC 9(4) COMP.
    03  STOQ-DATA          PIC X(9999).

PROCEDURE DIVISION.
BEGIN-PARA.
    MOVE "QUEUE1" TO STOQ-QUEUE.
    MOVE 6 TO STOQ-SUBQ-LENGTH.
    MOVE "SUBQ01" TO STOQ-SUBQ.
    MOVE 11 TO STOQ-DATA-LENGTH.
```

```
        MOVE "HELLO THERE" TO STOQ-DATA.
      SEND TO TOP STOQ-BLOCK.
        ACCEPT STOQ-BLOCK MESSAGE COUNT.
        DISPLAY STOQ-DATA-LENGTH.
        RECEIVE FROM TOP STOQ-BLOCK ON EXCEPTION GO ERR-HANDLER.
        STOP RUN.

    ERR-HANDLER.
        DISPLAY "ERROR OCCURRED".
        STOP RUN.
```

Waiting On CRCR or STOQ Events

Programs written in ALGOL or COBOL85 can use predeclared variables to detect events related to CRCR or STOQ communications. These predeclared variables can be used only in WAIT and WAITANDRESET statements. These predeclared variables are not available in other languages.

The following are the ALGOL names of these predeclared events and explanations of when the system causes them. The COBOL85 names for these events are the same except that hyphens (–) are substituted for the underscores (_).

CRCR_INPUT <program name>	Caused when the specified program is ready to send CRCR data to this program.
CRCR_OUTPUT <program name>	Caused when the specified program is ready to receive CRCR data from this program.
STOQ_INPUT <STOQ parameter block>	Caused when a message appears in a specified STOQ.
STOQ_OUTPUT <STOQ parameter block>	Caused when storage space is available to send a new message in a specified STOQ.

The following paragraphs present examples of ALGOL and COBOL85 statements that use these events.

CRCR and STOQ Events in ALGOL

The following ALGOL program fragment demonstrates the use of CRCR and STOQ events in a WAIT statement:

```
REPLACE PROGNAME BY "OBJECT/EPSILON";
EVORDER:= 1;
DONE:= FALSE;
WHILE NOT DONE DO
BEGIN
   EVT:= WAITANDRESET (CRCR_INPUT "OBJECT/DELTA",
                       CRCR_OUTPUT PROGNAME,
                       STOQ_INPUT STOQ_ALPHA,
                       STOQ_OUTPUT STOQ_BETA,
                       MYSELF.ACCEPTEVENT) [EVORDER];
```

