Britton Lee Host Software

RIC USER'S GUIDE

(R3v5)

March 1988

Part Number 205-1393-004

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 program running on a host computer written in a language which is a combination of a procedural programming language, such as C, and a non-procedural database query language, such as IDL.

Britton Lee has developed a precompiler called RIC, which handles IDL queries embedded in C. RIC translates program statements written in a mixture of IDL and C into pure C code which makes calls to Britton Lee's host software subroutine library, IDMLIB. The pure C output of the precompiler can then be compiled with a C compiler.

"Introduction to RIC" contains a general description of RIC and covers general information for anyone writing programs in RIC.

"Programming with RIC" describes in detail how IDL commands are embedded in C code, how C expressions may be used in IDL commands, and how IDL **retrieve** queries are handled in RIC programs.

"Advanced Programming With RIC" describes RIC support for stored commands and transactions. It also contains some examples of RIC programs which make direct calls to IDMLIB. Users of this chapter who are not familiar with IDMLIB should consult the *Idmlib User's Guide*.

The appendices cover various aspects of building a compilable pure C program from RIC source code.

1. Introduction to RIC

RIC is a precompiler which accepts a file with IDL commands embedded in C code as input and generates a file containing pure C code as output. The output makes calls to the subroutine library, IDMLIB. This C program can then be compiled with a C compiler and linked with IDMLIB to create an executable object which runs on the host computer system and accesses data on the Britton Lee database server.

$$\begin{tikzcd} \textit{prog.ric} \arrow[r, "\text{RIC precompiler}"] & \textit{prog.c} \arrow[r, "\text{C compiler}"] & \textit{prog} \end{tikzcd}$$

RIC itself is written in C and has been ported to all operating systems currently supported by Britton Lee Host Software.

The prerequisites for using RIC are a solid knowledge of the C programming language and some knowledge of the Intelligent Database Language (IDL). The requirement for a solid knowledge of C cannot be overemphasized. It is not feasible to learn C and RIC simultaneously, since the power of C is achieved at the cost of considerable difficulty for the novice. The vast majority of the difficulties novice C programmers encounter using RIC are found to be problems with the finer points of C.

1.1. Overview

Throughout this guide, "source file" refers to the file with embedded IDL commands precompiled by RIC and "output file" refers to the pure C file produced by RIC to be compiled by a C compiler. The word "RIC" refers both to the precompiler and to the embedded language which it precompiles.

1.1.1. Input to RIC

The following RIC program executes a single command on the database server.

```
/*
** SIMPLE.RIC
**   This program appends a tuple to "myrelation".
*/

main()
{
    INITRIC("simple");

    $append to myrelation
    (
        num = 1,
        name = "agatha"
    );

    exit(RS_NORM);
}
```

This brief example demonstrates the minimal requirements for any RIC program:

- The RIC source filename must have the suffix **ric, rc** or no suffix. We could name the source file *simple.ric*, *simple.rc* or *simplesrc* but not *simple.src*.
- All RIC programs must call INITRIC with the name of the executable object as its argument. INITRIC initializes IDMLIB and the RIC runtime library environment. INITRIC may be called only once by a RIC program.
- The program contains at least one embedded IDL statement introduced by a dollar sign ($). RIC would precompile a pure C program without any embedded IDL statements, but there would be little point to this exercise, since the purpose of using RIC is to write programs which contain IDL embedded in C.
- All programs must terminate with an explicit call to the IDMLIB function **exit** with an IDMLIB return code as its argument. These return codes 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.

1.1.2. Precompilation

We can precompile *simple.ric* with the command

ric –d "mydatabase" simple.ric (Unix)

ric /dbname=mydatabase simple.ric (other)

The **–d** or **/dbname** *database* option specifies the database to be accessed. There is no default value for the database, so this option must be specified if the database is being indicated at precompile time.

