Britton Lee Host Software

IDMLIB USER'S GUIDE

(R3v5m9)

May 1988

Part Number 205-1681-003

This edition is intended for use with Britton Lee Host Software Release 3.5 and future releases, until further notice.

The information contained within this document is subject to change without notice. Britton Lee assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under license and may only be used or copied by the terms of such license.

Table of Contents

Preface

Britton Lee's Integrated Database Management (IDM) system offers the means for sharing data among individuals who need direct access to the same information. Britton Lee systems allow dissimilar host computers to connect with a single data source.

The database resides totally within the Britton Lee hardware, so database tasks such as processing low-level database commands, maintaining data consistency, managing backup and restore operations, regulating resource sharing, scheduling processes, and monitoring performance are all handled by the Integrated Database Manager (IDM) RDBMS software running on the special purpose processor.

The IDM host-resident software performs a number of functions which involve communication with the user. A user on a host computer queries a database interactively using Britton Lee's Intelligent Database Language, IDL, or the IBM-compatible Structured Query Language, SQL.

It is also possible to query a database from a C program running on a host computer. The program can pass IDL or SQL query language statements to the Britton Lee database machine for execution. Data from the database machine can be returned to the host program for processing.

Britton Lee has developed an extensive subroutine library called IDMLIB to facilitate the efficient development of programs which access data on any Britton Lee database machine. This document is a practical guide to the use of IDMLIB.

"C Programming With IDMLIB" contains general information for anyone writing programs in C which utilize the functions provided by IDMLIB.

"The IDM Runtime System" describes a high-level programming language interface to the database machine. A programmer can use this interface to access a database on a database machine from a C program without having to learn the details of the machine's data structures and operations. This interface is of special interest to applications programmers.

"Introduction to Exception Handling" describes how the exception handling subsystem provided by IDMLIB can be used to specify actions to be taken when an exceptional condition occurs in an IDMLIB function.

"Command-Line Arguments" describes an IDMLIB function to process command-line arguments in a host-independent manner.

The appendices provide instructions for building an executable program incorporating IDMLIB in the various host environments supported by Britton Lee Host Software.

1. C Programming With IDMLIB

IDMLIB provides tools for the development of programs running on a host computer system which access data on a Britton Lee database server. All IDMLIB functions are written in C and are portable to all operating systems supported by Britton Lee Host Software.

1.1. General

Any program which uses IDMLIB must meet certain requirements, which are illustrated in the program below.

```
1  /*
2  **  SIMPLE.C -- This program prints a simple greeting.
3  */
4
5   #include <idmlib.h>
6
7   main()
8   {
9      INITIDMLIB("simple");
10     printf("Hi folks.\n");
11     exit(RS_NORM);
12  }
```

From this example one can infer the following requirements for any program which links to IDMLIB:

- The header file *idmlib.h* must be listed as the first header file when the routines in the source file use IDMLIB but not the standard I/O library. The *printf* in this program is an IDMLIB function, not a standard I/O function. If both the standard I/O library and IDMLIB are used, include *istdio.h* instead of *idmlib.h*. Other header files may be required, depending on which IDMLIB functions the program calls.
- All programs must initialize IDMLIB by calling "INITIDMLIB" with the name of the executable object as its argument. This must be the first executable statement in any program which uses IDMLIB.
- All programs must terminate with an explicit call to the IDMLIB function *exit()* with an IDMLIB return code as its argument.

1.2. Data Types

Some IDMLIB functions accept constants representing database server types as arguments. The following table shows the correspondence between these database server data types and C data types.

Database Server Type	*Length*	*C Type*
iINT1	1	char
iINT1	1	BOOL
iINT2	2	short
iINT4	4	long
iFLT4	4	float
iFLT8	8	double
iCHAR	variable	char*
iFCHAR	variable	char*
iPCHAR	variable	char*
iSTRING	variable	char*
iBCD	variable	BCDNO
iBCDFLT	variable	BCDNO
iBINARY	variable	BYTE*
iFBINARY	variable	BYTE*

The types BYTE, BOOL and BCDNO are defined in *idmlib.h*.

The database server supports BCD (binary coded decimal) integer and floating point types which correspond to the C type BCDNO. A BCDNO is defined as a **struct** containing the type, length and value of a BCD object on the database server. When a BCD value is retrieved from the database server and stored as a C object of type BCDNO in a C program, the "type" field of the C object gets the value of the type of the database server attribute. When an object of type BCDNO is sent to the database server, it is sent as the (database server) type which corresponds to the value of the "type" field in the BCDNO **struct**.

The database server cannot perform arithmetic or conversions with floating-point variables, so a program which expects the database server to perform these operations should convert the variables to type BCDNO.

1.3. Return Codes

Many IDMLIB functions return values of type RETCODE. These return codes are integer values indicating success or failure of an IDMLIB function. They are listed under RETCODE in Section 5I of the *Host Software Specification* for Unix systems and the *C Run-Time Library Reference* for other systems.

RS_NORM indicates success. Return codes with the prefix "RE_" signify a failing condition. Return codes with the prefix "RW_" are warnings. Warnings frequently indicate correct though unexpected conditions such as "end of data".

The specific definitions of these codes are machine-dependent. The macros RETSUCCESS, RETWARNING, and RETERROR provide machine-independent methods of determining how an IDMLIB function performed.

1.4. Macros

IDMLIB uses a number of macro definitions which may be unfamiliar. Some macros commonly used in IDMLIB are

Macro	*Definition*	*Meaning*
BITSET(word,bit)	(((word & (bit)) != 0	a bit is set
BYTENULL	((BYTE *) NULL)	null pointer to BYTE
CHARNULL	((char *) NULL)	null pointer to char
FUNCNULL	((FUNCP) NULL)	null pointer to function
RETSUCCESS(code)	*varies*	successful return from IDMLIB function
RETWARNING(code)	*varies*	unexpected return from IDMLIB function
RETERROR(code)	*varies*	unsuccessful return from IDMLIB function

1.5. Exceptions

In addition to returning a RETCODE, most errors in IDMLIB cause an exception to be raised. When this occurs, the exception handling subsystem normally displays a message in the standard IDMLIB format and continues execution, except in the case of severe exceptions which cause the program to abort.

A C program can catch raised exceptions before a message is displayed and specify another action to be taken by declaring an exception handler. Exception handlers are discussed under "Introduction to Exception Handling".

1.6. Reference Materials

All of the IDMLIB functions are described in Section 3 of the *Host Software Specification* for Unix systems and in the *C Run-Time Library Reference* for other systems. This reference is indispensable for writing programs which use IDMLIB.

The reference provides a formal description of each IDMLIB function which includes

- the name of the function,
- a summary of the calling format, including arguments and return values,
- a narrative description of the function,
- a description of all possible RETCODEs, if applicable,
- a list of exceptions commonly raised by the function,
- one or more examples of C code using the function,
- a list of IDMLIB functions of related interest.

2. The IDM Runtime System

This chapter describes the IDM Runtime System, which encompasses the IDMLIB functions having names beginning with "ir". A programmer can use these functions to execute database queries on the database server, without having to acquire expertise on the details of the database server's internal data structures and routines.

The prerequisites for using the IDM Runtime System are a working knowledge of the C programming language and one of two database query languages: the Intelligent Database Language (IDL) or the Structured Query Language (SQL). If you are not familiar with either of these database query languages, consult the *IDL Reference Manual* or *SQL Reference Manual* before proceeding with this section.

The examples in this guide use SQL, but IDL commands could be used instead. Familiarity with the concepts of *qualification* and *target-list* and the following database commands is required to understand the examples used to illustrate the IDM Runtime System.