```
      CASE EVT OF
      BEGIN
         1: %... Call the CRCR_RECV procedure
         2: %... Call the CRCR_SEND procedure
         3: %... Call the STOQ_RECV procedure
         4: %... Call the STOQ_SEND procedure
         5: DONE:= TRUE;
      END;
      EVORDER:= * + 1;
      IF EVORDER > 4 THEN
         EVORDER:= 1;
   END;
```

In this example, the WAIT statement waits on several different events: CRCR_INPUT, CRCR_OUTPUT, STOQ_INPUT, STOQ_OUTPUT, and MYSELF.ACCEPTEVENT.

The STOQ_INPUT and STOQ_OUTPUT events in the WAIT statement refer to the STOQ parameter blocks STOQALPHA and STOQBETA, declared previously in the program. The statements that initialize STOQALPHA and STOQBETA are omitted from the example.

The EVORDER variable specified in the WAIT statement indicates the first event to be tested for a HAPPENED state. Because the EVORDER value is changed after each WAIT statement, the WAIT statement tests a different event first each time it is executed. This technique prevents starvation problems, as discussed in “Preventing Starvation Problems” in Section 16, “Using Events and Interlocks.”

CRCR and STOQ Events in COBOL85

The following COBOL85 program fragment demonstrates the use of CRCR and STOQ events in a WAIT statement. This example performs the same functions as the preceding ALGOL example.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
 01  STOQALPHA.
     03  STOQ-QUEUE         PIC X(6).
     03  STOQ-SUBQ-LENGTH   PIC 9(2) COMP VALUE 6.
     03  STOQ-SUBQ          PIC X(6).
     03  STOQ-DATA-LENGTH   PIC 9(4) COMP.
     03  STOQ-DATA          PIC X(9999).
 01  STOQBETA.
     03  STOQ-QUEUE         PIC X(6).
     03  STOQ-SUBQ-LENGTH   PIC 9(2) COMP VALUE 6.
     03  STOQ-SUBQ          PIC X(6).
     03  STOQ-DATA-LENGTH   PIC 9(4) COMP.
     03  STOQ-DATA          PIC X(9999).
 77 DONE      BINARY PIC 9(11).
 77 PROGNAME  PIC X(256).
 77 EVT       REAL.
 77 EVORDER   BINARY PIC 9(11).
 77 DELTA-INPUT     PIC X(2000).
 77 PROGNAME-OUTPUT PIC X(2000).

PROCEDURE DIVISION.
WAITLOOP SECTION.
P1.
* The statements to initialize the STOQ blocks STOQALPHA & STOQBETA
* are omitted from this example
MOVE 0 TO DONE.
WAIT UNTIL CRCR-INPUT "OBJECT/DELTA"
           CRCR-OUTPUT PROGNAME
           STOQ-INPUT STOQALPHA
           STOQ-OUTPUT STOQBETA
           ATTRIBUTE ACCEPTEVENT OF MYSELF
     USING EVORDER
     GIVING EVT.

IF EVT = 1
   RECEIVE "OBJECT/DELTA" INTO DELTA-INPUT
ELSE IF EVT = 2 THEN
   SEND PROGNAME FROM PROGNAME-OUTPUT
ELSE IF EVT = 3 THEN
   RECEIVE FROM TOP STOQALPHA
ELSE IF EVT = 4 THEN
   SEND TO TOP STOQBETA
ELSE
   MOVE 1 TO DONE
END-IF.

COMPUTE EVORDER = EVORDER + 1.
IF EVORDER IS > 1
   MOVE 1 TO EVORDER
END-IF.
IF DONE IS = 0 THEN
   GO WAITLOOP.
STOP RUN.
```

Handling the CRCR or STOQ Result

The EVASUPPORT library procedures used for CRCR or STOQ communications each return a real value that follows the same format. CRCR and STOQ applications can use this value to determine whether each requested action succeeded or failed.

The following are the procedures that return a value in this format:

- CRCR_RECV
- CRCR_SEND
- STOQ_POLL
- STOQ_RECV
- STOQ_SEND

This procedure result is divided into the following fields:

Field	**Value and Meaning**
[47:32]	This field is reserved for future use.
[15:04]	If an invalid parameter was detected, this field indicates which parameter was in error. A value of 1 indicates the first parameter; a value of 2 indicates the second parameter, and so on.
[11:01]	This field is used only if field [0:1] stores a 1, meaning that the function was not successful. • 0 The failure is nonfatal. • 1 The failure is fatal. In this case, the system terminates the user program before the user program reads the procedure result. However, you can read the procedure result value in program dumps to determine why the fault occurred.
[10:01]	Type of operation. • 0 STOQ operation. • 1 CRCR operation.

Field	Value and Meaning
[09:06]	This field indicates the specific error that occurred. To interpret the meaning of this field, the calling program must first interrogate field [11:01] to determine whether the error was fatal, and field [10:01] to determine whether the error was related to STOQ or CRCR. For nonfatal CRCR errors, the following are the possible values and their meanings: • 1 No matching partner was available for the operation. • 2 Insufficient memory available for CRCR data. • 3 Maximum number of CRCR users reached. For nonfatal STOQ errors, the following are the possible values and their meanings: • 1 No receive queue entry found for STOQ. • 2 No memory available for STOQ data. • 3 No queue slots available for STOQ data. For fatal CRCR errors, the following are the possible values and their meanings: • 1 CRCR tables corrupted. • 2 Invalid CRCR parameter. Field [15:04] specifies which parameter incurred the error. • 3 No usercode. • 4 The program attempted a global send operation (that is, a CRCR_SEND operation with the program name parameter left blank). The global form is valid only for RECEIVE operations. For fatal STOQ errors, the following are the possible values and their meanings: • 1 The STOQ tables are corrupted. • 2 An invalid parameter was passed to a STOQ procedure. Field [15:04] specifies which parameter incurred the error.

Field	**Value and Meaning**
[03:03]	This field is reserved for future use.
[00:01]	• 0 The function was successful. • 1 The function was not successful.