An alternative to specifying the database and/or the device connecting the host computer with the database server at precompile time is specifing them at runtime using the macros RCDEVICE and RCDBNAME. These macros may appear before or after the call to INITRIC, but they must appear before the first executable IDL command. For example, we rewrite the program above in Section 1.1.1 as

```
/*
** SIMPLE2.RIC -- this program appends a tuple to "myrelation".
**
*/
main()
{
    RCDEVICE("host%xns");
    RCDBNAME("mydatabase");
    INITRIC("simple");

    $append to myrelation
    (
        num = 1,
        name = 'agatha'
    );

    exit(RS_NORM);
}
```

These macros are particularly useful if the program gets the database name from the command-line using the IDMLIB function *crackargv()*, as demonstrated by the following example. For more information on how to use *crackargv()*, consult Chapter 4 of the *Idmlib User's Guide.*

```
/*
 ** SIMPLE3.RIC -- this program appends a tuple to "myrelation".
 **
 **        Gets database name from the command-line using crackargv.
 **        Device name is the default in environmental variable IDMDEV.
 */

#include <crackargv.h>

char      *Dbase;

/* the argument list */
ARGLIST   Args[] =
{
   FLAGPOS, FLAGSTRING, 0, CHARNULL, CHARNULL, __ &Dbase,
   "Enter database:", CHARNULL,
   '\0'
};

main(argc, argv)
   int    argc;
   char   *argv[];
{
   INITRIC("simple");
   crackargv(argv, Args);
   RCDBNAME(Dbase);

   $append to myrelation
   (
       num = 1,
       name = "agatha"
   );

   exit(RS_NORM);
}
```

When the database name is supplied at runtime, the command to precompile the RIC source is

ric simple3.ric

1.1.3. Output from RIC

RIC's output from *simple.ric* is the following pure C file named *simple.c*:

```
#include <rcinclude.h>
static char *RcProg = "";
static char *RcCDB = "mydatabase";
static char *RcDevice = "host%xns";
#line 1 "simple.ric"
/*
** SIMPLE.RIC
**      This program appends a tuple to "myrelation".
*/

main()
{
        INITRIC("simple");

        {
                rcutree(getutree(0));
                RcResult = rcexec();
        }

#line 14 "simple.ric"
        exit(RS_NORM);
}

#line 18 "after end of file on simple.ric"

static char *utrees[] = {
                    "\000\000\000\000\000\000\001\000\000\000\000\000/\000\006
                    \000agatha\003\000\241\000\004\000name\000\0000\000\001
                    \000\001\003\000\241\000\003\000num\000\000",
                    "\002\000\000\000\003\000\264\000\002\000\000\000\000\000
                    \343\000\013\000\000myrelation\000\000)\000\002\000\001
                    \002\001\000\200\001\000\000\002\000\200\001\000\000\003",
                    "\000\200\001\000\000\003\000\303\000\000\000\001\000\374
                    \001\010\000\001\000\000\000\000\000\000\000\377\377\000",
                    "endoftrees"};

static short utlen[] = {
                128, 0};

static short utpos[] = {
                0, 3};

static ITREE *utp[1];
static ITREE *
        getutree(i) int i;
{
        if (utp[i] == ITNULL)
                utp[i] = (ITREE *) rcgetutree(utpos[i],
                utpos[i + 1], utlen[i], utrees);
        return (utp[i]);
}
```

A comparison between *simple.ric*, the source file in Section 1.1.1, and *simple.c*, the output file in Section 1.1.3, demonstrates how RIC precompiles a syntactically correct RIC source file.

First, RIC adds four lines of code to the top of the output file. The first line includes *<rcinclude.h>* which in turn includes all the header files needed to compile the source. The next three lines initialize static variables. *RcProg* holds the names of stored programs which are defined when the RIC source is compiled with the **–n** or **/progname** option. This option is discussed in Appendix A. Since we did not compile this program with this option, the value of *RcProg* is an empty string. *RcDevice* stores the name of the device connecting the host system to the database server, which here is the default device name taken from the environmental variable IDMDEV. *RcCDB* stores the name of the database, which was taken from the command-line option with which RIC was invoked.