IDL Command	*SQL Command*
range	*no SQL equivalent*
append to	**insert into**
begin transaction	**set autocommit off**
end transaction	**commit work**
define	**store**
execute	**start**
replace	**update**
retrieve	**select**

The examples in this chapter assume the following schema for the "title" table in the "books" database:

```
create table title
(
        docnum      smallint,
        title       char(35),
        onhand      smallint
)
```

2.1. Fundamental Routines

The following program inserts a specific row into the "title" table in the "books" database. It illustrates the most fundamental routines of the IDM Runtime System.

```
1   /*
2   **  APPEND.C -- This program inserts a row into the "title"
3   **              table in the "books" database.
4   **
5   */
6
7   #include <idmlib.h>
8   #include <idmrun.h>
9
10  main()
11  {
12     IDMRUN *idmrunptr;
13
14     /* initialize IDMLIB */
15     INITIDMLIB("append");
16
17     /* create an IDMRUN structure to access the "books" database */
18     idmrunptr = iropen("books");
19
20     /* parse and execute an SQL insert command */
21     irsql(idmrunptr, "insert into title (docnum, title, onhand)\
22           values(max(docnum) + 1, 'the great gatsby', 6)");
23     irexec(idmrunptr);
24
25     /* close the IDMRUN structure */
26     irclose(idmrunptr);
27
28     /* clean up and terminate */
29     exit(RS_NORM);
30  }
```

This program creates an IDMRUN structure, parses and executes some database commands, and removes the IDMRUN structure when all database server activity is completed. Because it uses the IDM Runtime System, it includes *idmrun.h*. The header file *idmrun.h* must be included after *idmlib.h*, and, in programs in which they are used, before the header files *env.h*, *idmdone.h*, *idmtlist.h* and/or *idmtree.h*.

The fundamental "ir" routines used by *append.c*, *iropen()*, *irclose()*, *irsql()*, and *irexec()*, are described below.

2.1.1. Create An IDMRUN Structure

The basic data structure used by the IDM Runtime System is the IDMRUN structure. An IDMRUN structure can be conceptualized as a pipe into which database commands are dropped. An IDMRUN structure can have only one parsed IDL or SQL command

string associated with it at a given time, but that command string can contain multiple commands. When a new command string is parsed, the previously parsed commands are flushed. If a command selects data from the database server, the data must be fetched and processed by the host before a new string of commands is dropped into the pipe.

A program must create at least one IDMRUN structure with *iropen()* before calling any other IDM Runtime System function. The function *iropen()* takes one argument, the name of the database to be queried. It returns a pointer to the IDMRUN structure which it has created. This pointer is the first argument passed to subsequent "ir" functions which access that IDMRUN structure.

The IDMRUN structure pointer *idmrunptr* is declared on line 12 of *append.c*. The call to *iropen()* on line 18 creates an IDMRUN structure and sets *idmrunptr* to point to it.

When the IDMRUN structure is no longer needed, it is removed with *irclose()*.

2.1.2. Parse a Command

After an IDMRUN structure has been created, IDL or SQL commands can be parsed and executed using that structure.

The function *iridl()* is called to parse IDL commands, *irsql()* to parse SQL commands. Both *iridl()* and *irsql()* take two arguments: a pointer to the IDMRUN structure and the command string to be parsed.

```
/* parse an IDL command */
iridl(idmrunptr, "retrieve (t.docnum) where t.onhand < 5");

/* parse an SQL command */
irsql(idmrunptr, "select docnum from title where onhand < 5");
```

Several database commands can be concatenated into a single command string to be passed to *iridl()* or *irsql()*. When more than one command is contained in the command string, the individual commands should be separated by blanks. Do not separate commands with the word **go** or a semicolon, as in interactive IDL or SQL.

```
#define  CMDS "insert into title \
              (docnum, title, onhand) \
              values (max(docnum), 'madame bovary', 6) \
              select docnum, title, onhand \
              where docnum = (select max(docnum) from title)"

/* parse multiple SQL commands */
irsql(idmrunptr, CMDS);
```

2.1.3. Execute a Command

After a command string has been parsed with *iridl()* or *irsql()*, it can be executed on the database server with *irexec()*. This function takes the IDMRUN pointer as a single argument.

```
irexec(idmrunptr);
```

If more than one command has been parsed and is awaiting execution, *irexec()* executes only the first command. In such cases, *irnext()* must be called to execute subsequent commands.

```
#define  CMDS  "insert into title \
               (docnum, title, onhand) \
               values (max(docnum) + 1, 'the great gatsby', 6) \
               select docnum, title, onhand \
               where docnum = (select max(docnum) from title)"

/* parse multiple SQL commands */
irsql(idmrunptr, CMDS);

/* execute the first command */
irexec(idmrunptr);

/* execute subsequent commands */
while (RETSUCCESS(irnext(idmrunptr)))
{
            /* code to process data */
}
```

Because of the way transactions are implemented in SQL, the commands **rollback work** and **commit work** are each considered as two internal commands by the database server. Therefore, the execution of a single instance of either of these commands requires both a call to *irexec()* and a call to *irnext()*.

```
/* this code parses and executes an SQL ''commit work'' command */
irsql(idmrunptr, "commit work");
irexec(idmrunptr);
while (RETSUCCESS(irnext(idmrunptr)))
       ;
```

2.2. Routines to Select Data

The following variation of the previous program not only inserts a new row into the "title" table but also selects it and displays the data at the terminal. Three variables, *docnum*, *title*, and *onhand*, are declared to receive data from the three target-list elements, *docnum*, *title*, and *onhand*, selected from the database server. The call to *irexec()* on line 29 causes the select command to be executed on the database server. The functions *irbind()* and *irfetch()* make the selected data available to the C program on the host. These functions are described in detail in Sections 2.2.1 and 2.2.2.

```
1   /*
2   **  APPRET.C --  This program inserts a row into the "title"
3   **       table in the "books" database and selects the new row.
4   */
5
6   #include <idmlib.h>
7   #include <idmrun.h>
8
9   main()
10  {
11     IDMRUN *idmrunptr;
12     short  docnum, onhand;
13     char   title[36];
14
15     /* initialize IDMLIB */
16     INITIDMLIB("appret");
17
18     /* create an IDMRUN structure */
19     idmrunptr = iropen("books");
20
21     /* parse and execute an SQL insert command */
22     irsql(idmrunptr, "insert into title(docnum, title, onhand) \
23           values(max(docnum) + 1, 'don quixote', 6)");
24     irexec(idmrunptr);
25
26     /* parse and execute an SQL select command */
27     irsql(idlrunpts, "select docnum, title, onhand from title \
28           where docnum = (select max(docnum) from title)");
29     irexec(idmrunptr);
30
31     /* bind selected data to program variables */
32     irbind(idmrunptr, 1, iINT2, 2, __ &docnum);
33     irbind(idmrunptr, 2, iSTRING, sizeof(title), __ title);
34     irbind(idmrunptr, 3, iINT2, 2, __ &onhand);
35
36     /* fetch the selected data from the database server */
37     while (RETSUCCESS(irfetch(idmrunptr)))
38           printf("%d        %s        %d\n", docnum, title, onhand);
39
40     irclose(idmrunptr);
41     exit(RS_NORM);
42  }
```

2.2.1. Bind Selected Data to Program Variables

The function *irbind()* designates a C program variable that is to receive selected data, and converts the selected data to the type of the specified variable. It takes five arguments:

(1) an IDMRUN pointer,

(2) the position of the selected data in the target-list,

(3) the type of the program variable which will be bound to the data,

(4) the length of the program variable which will be bound to the data,

(5) the address of the program variable which will be bound to the data, cast as a BYTE *.

There must be one call to *irbind()* for every program variable to be bound to a target-list element. The call to *irbind()* on line 32 of *appret.c* binds the first target-list element in the select command, *docnum* (a smallint) to the C program variable *docnum* (a short). The variable *docnum* is cast as a BYTE * using the special macro "__". Similarly, the call to *irbind()* on line 33 binds the C program variable *title* to the second target-list element, *title*, and the call on line 34 binds *onhand* to the third target-list element, *onhand*. Any selected target-list elements not bound with *irbind()* would be discarded.

2.2.2. Read Rows from the Database Server

Data from the database server is not transferred to the C program variables until the call to *irfetch()* on line 37. This function reads selected data from the database server and assigns it to the variables designated by *irbind()*, performing any necessary type conversions. It takes the IDMRUN pointer as its single argument. The function *irfetch()* is usually executed in a loop which continues until all of the selected rows have been read. The body of the loop contains the code to process the selected data.