If bits [11:08] are interrogated as a single field, then the possible values fall into the following ranges:

1 to 63	Nonfatal STOQ error
64 to 127	Nonfatal CRCR error
128 to 191	Fatal STOQ error
192 to 255	Fatal CRCR error

Result Handling in ALGOL

The following is an example of how an ALGOL program could detect errors by using the result returned by CRCR and STOQ functions.

```
EBCDIC ARRAY CRCR_PGM_NAME [0:255];
INTEGER CRCR_PGM_NAME_LEN;
EBCDIC ARRAY CRCR_DATA     [0:19999];
REAL          CRCR_FLAGS, RESULT_STATUS;

LIBRARY EVASUPPORT (LIBACCESS    =  BYFUNCTION,
                     FUNCTIONNAME = "EVASUPPORT.");
     REAL PROCEDURE CRCR_RECV (CRCR_PGM_NAME,
                                    CRCR_PGM_NAME_LEN,
                                    CRCR_DATA,
                                    CRCR_FLAGS);
        EBCDIC  ARRAY CRCR_PGM_NAME [*];
        INTEGER  CRCR_PGM_NAME_LEN;
        EBCDIC  ARRAY CRCR_DATA     [0];
        REAL          CRCR_FLAGS;
        LIBRARY  EVASUPPORT;

REPLACE CRCR_DATA[0] BY "DATA MESSAGE";
REPLACE CRCR_PGM_NAME[0] BY "(USER)PROGRAM/NAME";
CRCR_PGM_NAME_LEN:= 18;
CRCR_FLAGS:= 0;   % WAIT FOR HOOKUP

IF BOOLEAN(RESULT_STATUS:=
     CRCR_RECV(CRCR_PGM_NAME, CRCR_PGM_NAME_LEN,
                CRCR_DATA, CRCR_FLAGS))
           THEN
     IF BOOLEAN(RESULT_STATUS.[10:1]) THEN
           DISPLAY("CRCR ERROR")
                ELSE
           DISPLAY("STOQ ERROR");
```

Because fatal errors cause the program to be terminated, the program does not need to check whether the error returned is fatal or nonfatal.

Result Handling in COBOL74

The following is an example of how a COBOL74 program could detect errors by using the result returned by the CRCR and STOQ functions.

```
WORKING-STORAGE SECTION.
  77  RESULT-STATUS         REAL.
  77  CRCR-PGM-NAME-LEN     PIC S9(11) BINARY.
  77  CRCR-FLAGS            REAL.
  01  CRCR-PGM-NAME         PIC X(256) WITH LOWER-BOUNDS.
  01  CRCR-DATA             PIC X(20000).
  77 ERRBOOL               REAL.
  77  ERRTYPE               REAL.
PROCEDURE DIVISION.
BEGIN-PARA.
     CHANGE ATTRIBUTE LIBACCESS OF  "EVASUPPORT"
           TO BYFUNCTION.
     CALL "CRCR_RECV OF EVASUPPORT"
           USING CRCR-PGM-NAME, CRCR-PGM-NAME-LEN,
                 CRCR-DATA, CRCR-FLAGS
           GIVING RESULT-STATUS.
     MOVE RESULT-STATUS TO ERRBOOL [00:00:01].
     MOVE RESULT-STATUS TO ERRTYPE [10:00:01].
     IF ERRBOOL = 1
           IF ERRTYPE = 1
                GO TO CRCR-ERROR
                     ELSE
                GO TO STOQ-ERROR.
```

In the preceding example, CRCR-ERROR and STOQ-ERROR are names of error-handling sections located elsewhere in the program.

Because fatal errors cause the program to be terminated, the program does not need to check whether the error returned is fatal or nonfatal.

Operations Interfaces to CRCR and STOQ

The following MARC commands allow an operator to monitor or intervene in CRCR and STOQ communications:

- RQ (Remove STOQ Entries)
- WQ (Display STOQ Count)
- WY (Display Process Status)

Note: *The RQ, WQ, and WY commands described here can be used only in MARC. Unlike system commands, they cannot be entered at an ODT or submitted through the DCKEYIN function.*

User programs can also invoke these commands by calling procedures in the EVASUPPORT library. No special privileges are needed in order to call these procedures.

The following pages explain how to use these commands, either through MARC or through library calls.