The next line, #line 1 "simple.ric", is a directive to the C compiler which will ultimately compile the output file. It can be translated as "Consider the next line as line 1 for the purpose of error messages". Predictably, the next line in the output file corresponds to line 1 of the RIC source. These *#line* directives are used by the C compiler to synchronize the lines in the source file with the lines in the output file. This enables the C compiler to give error messages which reference lines in the original RIC source instead of in the pure C file which it is compiling.

From the beginning of *main* to the closing bracket of the source file, RIC scans the file copying the input file to the output file until it reads a dollar sign ($). The dollar sign ($) is a signal to RIC that it has encountered either an IDL statement or a C identifier which is used in an IDL statement. Either situation requires special processing by RIC. In this example, RIC encounters an IDL statement in the form of the **append** command in lines 10 through 14 of the source. RIC transforms this statement into

```
{
        rcutree(getutree(0));
        RcResult = rcexec();
}
```

in the output file. RIC adds the code for *getutree* to the bottom of the output file. It also adds the specific information passed to *getutree* (in this case the name of the relation and the attributes being appended) as a series of octal bytes comprising the variable *utrees*. These bytes define the specific IDL command as a tree which can be understood by the database server. RIC then synchronizes with another *#line* directive and continues in this manner until end-of-file.

1.1.4. Compilation

The output file, *simple.c*, which RIC creates from *simple.ric* can be compiled with the C compiler normally used on the host system. You must load IDMLIB and you may have to specify the directory containing the IDMLIB header files. We can compile *simple.c* with the command-line

```
cc -I/usr/include/idm -o simple simple.c -lidmlib
```

on our Unix system. This produces an executable object called ***simple***. Consult the entry for RIC (1I) in the *Host Software Specification* for Unix systems and the *Command Summary* for other systems for instructions on compiling and linking a RIC program in your host environment. If the program also makes calls to the standard I/O library, consult the appropriate appendix of the *Idmlib User's Guide* for instructions on building an object file incorporating IDMLIB and ISTDIO in your host environment.

1.2. Data Types

RIC maps C variables to corresponding data structures on the database server. The following table shows the correspondence between these database server types and C data types.

Database Server Type	*Length in Bytes*	*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[]
iBCD	variable	BCDNO
iBCDFLT	variable	BCDNO

The types BOOL and BCDNO are macros contained in one of the header files inserted by RIC.

Data of the database server BINARY data type should be retrieved from or appended to an array of type char[]. Conversion between BINARY and CHAR must be done on the database server, not on the host, using the *binary* and *string* functions within the query as indicated below.

```
$ char  bindata[100];

$ range of m = myrelation;

/* append binary data */
$ append to myrelation
(
        binattr = binary($bindata);
);

/* retrieve binary data */
$ retrieve ($bindata = string(100, m.binattr));
```

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 RIC 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. Error Messages

RIC produces error messages only for statements beginning with a dollar sign ($). It assumes that other statements are valid C and passes them through to the output file without parsing them. Thus it is perfectly possible for a RIC program which is full of C syntax errors to come through the precompiler stage eliciting no error messages. The error messages will appear at compile time.

Because of the *#line* directives mentioned in Section 1.1.3, error messages from the C compiler will reference the RIC source, not the C compiler source, so that the RIC programmer can make corrections on the original RIC source file and then run this file through RIC again. It should never be necessary for the RIC programmer to write on the RIC output (the **.c** file).

The line referenced in the error message is always the last line of the statement containing the error. An error message for a thirty-line **struct** declaration with an error in the fifth line will cite the thirtieth line as the source of the error.