If for any reason the *irfetch()* loop is exited before all of the selected rows have been read, the unread rows must be flushed. The function *irflush()* flushes unread rows and leaves the IDMRUN structure in the state it would be in had all of the selected rows been read.

```
while (RETSUCCESS(irfetch(idmrunptr)))
{
        if (onhand < 0)
        {
                printf("Error in database\n");
                irflush(idmrunptr);
                break;
        }
        else
                printf("%d        %s        %d\n", docnum, title, onhand);
}
```

If unread rows are not flushed, a subsequent call to any IDM Runtime System function involving that IDMRUN structure will raise an exception, because rows are still remaining to be read.

At times it may be desirable to cancel all current operations and those waiting to be processed, as well as flushing selected data. For example, if a keyboard interrupt is received from an impatient user, all outstanding operations should be canceled before the program continues. For this purpose, use *ircancel()*. This function flushes selected data and terminates all current activity on the specified IDMRUN structure, leaving it in a state to receive a new set of commands.

2.3. Other IDM Runtime System Routines

2.3.1. Substitute Constants in an SQL Command

Returning to the sample program in Section 2.1, suppose that instead of hardcoding the row to be inserted into the program, we wanted the ability to change the values of constants in the IDL or SQL command string during program execution without having to reparse the query. This can be done by using substitution constants in the command string which is passed to *iridl()* or *irsql()*. A substitution constant has the syntax of a name preceded by a percent symbol (%).

The following program gets input from the user for the values to be inserted and substitutes them in the SQL command string using *irsubst()*.

```
1   /*
2   ** APPSUB.C -- This program inserts new rows into the "title"
3   **         table in the "books" database, getting input for
4   **         the new row from the user.
5   */
6
7   #include <idmlib.h>
8   #include <idmrun.h>
9
10  main()
11  {
12      IDMRUN *idmrunptr;
13      short  onhand;
14      char   title[36], buf[36];
15
16      INITIDMLIB("appsub");
17      idmrunptr = iropen("books");
18
19      /* parse an SQL insert command with substitution constants */
20      irsql(idmrunptr, "insert into title (docnum, title, onhand) \
21              values(max(docnum) + 1, %title, %onhand)");
22
23      /* loop on user input until user signals <RETURN> */
24      for (;;)
25      {
26              /* get user input for title and onhand */
27              getprompt(buf, sizeof(buf),
28                      "Enter title or <RETURN> to quit: ");
29              if (buf[0] == '\0')
30                      break;
31              strcpy(title, buf);
32              getprompt(buf, sizeof(buf), "Enter quantity: ");
33              onhand = atos(buf);
34
35              /* substitute user values in the insert command */
36              irsubst(idmrunptr, "title", iSTRING, sizeof(title),
37                      __ title);
38              irsubst(idmrunptr, "onhand", iINT2, 2, __ &onhand);
39
40              /* execute the insert command */
41              irexec(idmrunptr);
42      }
43      irclose(idmrunptr);
44      exit(RS_NORM);
45  }
```

Note the differences between this program and the one in Section 2.1. The variables *onhand* and *title* declared on lines 13 and 14 hold the values to be substituted into the command string. The call to *irsql()* on line 20 contains the substitution constants "%title" and "%onhand" instead of hardcoded values.

After the call to *irsql()* which parses the insert command, and before the call to *irexec()* which executes it, *irsubst()* is called to specify the values which are to replace the substitution constants in the command string. There is one call to *irsubst()* for each value to be substituted. The function *irsubst()* takes five arguments:

(1) an IDMRUN pointer,

(2) the name of the substitution constant, in quotation marks, without the % symbol,

(3) the data type of the value to be substituted,

(4) the length of the value to be substituted, in bytes; if the type of the value to be substituted is ISTRING, this value is ignored and the actual length of the string is used,

(5) the address of the variable containing the value to be substituted, a BYTE *.

The call to *irsubst()* on line 36 substitutes the value of the C program variable *title* for the substitution constant "%title". The call on line 38 substitutes the value of *onhand* for "%onhand". The value is copied from the program variable, so subsequent changes to the value of that variable will not affect the value substituted by *irsubst()*.

The IDMLIB function *getprompt()* is called to get user input to be assigned to *title* and *onhand*. This function displays a prompt at the user's terminal and reads the user's response into a buffer, returning a pointer to the buffer. The function *getprompt()* provides the only reliable host-independent method of outputting a line which does not terminate with a newline to the terminal.

2.3.2. Set Up an Execute/Start Command

If the command to be executed is an IDL **execute** or SQL **start** command, use *irxcmd()* to create its tree and *irxsetp()* to supply the required parameters to the stored command. This is faster than parsing the command with *iridl()* or *irsql()*.

The function *irxcmd()* takes the IDMRUN pointer and the name of the stored command as its arguments. The function *irxsetp()* sets the values of the parameters for the stored command. There must be one call to *irxsetp()* for each parameter to the stored command, with each call passing five arguments:

(1) an IDMRUN pointer,

(2) the name of the parameter as it is defined in the stored command, in quotation marks, without the $ symbol,

(3) the data type of the parameter,

(4) the length of the parameter, in bytes; use a negative number to indicate a variable length parameter,

(5) the value of the parameter, a BYTE *.

Assume that there is a stored command on the database server named "addtitle" which is defined as follows:

```
store addtitle
        insert into title
        (docnum, title, onhand)
        values (max(docnum) + 1, &t, &q)
end store
```

We could then rewrite *append.c* as *stored1.c* invoking this stored command with *irxcmd()* and *irxsetp()*.

```
1   /*
2   **  STORED1.C -- This program inserts a row into the "title" table
3   **           in the "books" database, using the stored command "addtitle"
4   */
5
6   #include <idmlib.h>
7   #include <idmrun.h>
8
9   main()
10  {
11     IDMRUN *idmrunptr;
12     short  onhand;
13     char   *title;
14
15     INITIDMLIB("stored1");
16     idmrunptr = iropen("books");
17
18     /* assign values to pass to the stored command */
19     onhand = 6;
20     title = "the great gatsby";
21
22     /* arrange to execute an SQL start command */
23     irxcmd(idmrunptr, "addtitle");
24
25     /* supply parameters to the stored command */
26     irxsetp(idmrunptr, "q", iINT2, 2, __ &onhand);
27     irxsetp(idmrunptr, "t", iSTRING, -1, __ title);
28
29     /* execute the stored command */
30     irexec(idmrunptr);
31
32     irclose(idmrunptr);
33     exit(RS_NORM);
34  }
```

A stored command containing multiple commands is treated as multiple commands by the database server. Therefore, in a stored command containing multiple commands, a call to *irexec()* will execute only the first command; subsequent commands must be executed by *irnext()*. Assume that we have a variation on the stored command, "addtitle2" which both adds a new row and selects it:

```
store addtitle2
        insert into title
        (docnum, title, onhand)
        values(max(docnum) + 1, &t, &q)
        select docnum, title, onhand
        from title
        where docnum =
                (select(max(docnum) from title)
end store
```

In this case, the stored command consists of multiple commands, so the program must use *irnext()* to advance to the **select** before calling *irbind()*.

```
1   /*
2   **  STORED2.C -- This program inserts a row to the "title" table
3   **          in the "books" database, using the stored command "addtitle2".
4   */
5
6   #include <idmlib.h>
7   #include <idmrun.h>
8
9   main()
10  {
11     IDMRUN *idmrunptr;
12     short  onhand, docnum;
13     char   *title;
14
15     INITIDMLIB("stored2");
16     idmrunptr = iropen("books");
17
18     /* assign values to pass to the stored command */
19     onhand = 5;
20     title  = "im westen nichts";
21
22     /* arrange to execute an SQL start command */
23     irxcmd(idmrunptr, "addtitle2");
24
25     /* supply parameters to the stored command */
26     irxsetp(idmrunptr, "q", iINT2, 2, __ &onhand);
27     irxsetp(idmrunptr, "t", iSTRING, -1, __ title);
28
29     /* execute the first (insert) command in the stored command */
30     irexec(idmrunptr);
31     /* execute the next (select) command in the stored command */
32     while (RETSUCCESS(irnext(idmrunptr)))
33     {
34            /* bind selected data to program variables */
35            irbind(idmrunptr, 1, iINT2, 2, __ &docnum);
36            irbind(idmrunptr, 2, iSTRING, 36, title);
37            irbind(idmrunptr, 3, iINT2, 2, __ &onhand);
38
39            /* and fetch the selected data */
40            while (RETSUCCESS(irfetch(idmrunptr)))
41                   printf("%d    %s     %d\n", docnum, title, onhand);
42
43     }
44     irclose(idmrunptr);
45     exit(RS_NORM);
46  }
```