Clearing a STOQ

You can clear the messages from STOQs by using the RQ command or the STOQ_RQ library procedure.

Using the RQ (Remove STOQ Entries) Command in MARC

The RQ command has the following forms:

Form	Function
RQ <queue name>	Removes all messages from the STOQ with the specified queue name.
RQ =	Removes all messages from all STOQs currently in memory.

The following are an example RQ command and the response to that command:

```
RQ MSGQA

   MSGQA REMOVED
```

Using the STOQ_RQ Library Procedure

The following are the ALGOL declarations for the STOQ_RQ library procedure:

```
LIBRARY EVASUPPORT (LIBACCESS = BYFUNCTION,
                    FUNCTIONNAME = "EVASUPPORT.");

PROCEDURE STOQ_RQ(DATA_ARRAY);
     EBCDIC ARRAY DATA_ARRAY[0];
     LIBRARY EVASUPPORT;
```

The parameter DATA_ARRAY serves the following two purposes:

- The user program initializes this array with the STOQ name or an equal sign (=). The STOQ name must be six characters long, padded with blanks if necessary. Similarly, the equal sign must be left-justified and padded with five blanks.
- The system returns the response to the STOQ_RQ action in this same array.

The response returned in DATA_ARRAY consists of one or more entries that have the following format:

Length	Content
6 bytes	The STOQ name
3 bytes	The number of messages that were in the queue. This is a six-digit number stored in packed decimal form.
3 bytes	The number of user programs that are waiting on this STOQ. This is a six-digit number stored in packed decimal form.

The final entry is followed by a null character (48"00").

Polling a STOQ

You can display the number of messages in STOQs by using the WQ command or the STOQ_WQ library procedure.

Using the WQ (Display STOQ Count) Command in MARC

The WQ command has the following forms:

Form	Function
WQ <queue name>	Displays the message count for the STOQ with the specified queue name.
WQ =	Displays the message counts for all STOQs currently in memory.

The following is an example WQ command:

```
WQ MSGQA
```

The following response indicates that there are seven messages waiting in MSGQA:

```
MSGQA 7
```

The following response indicates that there no messages waiting in MSGQA, but a program has issued a receive request for that STOQ and is waiting for a message to arrive.

```
MSGQA 0  (WTG)
```

Using the STOQ_WQ Library Procedure

The following are the ALGOL declarations for the STOQ_WQ library procedure:

```
LIBRARY EVASUPPORT (LIBACCESS = BYFUNCTION,
                    FUNCTIONNAME = "EVASUPPORT.");

PROCEDURE STOQ_WQ(WQDATA);
     EBCDIC ARRAY WQDATA[0];
     LIBRARY EVASUPPORT;
```

The parameter WQDATA serves the following two purposes:

- The user program initializes this array with the STOQ name or an equal sign (=).
- The system returns the response to the STOQ_WQ action in this same array.

The parameter WQDATA serves two purposes. The user program should initialize this array with a six-byte queue name or an equal sign (=). The system also returns the response to the STOQ_WQ action in this array. The response consists of one or more entries that have the following format:

Length	**Content**
6 bytes	The STOQ name
3 bytes	The number of messages that were in the queue. This is a six-digit number stored in packed decimal form.
3 bytes	The number of user programs that are waiting on this STOQ. This is a six-digit number stored in packed decimal form.
1 byte	For the process associated with the following mix number, specifies the STOQ operation the process is waiting on. A value of 0 indicates a send operation, and a value of 1 indicates a receive operation.
5 bytes	The mix number of a process waiting on this STOQ.

The last two fields are repeated for each process that is waiting on this STOQ. Then the overall format is repeated for the next STOQ, if any. The final entry is followed by a null character (48"00").

Displaying Status of CRCR or STOQ Operations

You can display the number of messages in STOQs by using the WY command or the STOQ_WY library procedure.

Using the WY (Display Process Status) Command in MARC

To display the CRCR or STOQ communications status of a particular process, enter a command of the following form in MARC:

```
WY <mix number>
```

The WY command displays an extra line called the IPC State for any process that meets one of the following criteria:

- The process is using a WAIT statement to wait on one of the events CRCR_INPUT, CRCR_OUTPUT, STOQ_INPUT, or STOQ_OUTPUT.
- The process attempted a send or receive operation using CRCR or STOQ, and is waiting for the operation to complete.