1.4. References

1.4.1. RIC (1I)

An indispensible reference for anyone using RIC is the entry for RIC in Section 1I of the *Host Software Specification* for Unix Systems and the *Command Summary* for other systems. This document provides a formal description of the RIC program which includes

- a synopsis of the invocation of RIC
- a description of all flags and command-line arguments
- a description of what the program does
- a list of IDL queries that may be used in RIC programs
- a list of IDL functions which may be used in RIC programs
- a synopsis of the use of C expressions in IDL statements
- some examples of invocations of RIC
- a list of related documents

1.4.2. INITRC

The macros INITRIC, RCDBNAME, and RCDEVICE are formally documented under "INITRC" in Section 3 of the *Host Software Specification* for Unix Systems and the *C Run-Time Library Reference* for other systems.

1.4.3. IDMLIB

A formal description of all IDMLIB functions is contained in Section 3 of the *Host Software Specification* for Unix Systems and the *C Run-Time Library Reference* for other systems.

An informal description of some relevant portions of IDMLIB is available in the *Idmlib User's Guide*.

1.4.4. IDL

The Intelligent Database Language (IDL) is documented in the *IDL Reference Manual*. Any differences between the interactive version of IDL used in this reference and the IDL accepted by RIC are noted in the reference documentation for RIC (1I) listed above.

1.4.5. RSC

Britton Lee also has a precompiler called RSC which precompiles programs containing SQL statements embedded in C. For more information concerning RSC, consult the *RSC User's Guide.*

2. Programming with RIC

This chapter provides basic information needed to use RIC: how to **embed** IDL statements in C code, how to use C expressions in IDL statements, and **how** to construct code for retrieving data from the database server.

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

```
create title
(
        docnum = i2,
        title = c35,
        onhand = i2
)

create author
(
        authnum = i2,
        first = c10,
        last = c15
)

create authttl
(
        authnum = i2,
        docnum = i2
)
```

2.1. IDL in C Code

The following RIC program embeds an IDL **range** statement and an IDL **append** command in C code.

```
/*
**  APPLIT.RIC
**
**          This program appends a tuple to the "title" relation.
*/

main()
{
    INITRIC("applit");

    $range of t is title;
    $append to title
    (
            docnum = max(t.docnum) + 1,
            title = "a flag for sunrise",
            onhand = 7
    );

    exit(RS_NORM);
}
```

2.1.1. Delineating IDL Commands

An embedded IDL command always begins with a dollar sign ($). If there were no dollar sign, RIC would assume that the command was C code and pass it through to the output file where it would elicit a syntax error from the C compiler.

An embedded IDL command terminates with a semicolon (;) except for a looped **retrieve** command which has its IDL portion delineated by a left curly brace ({). This exception is illustrated in Section 2.3.2.

An IDL command embedded in a RIC program may span more than one line. Multiple IDL commands may share a single line, as long as each command is preceded by a dollar sign and terminated with a semicolon (;) or left curly brace ({). The following embedded commands are valid input to RIC:

```
$range of a is author;  $range of t is title;  $range of l is authttl;

$append to title
(
docnum = max(t.docnum) + 1,
title = "a flag for sunrise",
onhand = 7
);

$retrieve(a.last, a.first)
{
        printf("\n %s, %s\n", first, last);
}
```

2.1.2. Range Statements

An IDL **range** statement is a declarative statement which assigns a relation name to a range variable. IDL commands subsequent to the **range** statement reference the relation through its range variable. The range variable is used by the **idl program**, but it has no direct effect on the database. Consult the entries for **range and** *range_var* in the *IDL Reference Manual* for general information about **range** statements.

2.1.2.1. Scope of Range Variables

A range variable may be declared local to a function or global and therefore visible to all functions. As with C identifiers, a local range variable with the same name as a global range variable hides the declaration of the global variable for the extent of the function. This is demonstrated in the following program.

```
/*
** SCOPE.RIC
**
** This program demonstrates the scope of range variables across
** function calls.
*/

/* a and t are global */
$range of a is author;
$range of t is title;

main()
{
   $char  first[11];
   void   func();

   INITRIC("scope");

   /* a is bound to author */
   $delete a where a.authnum > 12;

   /* t is bound to title */
   $delete t where t.docnum > 15;

   func();

   /* a is still bound to author, unchanged by func() */
   $retrieve (a.first) where a.last = "lawrence";

   exit(RS_NORM);
}

/*
** FUNC -- declares a local range variable 'a'.
*/

void
func()
{
   /* a is local */
   $range of a is animals;

   /* a is bound to animals */
   $replace a (type = "canine")
          where a.type = "dog";

   /* t is bound to title from the global declaration */
   $replace t (title = "tender is the night")
          where t.title = "the great gatsby";
}
```