2.3.3. Obtain Information on a Target-List Element

In a dynamic application, in which database commands are supplied by the user during program execution, the source code cannot literally specify the data types of variables to be bound with selected data, because the types of the selected data are not known prior to runtime. During program execution, after the user has input the target-list, the data types of target-list elements can be obtained with *irdesc()*. This function returns information on the data type, length and name of attributes on the database server. This information can then be stored in an array and passed to *irbind()*. The function *irdesc()* takes five arguments:

(1) an IDMRUN pointer,

(2) the position of the element in the target-list,

(3) the address of the integer where data type information will be stored,

(4) the address of the integer where length information will be stored,

(5) the address of the character string where name information will be stored.

The next example program, *mysql.c*, executes any syntactically correct SQL commands provided by the user. If the command is a **select** command, it calls *irdesc()* to obtain the types and lengths of target-list elements so the target-list elements can be bound to variables of the appropriate type.

2.3.4. Obtain Information On An IDMRUN Structure

The function *irget()* supplies information concerning the state of the IDMRUN structure. Consult the entry for *irget()* in the *Host Software Specification* or *C Run-Time Library Reference* for a complete list of the fields that can be requested and the predefined constants which are associated with them.

The function *irget()* takes four arguments:

(1) an IDMRUN pointer,

(2) the address at which returned information should be stored, as a BYTE *,

(3) the information requested, as an **int** (usually a predefined constant),

(4) the item number, if the requested information has compound fields.

A common use of *irget()* is to request a count of the number of rows selected. The code segment below requests the "done count" or number of rows read by the database server upon termination of an *irfetch()* loop. The constant for the "done count" is IP_DCNT.

```
short  docnum, onhand;
long   count;

irsql(idmrunptr, "select docnum, onhand where onhand < 2");
irexec(idmrunptr);
irbind(idmrunptr, 1, iINT2, 2, __ &docnum);
irbind(idmrunptr, 2, iINT2, 2, __ &onhand);
while (RETSUCCESS(irfetch(idmrunptr)))
       printf("%d   %d\n", docnum, onhand);

irget(idmrunptr, __ &count, IP_DCNT, 0);
printf("A total of %ld title must be reordered", count);
```

The program *mysql.c* demonstrates the use of *irget()* with compound fields.

```
1   /*
2   ** MYSQL.C -- demonstrates irdesc and irget.
3   **
4   **        Parses and executes SQL commands supplied by the user.
5   **        Displays any selected data requested.
6   **        Displays the "done status" after executing each command.
7   */
8
9  #include <idmlib.h>
10 #include <idmdone.h>
11 #include <idmrun.h>
12
13  IDMRUN    *Idmrunptr;
14
15  main()
16  {
17     char   buf[128];
18     BOOL   retrieve();
19     void   donestat();
20
21     INITIDMLIB("mysql");
22     Idmrunptr = iropen("books");
23
24     /* get input of SQL commands */
25     getprompt(buf, sizeof(buf), "Enter SQL commands:\n");
26
27     /* parse the command string */
28     if (RETSUCCESS(irsql(Idmrunptr, buf)))
29     {
30            /* execute the first SQL command */
31            irexec(Idmrunptr);
32            if (!retrieve())
33                   printf("\nWARNING: some data may have been lost.");
34            donestat();
35
36            /* execute any subsequent commands */
37            while (RETSUCCESS(irnext(Idmrunptr)))
38            {
39                   if (!retrieve())
40                          printf("\nWARNING: some data may have been lost.");
41                   donestat();
42            }
43     }
44     irclose(Idmrunptr);
45     exit(RS_NORM);
46  }
```

```
47  /*
48  **  RETRIEVE -- reads and displays data selected from the database
49  **              server.
50  **
51  ** displays only shorts, longs and char strings;
52  ** if data is another type returns FALSE, otherwise TRUE
53  */
54
55  #define MAXTARGS 6
56
57  BOOL
58  retrieve()
59  {
60     int    type, length, i, j;
61     char   *p;
62     struct
63     {
64            int    type;
65            char   name[16];
66            union
67            {
68                   short  sval;
69                   long   lval;
70                   char   pval[36];
71            }dval;
72     } datatab[MAXTARGS]; /* for selected data */
73
74     /* clear buffers for selected string data */
75     for (j = 0; j < MAXTARGS; j++)
76            strcpy(datatab[j].dval.pval,"                    ");
77
78     i = 0;
79     /* loop through target list, store description of ith element */
80     while (RETSUCCESS(irdesc(Idmrunptr, ++i, &type, &length, &p)))
81     {
82            strcpy(datatab[i].name, p);
83            datatab[i].type = type;
84            if (type == iCHAR || type == iINT2 || type == iINT4)
85                   irbind(Idmrunptr, i, type, length, __ &datatab[i].dval);
86            else
87            {
88                   printf("\nOnly characters, shorts and longs printed.\n");
89                   irflush(Idmrunptr);
90                   return (FALSE);
91            }
92     }
93     while (RETSUCCESS(irfetch(Idmrunptr)))
94     {
95            for (j = 1; j < i; j++)
96            {
97                   printf("%s = ", datatab[j].name);
98                   if (datatab[j].type == iCHAR)
99                   {
100                         printf("%s \t", datatab[j].dval.pval);
101                         strcpy(datatab[j].dval.pval, "                    ");
102                  }
103                  else if (datatab[j].type ==  iINT2)
104                         printf("%d \t", datatab[j].dval.sval);
105                  else if (datatab[j].type == iINT4)
```

```
106                         printf("%ld \t", datatab[j].dval.lval);
107             }
108             printf("\n");
109     }
110     return (TRUE);
111 }
```

```
112 /* This structure declaration is used by DONESTAT. */
113    struct item
114    {
115           char   *string;
116           int    mask;
117    };
118
119    struct  item items[] =
120    {
121           "continue",    ID_CONTINUE,
122           "error",       ID_ERROR,
123           "interrupt",   ID_INTERRUPT,
124           "abort",       ID_ABORT,
125           "count",       ID_COUNT,
126           "overflow",    ID_OVERFLOW,
127           "divide",      ID_DIVIDE,
128           "dup",         ID_DUP,
129           "timer",       ID_TIMER,
130           "inxact",      ID_INXACT,
131           "round",       ID_ROUND,
132           "underflow",   ID_UNDERFLOW,
133           "badbcd",      ID_BADBCD,
134           "tminutes",    ID_TMINUTES,
135           "logoff",      ID_LOGOFF,
136           "volume",      ID_VOLUME
137    };
138
139 /*
140 ** DONESTAT -- display donestatus after completion of every SQL command.
141 **
142 */
143
144 void
145 donestat()
146 {
147    long   status;
148    int    i;
149
150    for (i = 0; i < 16; i++)
151    {
152           irget(Idmrunptr, __ &status, IP_DSTAT, items[i].mask);
153           if (BITSET(items[i].mask, status))
154                  printf("\n%s  %ld", items[i].string, status);
155    }
156
157    irget(Idmrunptr, __ &status, IP_DINT, 0);
158    printf("\nDone integer = %ld", status);
159
160    irget(Idmrunptr, __ &status, IP_DCNT, 0);
161    printf("\nDone count = %ld\n", status);
162 }
```

2.4. Independent IDMRUN Structures

A single program can call *iropen()* several times to create several independent IDMRUN structures working simultaneously. To the database server, each structure appears as an independent user executing IDL or SQL commands. IDMLIB knows which structure to use for each function call, because each IDMRUN function passes a pointer to an IDMRUN structure as its first argument.