The following is an example of the WY command output for a process waiting on a receive operation or the STOQ_INPUT event relative to the STOQ named QUEUE1:

```
Status of Task 9069\9109 AT 13:56:45
Priority: 50
Origination: MP021/OCJFM_1/CANDE/2/CANDE/1  (LSN 1018)
MCS: SYSTEM/CANDE
Usercode: DELTA1
Stack State: Waiting on an event
Program name: (CPB)OBJECT/UTP/STOQ/RECV/TOPQ/WAIT ON EVAMCP
IPC State: WTG STOQ RECV QUEUE1
```

If the process is using a WAIT statement to wait on multiple events, at least one of which is a CRCR or STOQ event, then the IPC Status line displays *WTG COMPLEX WAIT* and is followed by lines listing the individual CRCR or STOQ events. The following is an example of the IPC portion of this display:

```
IPC State: -WTG COMPLEX WAIT-
WTG STOQ RECV QUEONE
WTG CRCR RECV (DELTA1)OBJECT/CWT/SEND/ONE
WTG CRCR SEND (DELTA1)OBJECT/CWT/RECV/TWO
```

If you enter the WY command without a mix number, the output displays abbreviated information about all the user processes on the system. CRCR and STOQ status information is displayed for those processes waiting on CRCR or STOQ events. In the following output example, processes 9113 and 8996 are waiting on CRCR and STOQ events:

```
Response returned at 20:10:42

 JOB# MIX# Pr (UC)Name/ Status
 ==== ==== == =========================================================
 7179/6903 99 (OPCON)*SYSTEM/DELTA/REFIT
 8699/8715 80 *CANDE/STACK02
 6954/6954 80 *SYSTEM/CANDE
 9110/9113 50 (DELTA1)(CPB)OBJECT/CWT/WAIT ON EVAMCP
              WTG COMPLEX WAIT
                  WTG STOQ RECV QUEONE
                  WTG CRCR RECV (EVAENG)OBJECT/CWT/SEND/ONE
 6955/6981 80 *SYSTEM/COMS
 6955/6978 80 COMS/INPUT
 6955/6979 80 COMS/TANK
 8962/8996 50 (DELTA1)(CPB)OBJECT/UTP/CRCR/RECV/WAIT ON EVAMCP
                  WTG CRCR RECV (EVAENG)OBJECT/UTP/CRCR/SEND/WAIT
```

The MARC *WY* command should not be confused with the Y (Status Interrogate) system command. The differences are as follows:

- At an ODT, the commands *Y*, *WY*, and *WHY* each invoke the Y (Status Interrogate) system command. This command does not give any specific information about the use of CRCR or STOQ by a process.
- In MARC, the commands *Y*, *WY*, and *WHY* each invoke the MARC *WY* command. This command displays IPC Status information if the process is waiting on CRCR or STOQ communications.

Using the EVA_WY Library Procedure

The following are the ALGOL declarations for the EVA_WY library procedure:

```
LIBRARY EVASUPPORT (LIBACCESS = BYFUNCTION,
                    FUNCTIONNAME = "EVASUPPORT.");

PROCEDURE EVA_WY(MIX_NUM, RESULT_ARRAY);
     VALUE MIX_NUM;
     INTEGER MIX_NUM;
     ARRAY RESULT_ARRAY[0];
     LIBRARY EVASUPPORT;
```

The following are explanations of the parameters:

MIX_NUM

The user program uses this parameter to specify the process that is being interrogated.

RESULT_ARRAY

The system uses this parameter to return information to the user about the specified process or processes. The following format is repeated for each process that is reported on:

Word	Field	Value and Meaning
0	47:02	Type of action • 1 Core-to-core • 2 Storage queue • 3 Complex wait. This value indicates that the process is waiting on one or more of the following events: CRCR_INPUT, CRCR_OUTPUT, STOQ_INPUT, or STOQ_OUTPUT.
	45:01	Send or receive indication • 0 Send for CRCR or STOQ • 1 Receive for CRCR or STOQ
	15:16	Mix number of the process
1		Length in bytes of program name for CRCR or queue Name for STOQ. For complex wait entries, this word is unused.
2 through N		Program name for CRCR or queue name for STOQ. For complex wait entries, this word is unused. N is the number of the last word in this entry. The value of N depends on the program name length specified in Word 1. If the program name does not end at a word boundary, the remainder of word N is unused. The next entry, if any, starts at word N + 1.