Unlike C identifiers, range variables do not nest within functions. A redefinition of the same range variable within a function holds for the remainder of the function, even if the redefinition takes place at a deeper level of nesting than the original definition.

In addition, when the declaration of a range variable is subject to the evaluation of a conditional statement, the declaration will be made regardless of the evaluation of the conditional statement.

```
/*
**  SURPRISE
**
**        Demonstrates how range variables do not nest
**        within functions.
*/

void
surprise(x)
   BOOL x;
{
   $range of a is author;

   if (x)
   {
         /* some C code */

         $range of a is animals;
         $replace a (type = "feline")
               where a.type = "cat";
   }

   /* name is now bound to animals, whether x was TRUE or FALSE */
   $replace a (type = "homo sapiens")
         where a.name = "bonzo";
}
```

2.1.2.2. Fixed Range Variables

All of the range variables declared thus far are fixed range variables, meaning that the binding of the range variable to a relation by the IDL **range** statement is not altered from one run of the application to the next.

The syntax of a **range** statement declaring a fixed range variable is

$range of range_var **is** rel_name [**with** options];.

Fixed range variables are defined during precompilation. Fixed range variables are not C identifiers and thus may not be substituted for C expressions. They may not have values assigned to them, be passed as arguments, or be declared as formal parameters.

2.1.2.3. Dynamic Range Variables

Dynamic range variables are used when the binding between a range variable and a relation name occurs at runtime. There are two discrete steps involved in setting a dynamic range variable: declaration and assignment.

The syntax of a **range** statement declaring a dynamic range variable is

$range of range_var **is** :range_var [**with** options];.

Early versions of RIC used a percent sign (%) instead of a colon (:) as in

$range of range_var **is** %range_var [**with** options];.

This allocates storage in the range table maintained by the host software.

Dynamic range variables must be declared as global. The precompiler gives an error message for a dynamic range variable declared within a function. Any options associated with a dynamic range variable are specified in the declaration:

```
$range of a is :a with logging;
```

Once the relation name to be bound to the dynamic range variable is known, it can be assigned. The syntax of an executable **range** statement for assigning a relation name to a range variable is

$range of range_var **is** $(rel_name);

The *rel_name* may be either a literal character string or an identifier having a string valued expression. If a literal character string is used, it must be enclosed in parentheses. If an identifier is used, the parentheses are optional. Either of the following statements binds the range variable 'a' to the "author" relation.

```
$range of a is $("author");
```

or

```
strcpy(relname, "author");
$range of a is $relname;
```

The following example demonstrates the use of dynamic range variables.

```
/*
**  GETREL -- assigns a dynamic range variable.
*/

/* declare a dynamic range variable 'r' outside the function */
$range of r is :r;

void
getrel()
{
    $char  relname[13];

    /* get the relation name using the IDMLIB function getprompt() */
    getprompt(relname, sizeof(relname), "Enter the relation name:");

    /* bind 'r' to the name input by the user */
    $range of r is $relname;
}
```

2.1.3. Placement of IDL Commands

There are two types of IDL statements which can appear in a RIC program: executable and declarative. Executable statements are commands which affect a database, such as **append**, **delete**, **replace**, and **retrieve**. A **range** statement which assigns the value of a dynamic range variable is also an executable statement. Embedded executable IDL statements may occur anywhere in a RIC source file where executable C statements may occur.