Multiple IDMRUN structures are necessary if a program accesses more than one database. They can be used to copy data between databases or even between database servers. A single IDMRUN structure is associated with a single database server.

Multiple IDMRUN structures are also necessary to handle a "parts explosion task". Consider a "parts" table in a database. The parts recorded in this table may be individual parts or assemblies comprised of individual parts or of other assemblies. A program which produces a list of all the parts of a given product must run a query on the main assembly and on each subassembly. These subqueries must be run on different IDMRUN structures, so that one does not loose one's position in the primary assembly. This kind of recursive query is called a "transitive closure" operation. The following example demonstrates this use of multiple IDMRUN structures.

The program *subexam.c* displays all of the host software documents for a given environment. The "parts" table in a hypothetical database contains all of the documents and document sets in the database. The "assemblies" table records the relationship between an assembly and another assembly or between an assembly and an individual document. Part numbers for top level sets (for a given environment), begin with an 'S' as in

S03 Vax/VMS Documentation Set.

Part numbers for subassemblies begin with an 'A' as in

A01 General Documentation Series.

Part numbers for individual documents begin with other letters.

To find all the documents that make up a given documentation set, a query must be run on the top-level set (which has a part number beginning with 'S'), and a subquery must be run on each subassembly (those with part numbers beginning with 'A'). The subqueries must be run on a different IDMRUN structure from the primary query. If there were deeper levels of subassembly, each level of subquery would be run with a different IDMRUN structure.

```
1   /*
2   **  SUBEXAM.C -- This program demonstrates multiple IDMRUN structures.
3   **        It displays all of the documents for a given environment.
4   */
5
6   #include <idmlib.h>
7   #include <idmrun.h>
8
9   #define   NUMSIZE       4
10  #define   TITLESIZE     50
11
12     IDMRUN *Outerptr,    *Partsptr;
13
14  main()
15  {
16     char    buf[10];
17
18     INITIDMLIB("subexam");
19
20     /* create IDMRUN structures */
21     Outerptr = iropen("books");
22     Partsptr = iropen("books");
23
24     /* select and display all the top-level sets */
25     printdoc("S*", iPCHAR);
26
27     /* get input */
28     getprompt(buf, sizeof(buf),"\nEnter a part number from one \
29           of these documentation sets: ");
30
31     /* print number and title of set to be exploded */
32     printdoc(buf, iSTRING);
33     printf("\n*********************************************\n");
34
35     /* print all the documents in the set */
36     explode(Outerptr, buf);
37
38     irclose(Outerptr);
39     irclose(Partsptr);
40     exit(RS_NORM);
41  }
```

```
42  /*
43  ** EXPLODE -- expands all assemblies into their constituent documents;
44  **        if the current document is an assembly, calls itself with a new
45  **        IDMRUN pointer to find all the constituents of the subassembly
46  **
47  **        Parameters:
48  **        irptr:    pointer to IDMRUN structure
49  **        partnum:  number of assembly for which constituent parts will
50  **                  be searched
51  */
52
53  explode(irptr, partnum)
54     IDMRUN *irptr;
55     char   partnum[NUMSIZE];
56  {
57     char   child[NUMSIZE];
58     IDMRUN *innerptr;
59
60     /* find children of the current assembly */
61     irsql(irptr, "select child from assemblies where parent = %num");
62     irsubst(irptr, "num", iSTRING, 0, partnum);
63     irexec(irptr);
64     irbind(irptr, 1, iSTRING, NUMSIZE, child);
65
66     /* print data from "parts" table for the children */
67     while (RETSUCCESS(irfetch(irptr)))
68     {
69           /* select row from "parts" table */
70           printdoc(child, iSTRING);
71
72           /* if current doc is an assembly */
73           if (child[0] == 'A')
74           {
75                 /* open a new IDMRUN structure */
76                 innerptr = iropen("books");
77                 explode(innerptr, child);
78                 irclose(innerptr);
79                 printf("\n");
80           }
81     }
82  }
```

```
83  /*
84 ** PRINTDOC -- selects and displays sets qualified by pnum.
85 **
86 **  Parameters:
87 **       pnum:  number of part for which data is displayed
88 **       ptype: type of parameter
89 **
90 **  Assumptions:
91 **       A stored command called "getpartsdata" has been defined as:
92                  store getpartsdata
93                     select docnum, title from parts
94                     where docnum = &pnum
95                  end store
96 */
97
98 printdoc(pnum, ptype)
99     char  pnum[NUMSIZE];
100    int   ptype;
101 {
102    char  partnum[NUMSIZE];
103    char  partname[TITLESIZE];
104    char  tab;
105
106    /* parse a "start getpartsdata" command */
107    irxcmd(Partsptr, "getpartsdata");
108
109    /* set stored command parameter &pnum to value of pnum */
110    irxsetp(Partsptr, "pnum", ptype, -1, (char *)pnum);
111
112    /* execute the stored command */
113    irexec(Partsptr);
114
115    /* bind targets selected by "getpartsdata" */
116    irbind(Partsptr, 1, iSTRING, NUMSIZE, partnum);
117    irbind(Partsptr, 2, iSTRING, TITLESIZE, partname);
118
119    while (RETSUCCESS(irfetch(Partsptr)))
120    {
121           /* tab if not an assembly */
122           if (partnum[0] != 'A' && partnum[0] != 'S')
123                  tab = '\t';
124           else
125                  tab = '\0';
126
127           printf("%c%s        %s\n", tab, partnum, partname);
128    }
129 }
```

2.5. Related IDMRUN Structures

Independent IDMRUN structures may not update and select a table simultaneously. If this is attempted, it appears to the database server that one user is updating the table while another is selecting data from it, and the database server waits for the select loop to complete before beginning the update. Since the application is waiting for the update to complete in order to continue the select loop, the program deadlocks.

To update a table nested inside a select loop and avoid such a deadlock, create a parent IDMRUN structure to run the select, and run the update on a child IDMRUN structure. A child IDMRUN structure is created with *irreopen()*. This function accepts a pointer to the parent IDMRUN structure as its argument and returns a pointer to a child IDMRUN structure. The database server then knows that the structures are related and allows them to share locks. A new *irreopen()* call must be made for every level of nesting.

There are restrictions on the use of *irreopen()*. It can only be called inside a transaction, and no updates can have been issued on the parent IDMRUN structure between the execution of the **set autocommit off** command and the *irreopen()* of the child IDMRUN structure. Any child IDMRUN structures opened within the transaction must be closed before a **commit work** or a **rollback work**. Updates are not committed to the database until the transaction has completed.

If two users are updating the same tuple at the same time, only one of the updates will be performed. To prevent a user from attempting to update a table which is being updated by another user, the table should be accessed with the **fulllock** option. This is the default when the type of lock is not specified. If **minlock** is specified, there is a possibility that a user may not discover until end of the transaction that none of the updates (s)he has made will be processed because another user was accessing the same rows.

The correct use of *irreopen()* is demonstrated in *reexam.c*. This program updates the "onhand" column in the "title" table of the "books" database. As each title is selected, the user is prompted for the number of copies to be added or subtracted. The program then updates the "onhand" column based on the user's input.