Section 21
Using ONC+ Remote Procedure Call (RPC)

Remote Procedure Call (RPC) enables processes to execute procedures on remote hosts over a network. ONC+ RPC is one of a family of distributed services known as the Open Network Computing Plus (ONC+) architecture. ONC+ has been widely accepted as a standard for distributed computing.

ONC+ RPC applications are divided into a server program, which provides procedures, and a client program, which invokes procedures in the server program. ONC+ RPC server programs serve a purpose similar to server library programs, in that both types of programs provide procedures for use by other programs. However, server libraries and their clients must reside on the same host, whereas ONC+ RPC applications can be distributed across multiple hosts in a multivendor environment.

ONC+ RPC for ClearPath MCP programs can operate as a client, as a server, or as both a client and a server.

ONC+ RPC isolates a program completely from the network transport, meaning that no changes to the application are needed if new transports are added to a system. Currently, ONC+ RPC for ClearPath MCP supports the following network transports provided by Transmission Control Protocol/Internet Protocol (TCP/IP), release level 32.0 or later:

- Transmission Control Protocol (TCP)
- User Datagram Protocol (UDP)

You can use the NETPATH task attribute to specify which of these network transports is to be used for a particular application. The possible values of this attribute are "TCP" and "UDP".

ONC+ RPC applications on A Series systems can be developed in the C and ALGOL programming languages. You can develop C language ONC+ RPC applications in either of the following ways:

- By using the RPCGEN stub compiler. First, you use the RPC language to describe the interfaces between the client and server. You then use the RPCGEN stub compiler to process the RPC code. RPCGEN creates several output files, including user-modifiable client and server skeleton files. These files contain calls on *portal routines*, which provide the application programming interface for ONC+ RPC. You can then modify these files and write the remainder of the application in C.
- By using direct calls on portal routines. You can bypass the stub compiler by coding calls on the portal routines yourself. You write these calls in C, rather than in RPC language.

You can develop ALGOL language ONC+ RPC applications by making calls on ONCRPCSUPPORT library procedures. The file SYMBOL/ONC/RPC/INTERFACE contains code to make development easier. You can use the $INCLUDE compiler option to include the contents of this file in your program.

For detailed information about how to implement ONC+ RPC applications, refer to the *ONC+ Remote Procedure Call (RPC) for MCP/AS Installation and Programming Guide.*

Appendix A
Related Product Information

The following documents provide information that is directly related to the primary subject of this publication.

***EVA Application Programs Transition Guide* (3957 6145)**

This guide describes those areas of the V Series application programs that must be modified or filtered to obtain equivalent source code. This guide is intended for V Series system programmers at sites where transition to an enterprise server system is anticipated.

***MCP/AS ALGOL and MCP Interfaces to POSIX Features Programming Reference Manual* (7011 8351)**

This manual describes how to access POSIX features in programs that are not written in C language. Two basic interface methods are described: ALGOL include file functions and client library procedures that import objects from the MCPSUPPORT library. The ALGOL include file defines a subset of POSIX functions that can be used only in ALGOL programs. Client library procedures can be used by any programming language that supports the use of libraries. This manual is written for systems programmers.

***MCP/AS Binder Programming Reference Manual* (8600 0304)**

This manual describes the functions and applications of the Binder, an efficiency tool that reduces the need to recompile an entire program when only a portion of the program has been modified. This manual is written for programmers who are familiar with programming language concepts and terms.

***MCP/AS C Programming Reference Manual, Volume 1: Basic Implementation* (8600 2268)**

This manual describes the C programming language. It includes descriptions of syntax, status messages, the preprocessor, compiling system, binding system, and run-time library. Extensions such as compiler control options and the library facility are also documented. This manual is written for systems and applications programmers.

***MCP/AS C Programming Reference Manual, Volume 2: Headers and Functions* (8600 2278)**

This manual describes the C headers in detail, and the functions, macros, and types defined in those headers. This manual is written for systems and applications programmers.

MCP/AS CANDE Operations Reference Manual **(8600 1500)**

This manual describes how CANDE operates to allow generalized file preparation and updating in an interactive, terminal-oriented environment. This manual is written for a wide range of computer users who work with text and program files.