The **select** runs in an outer loop on the parent IDMRUN structure, while the updates are nested in a transaction running on the child IDMRUN structure. The updates are not committed to the database until the entire transaction has completed and all of the rows in the table have been accessed or the user has signaled a wish to quit.

```
1   /*
2   **  REEXAM.C -- demonstrates use of related IDMRUN structures.
3   */
4
5   #include <idmlib.h>
6   #include <idmrun.h>
7
8
9   main()
10  {
11     IDMRUN  *parentptr, *childptr;
12     short   docnum, onhand, newstock;
13     char    title[36], buf[5];
14
15     INITIDMLIB("reexam");
16
17     /* set up the parent IDMRUN structure */
18     parentptr = iropen("books");
19
20     irsql(parentptr, "set autocommit off");
21     irexec(parentptr);
22
23     /* create the child IDMRUN structure */
24     childptr = irreopen(parentptr);
25
26     /* parse and execute the select loop on parent IDMRUN structure */
27     irsql(parentptr, "select docnum, title, onhand from title");
28     irexec(parentptr);
29     irbind(parentptr, 1, iINT2, 2, __ &docnum);
30     irbind(parentptr, 2, iSTRING, sizeof(title), __ title);
31     irbind(parentptr, 3, iINT2, 2, __ &onhand);
32
33     /* parse the update on the child IDMRUN structure to execute later */
34     irsql(childptr, "update title set onhand = onhand + %newstock \
35            where docnum = %docnum and title = %title");
36
37     while (RETSUCCESS(irfetch(parentptr)))
38     {
39            printf("\nDOCNUM   TITLE                ONHAND\n");
40            printf("%d    %s                %d\n", docnum, title, onhand);
41            getprompt(buf, sizeof(buf), "\nEnter new stock \
42                   or <RETURN> to quit: ");
43            if (buf[0] == '\0')
44            {
45                   irflush(parentptr);
46                   break;
47            }
48            newstock = atos(buf);
49            printf("\nAdding %d copies of %s.\n", newstock, title);
50
51            irsubst(childptr, "newstock", iINT2, 2, __ &newstock);
52            irsubst(childptr, "docnum", iINT2, 2, __ &docnum);
53            irsubst(childptr, "title", iSTRING, sizeof(title), __ title);
54            irexec(childptr);
55     }
```

```
56      irclose(childptr);
57      irsql(parentptr, "commit work");
58      irexec(parentptr);
59
60      /* ''commit work'' processed as multiple commands */
61      while (RETSUCCESS(irnext(parentptr)))
62              ;
63      irclose(parentptr);
64      exit(RS_NORM);
65  }
```

3. Introduction to Exception Handling

In our sample programs we have made no provisions to handle abnormal returns from IDMLIB functions. We could have tested the RETCODE returned by every IDMLIB routine and taken some action on a abnormal return as in

```
if (RETSUCCESS(irexec(Idmrunptr)))
        errorfunc();
```

This is a commonly used approach, which requires that each function not only return its own RETCODE, but also check the RETCODEs of all of the functions that it calls to see if it should also return a failure. This method requires numerous conditional statements to verify normal returns and tends to isolate error messages from the functions in which the error actually occurred.

The exception handling system provided by IDMLIB uses another approach in which a single declaration is made: "If this condition exists, call this function to handle it". This method separates the detection of an exception from the code which handles it. The detection of the abnormal condition and subsequent raising of an exception is localized in the function in which the condition occurs. The action to be taken when an exception is raised is specified elsewhere, in an exception handler, which is a special function that has been declared to handle specific exception conditions.

Most IDMLIB functions raise exceptions. The specific exceptions which a particular IDMLIB routine may raise are listed in the description of the function in the *Host Software Specification* or *C Run-Time Library Reference.*

A program may recognize these exceptions and declare an exception handler to take appropriate action when an exception is raised.

3.1. Exception Identification

All IDMLIB exceptions are named by a character string prefaced by a severity code as in

A:IDMLIB.IO.CANTOPEN

The 'A' is one of nine single upper-case characters used to indicate exception severity. The severity codes are:

I: Information
: These exceptions give information. For example, copy utilities may raise an 'I' exception periodically with the expectation that the exception handler will display how much of a file or relation has been copied.

S: Success
: These are raised on successful completion of a task. For example, copy utilities may end by invoking a handler to print a success message displaying the number of tuples copied; expert users may prefer to have this information suppressed.

C: Continue
: These exceptions are raised so that a handler may prompt the user to continue with some action; for example, in a screen-based system a continue message might be generated between each frame.

R: Respond
: These exceptions indicate that an unusual but not erroneous condition has occurred that requires human intervention, e.g., "End of tape; mount next volume."

W: Warning
: These exceptions are raised when a condition may be an error.

T: Transient
: Transient exceptions are usually caused by asynchronous events, operator interrupts, transient resource exhaustion, or some problem that is due not to a user error but to a condition that is unlikely to occur again in the same section of code. The handler may invite the user to try again later.

E: Error
: Error exceptions are caused by user error. If the program continues to process, incorrect results are certain.

U: Usage
: Usage exceptions are raised when a program is invoked incorrectly. If there is a message associated with this exception it will be displayed. A "Usage" exception terminates the process exactly like an "Abort" severity exception (see below).

A: Abort

These indicate catastrophic errors. It is not possible for the current routine to continue processing. Processing aborts immediately unless an exception handler backs out.

The remainder of the exception name consists of one or more upper-case mnemonics separated by periods. The first gives the name of the facility that raised the error, in this case IDMLIB. Succeeding mnemonics specify the error condition more precisely. The last one states the error explicitly. For example, the exception name A:IDMLIB.IO.CANTOPEN indicates that the I/O subsystem of IDMLIB could not open something. What could not be opened could be passed as an exception argument by the function which raised the exception.

3.2. Exception Handlers

The program using IDMLIB can specify the action to be taken when a specific exception or type of exception is raised by declaring an exception handler with *exchandle()*.

The function *exchandle()* takes two arguments. The first is a character string naming the exception to be handled, in quotation marks. The character string identifying the exception may contain wild cards, allowing for the declaration of a handler to deal with a class of exceptions, rather than a single, specific exception. The second argument is the name of the exception handler which will handle the exception. If this argument is FUNCNULL, any handler currently in effect for the specified character string is disabled.

Suppose that we want to specify that whenever an IDMLIB function raises an exception of severity Error anywhere in a program, the program should invoke an exception handler called *myhandler*. This is arranged by including *exc.h* and calling *exchandle()* to declare the exception handler.

```
#include <idmlib.h>
#include <idmrun.h>
#include <exc.h>

main()
{
        extern  int  myhandler();

        INITIDMLIB("myprog");

        /* call myhandler for any E level exceptions raised by IDMLIB */
        exchandle("E:IDMLIB.*", myhandler);

        /* rest of program */
}
```

The scope of an exception handler is associated with the function in which the handler was declared. The handler will be called for any exception of the specified pattern raised in the same function as the call to *exchandle()* which declared the handler or at a deeper function call. The handler remains in effect until one of three events occurs:

- The function which declared the handler declares another handler or FUNCNULL associated with the same exception pattern.
- The function which declared the handler returns or exits.
- The function which declared the handler is aborted by a backout.

3.2.1. Default Handlers

If a program does not declare a handler for a particular exception, a default handler is invoked. These operate like any other exception handler except that (1) they are not automatically removed when the function that set them returns or exits, and (2) they never back out.

The default actions taken by IDMLIB abort the program for Abort level exceptions and print a message and continue processing for all exceptions of lesser severity. This is the action that will be taken when an exception is raised and no exception handlers have been declared.

If you wish your program to take some other action when an exception is raised, you must declare an exception handler. You may declare one of the standard handlers provided by IDMLIB, or you may supply your own handler. Exceptions which do not match the patterns specified in the call to *exchandle()* will be processed by the default handlers.

3.2.2. Standard Handlers

IDMLIB provides five standard handlers which are available to applications programs to use for handling a variety of exceptions. They are declared in *exc.h*. The standard handlers are

excignore()
: Ignores the exception.

excbackout()
: Backs out the program to the function that declared the handler for the named exception. Returns control to the statement following the declaration of the exception handler.

excprint()
: Prints the message associated with the error and then ignores the exception.

excprbo()
Prints the message associated with the error and then backs out.

excabort()
Aborts the program.

The following statement declares a handler to ignore all Information and Success exceptions instead of printing messages, which would be the default action. This is done by declaring the standard handler *excignore()* for all I and S exceptions.

```
exchandle("[IS]:*", excignore);
```

3.2.3. Customized Handlers

If the standard handlers do not provide the functionality required by your program, you can write your own exception handler. It must be a function which returns an integer.

The value returned by the handler determines whether the program continues execution or backs out. If the handler returns zero, the exception subsystem ignores the exception. The result is that the *excraise()* function which raised the exception returns and execution of the program continues from that point. If the handler returns nonzero, the program backs out to the *exchandle()* call which declared the handler and causes *exchandle()* to return again with the value the handler returned. Since the original call to *exchandle()* (the one that declared the exception handler) returns zero and these subsequent calls return nonzero, the program can detect the circumstances under which *exchandle()* is returning.

The handler is called with one argument, an argument vector of type char**. The zeroth element is the exception being raised and any remaining elements, all text strings, are passed by the function raising the exception. The last pointer element must be CHARNULL.

The following exception handler displays the name of the exception and a special message, calls *ircancel()*, and backs out to the *exchandle()* call that declared the handler.

```
1   myhandler(excv)
2   char   **excv;
3   {
4      extern  IDMRUN  Idmrunptr;
5
6      printf("\nERROR: %s\n", excv[0]);
7      printf("Cancelling current operations and backing out.\n")
8      printf("Please submit a new request:\n");
9      ircancel(Idmrunptr);
10     return (-1);
11  }
```

The function *excahandle()* performs just like *exchandle()* except that it allows an additional argument to be passed to the exception handler. The following call to *excahandle()* passes the IDMRUN pointer as an argument to the exception handler:

```
excahandle("E.IDMRUN.*", myhandler, idmrunptr);
```

In the following example, *myhandler()* displays a warning message, flushes selected data and returns to the function which raised the exception.

```
1   myhandler(excv, idmrunptr)
2   char      **excv;
3   IDMRUN    *idmrunptr;
4   {
5      printf("\nWARNING: %s\n", excv[0]);
6      printf("Some selected data may be lost.\n");
7      irflush(idmrunptr);
8      return (0);
9   }
```

The following program is a variation on *main()* in the source file *myidl.c* used in Section 2.3.3. Exception handling has been added to call *irflush()* or *ircancel()* if the user enters CTRL-C, and to back out if there is a syntax error in the SQL command string.

```
1  /*
2   **  MYSQL.C -- demonstrate exchandle.
3   */
4
5   #include <idmlib.h>
6   #include <idmrun.h>
7   #include <exc.h>
8
9      IDMRUN *Idmrunptr;
10     BOOL   Cancel;
11     int    handler();
12
13  main()
14  {
15     char          buf[128];
16     extern BOOL   retrieve();
17     extern void   donestat();
18
19     INITIDMLIB("mysql");
20     Idmrunptr = iropen("books");
21
22     /* back out to here on CTRL-C */
23     exchandle("T:IDMLIB.ASYNC.INT", handler);
24
25     /* get desired interpretation of CTRL-C */
26     getprompt(buf, sizeof(buf), "Enter action to take on CTRL-C \
27            (flush or cancel): ");
28     if (buf[0] == 'c' || buf[0] == 'C')
29            Cancel = TRUE;
30     else
31            Cancel = FALSE;
32
33     /* back out to here if there are errors in the command string */
34     if (exchandle("E:IDMLIB.IDM.SYNTAX", excbackout) != 0)
35            printf("Syntax error in command string; try again.\n");
36
37     /* get input of SQL commands */
38     getprompt(buf, sizeof(buf), "Enter SQL commands:\n");
39
40     /* parse the command string */
41     if (RETSUCCESS(irsql(Idmrunptr, buf)))
42     {
43            /* execute the first SQL command */
44            irexec(Idmrunptr);
45            if (!retrieve())
46                   printf("\nWARNING: some data may have been lost.");
47            donestat();
48
49            /* execute any subsequent commands */
50            while (RETSUCCESS(irnext(Idmrunptr)))
51            {
52                   if (!retrieve())
53                          printf("\nWARNING: some data may have been lost.");
55                   donestat();
56            }
57     }
58     irclose(Idmrunptr);
59     exit(RS_NORM);
60  }
```

```
61  /*
62  **  HANDLER -- call irflush or ircancel depending on value of Cancel.
63  **
64  */
65  handler(excv)
66     char   **excv;
67  {
68     if (Cancel)
69           ircancel(Idmrunptr);
70     else
71           irflush(Idmrunptr);
72     return (0);
73  }
```

4. Command-Line Arguments

IDMLIB provides a system-independent function called *crackargv()* to read command-line arguments into program variables.

To use *crackargv()* include *crackargv.h* and declare two arguments to *main()*, an integer and a pointer to an argument list as a char **. The integer is a value representing the total number of command-line arguments; it is calculated when the command-line arguments are read. The argument list (ARGLIST) describes each argument and must be declared and initialized in your C program.

4.1. The Command-Line

A command-line argument is either positional, if it is identified by its position on the command-line, or flagged, if it is identified by a flag which specifically labels the argument. When a positional argument is used, its position among the other arguments on the command-line is inflexible. When a flagged argument is used, its position on the command-line is arbitrary, but the argument must be flagged so that *crackargv()* can match the argument with the appropriate program variable based upon the name information supplied by the argument list.

The command-line

myprog 3 w smith

has three positional arguments; the program expects the numeric argument followed by the character argument followed by the string argument. If you want the user to be able to enter arguments in any order, each argument must be flagged with a name, as in

myprog –t smith –s 3 –c w (short flag)

or

myprog /text=smith /short=3 /char=w (long flag)

which could also be invoked as

myprog –c w –s 3 –t smith (short flag)

or

myprog /char=w /short=3 /text=smith (long flag)

The host environment determines whether long or short flag names should be chosen to identify flagged arguments. Short flags are used on Unix systems, long flags on most other systems.

A program may combine positional and flagged arguments. The following command-line passes one argument, a database name, as a positional argument. The other two arguments are flagged as demonstrated above.

myprog mydatabase –t smith –s 3

or

myprog mydatabase /text=smith /short=3

4.2. The Argument List

For each command-line argument being passed to the program, the following information is supplied in the initialization of the argument list:

short name

If you have chosen to use flagged arguments, and the host operating system expects short flags, you must specify the one-character flag which names the argument on the command-line. If the program uses positional arguments, the value of this field should be FLAGPOS. If the program uses long flags, this field will be ignored, but do not assign it the NULL character '\0', because this would be interpreted as the end of the argument list.

type

You must specify the type of the argument, such as FLAGINT, FLAGSHORT, FLAGLONG, FLAGCHAR, FLAGSTRING, etc. For a complete list of possible values for this field, consult the entry for *crackargv* in the *Host Software Specification* or *C Run-Time Library Reference.*

minimum length
: When a flag has a long name (see below), you must specify an integer value representing the minimum number of characters that must be specified on the command-line to make the name string unique. A value of zero may be assigned if short flags or positional arguments are used.

long name
: Some host operating systems require a long name for flags. The long name should be unique in its first four characters. If short flags or positional arguments are used, this field may be CHARNULL.

alternate name
: An alternate flag name is required on some (Multics) hosts. It can be initialized to CHARNULL if not required on your system.

value
: You must specify the address of the variable to which the argument should be assigned. It is cast as a character pointer with the "__" macro.

prompt
: A string may be displayed as a prompt if the user omits a required argument. It may be initialized to CHARNULL.

usage name
: This is the name of the argument to be displayed in a "usage" message, when the program is invoked incorrectly. If it is CHARNULL, "prompt" will be used.

The argument list is terminated by a NULL character '\0'.