MCP/AS FORTRAN 77 Programming Reference Manual **(3957 6053)**

This manual describes the implementation of FORTRAN77 (the designation for the American National Standard Programming Language FORTRAN, ANSI X3.9-1978.) It describes the various options, control statements, and extensions used with this compiler. This manual is written for programmers.

MCP/AS Menu-Assisted Resource Control (MARC) Operations Guide **(8600 0403)**

This guide provides an overview of MARC, a description of the menu structure, and information on how to use help text, commands, security features, and Communications Management System (COMS) windows from MARC. The guide also explains how to run programs from MARC, how to customize MARC to meet user needs, and how to use MARC in a multinational environment. This guide is written for a wide audience, ranging from experienced system administrators to end users with no previous knowledge of MARC.

MCP/AS Pascal Programming Reference Manual, Volume 1: Basic Implementation **(8600 0080)**

This manual describes the basic features of the Pascal language. This manual is written for programmers who are familiar with programming concepts.

MCP/AS POSIX User's Guide **(7011 8328)**

This guide describes the basic concepts of the POSIX interface, including process control and file management. It also describes specifically how the POSIX.1 interface is implemented and used on the enterprise server. This guide is written for programmers and any user who wants to understand the POSIX interface.

MCP/AS Report Program Generator (RPG) Programming Reference Manual, Volume 1: Basic Implementation **(8600 0544)**

This manual describes the basic features of the RPG programming language, including both RPG I and RPG II dialects. This manual is written for application programmers who are familiar with programming concepts.

MCP/AS Security Features Operations and Programming Guide **(8600 0528)**

This guide describes the security features available to users and provides instructions for their use. This guide is written for users who are responsible for maintaining the security of their individual programs and data.

MCP/AS System Administration Guide **(8600 0437)**

This guide provides the reader with information required to make decisions about system configuration, peripheral configuration, file management, resource use, and other matters related to system administration. This guide is written for users with some, little, or no experience who are responsible for making decisions about system administration.

***ONC+ Remote Procedure Call (RPC) for MCP/AS Installation and Programming Guide* (8600 2383)**

This guide describes the facilities for implementing distributed applications. All utilities, options, and library functions in this guide reflect ONC+ Release 1.1.

***Unisys e-@ction Application Development Solutions ALGOL Programming Reference Manual, Volume 1: Basic Implementation* (8600 0098)**

This manual describes the basic features of the Extended ALGOL programming language. This manual is written for the applications programmer or systems analyst who is experienced in developing, maintaining, and reading ALGOL programs.

***Unisys e-@ction Application Development Solutions COBOL ANSI-74 Programming Reference Manual, Volume 1: Basic Implementation* (8600 0296)**

This manual describes the basic features of the standard COBOL ANSI-74 programming language, which is fully compatible with the American National Standard, X3.23-1974. This manual is written for programmers who are familiar with programming concepts.

***Unisys e-@ction Application Development Solutions COBOL ANSI-85 Programming Reference Manual, Volume 1: Basic Implementation* (8600 1518)**

This manual describes the basic features of the COBOL ANSI-85 programming language. This manual is written for programmers who are familiar with programming concepts.

***Unisys e-@ction Application Development Solutions DCALGOL Programming Reference Manual* (8600 0841)**

This manual describes the Data Communications ALGOL (DCALGOL) language. This language is designed to support the implementation of message control systems (MCSs) and other resource monitoring and controlling programs that require access to special operating system interfaces. This manual is written for systems programmers.

***Unisys e-@ction ClearPath Enterprise Servers Distributed Systems Services Operations Guide* (8600 0122)**

This guide describes the capabilities and features of distributed systems services and how to use them. It is intended for system operators, system administrators, and general computer users.

***Unisys e-@ction ClearPath Enterprise Servers File Attributes Programming Reference Manual* (8600 0064)**

This manual contains information about each file attribute and each direct I/O buffer attribute. The manual is written for programmers and operations personnel who need to understand the functionality of a given attribute. The *I/O Subsystem Programming Guide* is a companion manual.

***Unisys e-@ction ClearPath Enterprise Servers I/O Subsystem Programming Guide* (8600 0056)**

This guide contains information about how to program for various types of peripheral files and how to program for interprocess communication, using port files. This guide is written for programmers who need to understand how to describe the characteristics of a file in a program. The *File Attributes Programming Reference Manual* is a companion manual.

***Unisys e-@ction ClearPath Enterprise Servers MCP System Interfaces Programming Reference Manual* (8600 2029)**

This manual describes selected library objects exported from the MCPSUPPORT library, and describes the ARCHIVESUPPORT, BILLINGSUPPORT, and TAPEMANAGER libraries. This manual is written for system programmers who want to write programs that interface with the system software.

***Unisys e-@ction ClearPath Enterprise Servers MultiLingual System Administration, Operations, and Programming Guide* (8600 0288)**

This guide describes how to use the MLS environment, which encompasses many products. The MLS environment includes a collection of operating system features, productivity tools, utilities, and compiler extensions. The guide explains how these products are used to create application systems tailored to meet the needs of users in a multilingual or multicultural business environment. It explains, for example, the procedures for translating system and application output messages, help text, and user interface screens from one natural language to one or more other languages; for instance, from English to French and Spanish. This guide is written for international vendors, branch systems personnel, system managers, programmers, and customers who wish to create customized application systems.

***Unisys e-@ction ClearPath Enterprise Servers Print System and Remote Print System Administration, Operations, and Programming Guide* (8600 1039)**

This guide describes the features of the Print System and provides a complete description of its command syntax. This guide is written for programmers, operators, system administrators, and other interactive users of Menu-Assisted Resource Control (MARC) and CANDE.

***Unisys e-@ction ClearPath Enterprise Servers Security Administration Guide* (8600 0973)**

This guide describes system-level security features and suggests how to use them. It provides administrators with the information necessary to set and implement effective security policy. This guide is written for system administrators, security administrators, and those responsible for establishing and implementing security policy.

***Unisys e-@ction ClearPath Enterprise Servers System Commands Operations Reference Manual* (8600 0395)**

This manual gives a complete description of the system commands used to control system resources and work flow. This manual is written for systems operators and administrators.

***Unisys e-@ction ClearPath Enterprise Servers System Log Programming Reference Manual* (8600 1807)**

This manual describes the format and contents of all the Major Type and Minor Type entries of the system log. It also contains information about controlling the log contents and about writing log analysis programs.

***Unisys e-@ction ClearPath Enterprise Servers System Operations Guide* (8600 0387)**

This guide describes concepts and procedures required to operate most Unisys systems. Sections 1 and 2 contain information and procedures that can be done by novice operators. Section 3 contains operations and procedures that require more advanced operations experience. This guide is written for operators responsible for operating the enterprise server, especially operators with little or no experience.

***Unisys e-@ction ClearPath Enterprise Servers System Software Utilities Operations Reference Manual* (8600 0460)**

This manual provides information on the system utilities BARS, CARDLINE, CDFORMAT, COMPARE, DCAUDITOR, DCSTATUS, DUMPALL, DUMPANALYZER, FILECOPY, FILEDATA, HARDCOPY, INTERACTIVEXREF, ISTUTILITY, LOGANALYZER, LOGGER, PATCH, PCDRIVER, PRINTCOPY, RLTABLEGEN, SORT, XREFANALYZER, and the V Series conversion utilities. It also provides information on KEYEDIO support, Peripheral Test Driver (PTD), and mathematical functions. This manual is written for applications programmers, system support personnel, and operators.

***Unisys e-@ction ClearPath Enterprise Servers Task Attributes Programming Reference Manual* (8600 0502)**

This manual describes all the available task attributes. It also gives examples of statements for reading and assigning task attributes in various programming languages. The *Task Management Programming Guide* is a companion manual.

***Unisys e-@ction ClearPath Enterprise Servers Work Flow Language (WFL) Programming Reference Manual* (8600 1047)**

This manual presents the complete syntax and semantics of WFL. WFL is used to construct jobs that compile or run programs written in other languages and that perform library maintenance such as copying files. This manual is written for individuals who have some experience with programming in a block-structured language such as ALGOL and who know how to create and edit files using CANDE or the Editor.

***Unisys e-@ction Transaction Server for ClearPath MCP Programming Guide* (8600 0650)**

The guide explains how to write online, interactive, and batch application programs that run under the Transaction Server. This guide is written for experienced applications programmers with knowledge of data communication subsystems.

Index

Special Characters

86000494-506