The following examples illustrate the use of *crackargv()* with positional and flagged arguments.

```
1   /*
2   **  MYPROG1.C -- demonstrates use of crackargv with positional
3   **               arguments;
4   **
5   **  program must be invoked with arguments in the order of
6   **  their declarations.
7   **  i.e. "myprog 3 w smith"
8   **
9   */
10
11  #include <idmlib.h>
12  #include <crackargv.h>
13
14  /* variables for command-line arguments */
15  short   Num;
16  char    Ch;
17  char    *Text;
18
19  /* the argument list */
20  ARGLIST Args[] =
21  {
22     FLAGPOS, FLAGSHORT,  0, CHARNULL, CHARNULL, __ &Num,
23           CHARNULL, CHARNULL,
24     FLAGPOS, FLAGCHAR,   0, CHARNULL, CHARNULL, __ &Ch,
25           CHARNULL, CHARNULL,
26     FLAGPOS, FLAGSTRING, 0, CHARNULL, CHARNULL, __ &Text,
27           CHARNULL, CHARNULL,
28     '\0'
29   };
30
31   main(argc, argv)
32   int   argc;
33   char  **argv;
34   {
35           INITIDMLIB("myprog");
36           crackargv(argv, Args);
37           printf("Num = %d, Ch = %c, Text = %s\n", Num, Ch, Text);
38           exit(RS_NORM);
39   }
```

```
1   /*
2   **  MYPROG2.C -- demonstrates use of crackargv with flagged arguments;
3   **
4   ** specifies prompt string to be used when arguments are omitted;
5   ** arguments must be flagged as in "myprog -s3 -tsmith -cw"
6   ** or "myprog /short=3/text=smith/char=w"
7   */
8   #include <idmlib.h>
9   #include <crackargv.h>
10
11  /* variables for command-line arguments */
12  short   Num;
13  char    Ch;
14  char    *Text;
15
16  /* the argument list */
17  ARGLIST Args[] =
18  {
19     's', FLAGSHORT,  1, "short", CHARNULL, __ &Num,"number: ",
20           CHARNULL,
21     'c', FLAGCHAR,   1, "char",  CHARNULL, __ &Ch,  "character: ",
22           CHARNULL,
23     't', FLAGSTRING, 1, "text",  CHARNULL, __ &Text, "text: ",
24           CHARNULL,
22     '\0'
23   };
24
25   main(argc, argv)
26   int   argc;
27   char  **argv;
28   {
29           INITIDMLIB("myprog");
30           crackargv(argv, Args);
31           printf("Num = %d, Ch = %c, Text = %s\n", Num, Ch, Text);
32           exit(RS_NORM);
33   }
```

Appendix C: VMS

Compilation

The C program must be compiled with the DEC VAX-11 C compiler. The VMS logical name VAXC$INCLUDE must be defined as the name of the directory containing the include files supplied with the Host Software. This directory name should be IDM_DIR unless local modifications have been made. If no include files appear in this directory, the system manager should run the ININC.COM command procedure in the IDM_DIR directory. If VAXC$INCLUDE is already defined, add IDM_DIR to the definition of VAXC$INCLUDE by using a search list.

Loading IDMLIB Without the VAX C Run-Time Library

There are two versions of IDMLIB: a shared image version and an object library version. The shared image version is a production version. The object library version is necessary if a copy of the IDMLIB source code is required for debugging.

To link your C program to the shared image version of IDMLIB

LINK myprog, IDMLIB/OPT

If you require an object library containing all the internal and external symbols of IDMLIB for debugging, you can link with the object library version. This will take considerably longer to link and produce a much larger image than the shared image version.

LINK myprog, IDMOBJ/OPT

Loading IDMLIB With the VAX C Run-Time Library

If ISTDIO is loaded, all modules which make calls to standard I/O routines must include *istdio.h* instead of *idmlib.h*.

The command-line for linking the shared image versions of IDMLIB and the standard library compatibility modules is

LINK myprog, IDMLIBRTL/OPT

Appendix B: UNIX

Compilation

The requirements for a suitable C compiler to support IDMLIB are listed in the introduction to the *Host Software Specification.*

Loading IDMLIB Without the Standard I/O Library

Linking the IDM support library with the C program can be done as part of the compilation phase, as in

cc –o myprog myprog.c –lidmlib

Loading IDMLIB With the Standard I/O Library

If ISTDIO is loaded, all modules which make calls to standard I/O routines must include *istdio.h* instead of *idmlib.h.*

Load ISTDIO before IDMLIB.

cc –o myprog myprog.o –listdio –lidmlib.

```
1  #include <istdio.h>
2
3  main()
4  {
5      INITIDMLIB("test");
6      fprintf(stderr, "This will use the standard I/O library.");
7      exit(RS_NORM);
8  }
```

or

```
1  #include <istdio.h>
2
3  main()
4  {
5      INITIDMLIB("test");
6      ifprintf(istderr, "This will use IDMLIB.");
7      exit(RS_NORM);
8  }
```

Implementation

When ISTDIO is loaded and *istdio.h* included, all IDMLIB names which previously conflicted with standard I/O names will unambiguously refer to the standard I/O library routine. To reference the IDMLIB counterpart, preface the standard I/O library name with the character "i".

Standard I/O	*IDMLIB*
printf	iprintf
sprintf	isprintf
stdin	istdin
stdout	istdout
stderr	istderr

When using both libraries, the programmer must insure that standard I/O routines use standard I/O file pointers, and IDMLIB routines use IDMLIB file pointers. The following program is incorrect, because it passes the standard I/O library variable, *stderr* to the IDMLIB function *ifprintf*.

```
1  #include <istdio.h>
2
3  main()
4  {
5      INITIDMLIB("test");
6      ifprintf(stderr, "This might produce undesirable results.");
7      exit(RS_NORM);
8  }
```

A correct implementation would be:

Appendix A: Incorporating the Standard I/O Library

Standard I/O Functionality in IDMLIB

Some of the functionality of the standard I/O library is provided by IDMLIB routines and symbols with the same names as their counterparts in the standard I/O library. These routines are

Functions	*Macros*	*Variables*
printf	EOF	stderr
sprintf	NULL	stdin
strcat		stdout
strncat		
strcmp		
strncmp		
strcpy		
strncpy		
strlen		
strchr		
strrchr		
atoi		
atof		
atol		
exit		

The standard library is generally incompatible with IDMLIB. This means that standard I/O functions should not be called, and header files containing definitions for those functions, such as *stdio.h*, should not be included for modules which use IDMLIB routines, unless the provisions described below are made to resolve the incompatibilities. These provisions are currently available for programs running on Unix, VMS or PC/MS-DOS systems.

Compatibility Using ISTDIO

If a C program module makes calls to IDMLIB and the standard I/O library, a special compatibility library called ISTDIO must be loaded before IDMLIB. If ISTDIO is loaded, the modules using both libraries must include *istdio.h* instead of *idmlib.h*. Consult the appropriate appendix for your host environment for instructions on loading IDMLIB and ISTDIO.

To [illegible] the object library versions of IDMLIB and the standard library compatibility modules

To link the object library versions of IDMLIB and the standard library compatibility modules

```
LINK myprog, IDMOBJRTL/OPT
```

Appendix D: PC/MS-DOS

Compilation

The C program should be compiled using the Microsoft C compiler, version 4.0 or later. Modules which will be linked with IDMLIB must be compiled with the /AL flag to specify the large model library and the /Gs flag to turn off stack checking.

The following example compiles a source file called *MYPROG.C* and assumes that the environment variable INCLUDE is set to the directory which contains the IDMLIB header files.

```
MSC /AL /Gs MYPROG.C, MYPROG.OBJ;
```

Loading IDMLIB Without the Standard I/O Library

Object files must be linked with the Microsoft linker. The following example loads the object file and IDMLIB to produce an executable file named *MYPROG.EXE*. The /STACK flag is used because most modules linked with IDMLIB require 10000 bytes of stack space. This example assumes that the environment variable LIB includes the path to the the directory containing Microsoft's libraries and the path to the directory which contains IDMLIB.

```
LINK /STACK:10000 MYPROG.OBJ,MYPROG.EXE,,IDMLIB;
```

Loading IDMLIB With the Standard I/O Library

If ISTDIO is loaded, all modules which make calls to standard I/O routines must include *ISTDIO.H* instead of *IDMLIB.H*.

This example assumes that the environment variable LIB includes the path to the the directory containing Microsoft's libraries and the path to the directory which contains ISTDIO.

```
LINK /STACK:10000 MYPROG.OBJ,MYPROG.EXE,,ISTDIO,IDMLIB;
```

Appendix E: AOS/VS

Compilation

The command to compile a C program named *myprog.c* is:

```
cc myprog.c :idm:include/search
```

Loading IDMLIB Without the Standard I/O Library

To link myprog.ob with IDMLIB to produce an executable object called *myprog*, the command is:

```
ccl/o=myprog/tasks=4 myprog.ob :idm:lib:rclib.lb &
      :idm:lib:idmlib.lb :idm:lib:itpusr.lb
```

The ampersand indicates a continuation line and is not part of the link command.

ISTDIO is not currently available for this environment.

Index of Terms