Range statements which declare fixed range variables, and the declarative portion of a **range** statement which allocates storage for a dynamic range variable are declarative IDL statements. These may occur anywhere in a source file, but they are usually placed with C declarations before the executable code. Declarations of dynamic range variables are global and must appear outside of functions.

2.2. C Expressions in IDL Commands

C identifiers which appear in embedded IDL commands must be flagged with a dollar sign, both where they are declared and in the IDL statement in which they are used. The following program uses C identifiers in an IDL **append** command.

```
 1 /*
 2 **  APPVAR.RIC
 3 **
 4 **        This program appends tuples to the "title" relation,
 5 **        getting input from the user.
 6 **
 7 **        Demonstrates the use of C variables in RIC code.
 8 **        Gets database from the user at runtime using getprompt().
 9 */
10
11
12 main()
13 {
14     /* $ prefaces declaration of C identifiers to be used in IDL code */
15     $char  newtitle[36];
16     $short newquan;
17     char   buf[5];
18     char   dbname[13];
19
20     $range of t is title;
21
22     INITRIC("appvar");
23     getprompt(dbname, sizeof(dbname), "Enter database name: ");
24     RCDBNAME(dbname);
25
26     /* loop on user input until user signals <RETURN> */
27     for (;;)
28     {
29            /* get user input for newtitle and newquan */
30            getprompt(newtitle, sizeof(newtitle),
31                   "Enter title or <RETURN> to quit:");
32            if (newtitle[0] == '\0')
33                   break;
34            getprompt(buf, sizeof(buf), "Enter quantity: ");
35            newquan = atos(buf);
36
37            /* append the new tuple to the "title" relation */
38            $append to title
39            (
40                   docnum = max(t.docnum) + 1,
41                   title = $newtitle,
42                   onhand = $newquan
43            );
44     }
45
46     exit(RS_NORM);
47 }
```

2.2.1. C Declarations Used by RIC

The C declarations on lines 15 and 16, for *newtitle* and *newquan*, are prefaced with a dollar sign ($) because the identifiers are used in IDL statements in the executable part of the program. This is in contrast to the C declarations declared on lines 17 and 18, for *buf* and *dbname*, which do not require a dollar sign ($) because these identifiers are used only in pure C code.

In lines 41 and 42, inside the **append** command, *newtitle* and *newquan* appear prefaced with dollar signs ($) because they are embedded in IDL code. When these same variables are used in pure C code, in lines 30, 32, and 35, they are not prefaced with a dollar sign ($). The point to remember is that the dollar sign ($) must preface a C declaration only if the identifier will later appear in an IDL statement, and the dollar sign ($) must preface the identifer when it appears in an IDL statement.

C identifiers used in IDL code can be of any data type and storage class known to C, except for the **register** storage class.

The default size of the symbol table maintained for declarations prefaced with dollar signs ($) is 100. A program which requires more than 100 C identifiers in IDL code should be precompiled with the **–S** or **/symtabsize** *size* option, where *size* is the desired size of the symbol table.

Any C expression may be used in place of an IDL expression. If the C expression is a simple C variable name, it need not be enclosed in parentheses when it is used in the IDL statement:

```
$short newquan;

/* no parentheses necessary - newquan is a simple variable name */
$replace t (onhand = $newquan);
```

If the C expression is more complex than a simple variable name, the expression must be enclosed in parentheses in the IDL statement. In addition, each C identifier used in the expression must be declared with a dollar sign ($).

```
$short newquan;
$short oldquan;
$int   num;

/* parentheses necessary for complex expression */
$replace t (onhand = $((newquan + oldquan) / num)) . . .
```

RIC evaluates the types of C expressions prefaced with dollar signs ($) and generates the code for performing any necesssary type conversions. It will issue an error message when type conversions are not possible or do not make sense such as trying to convert "ab123" to an **int** or storing too large a number as a **short**.

RIC can handle all C expressions, including casts such as

```
$replace t (docnum = $((short) 3.7)) . . .
```

When a C string's length is greater than the length of the attribute in which it is to be stored, the string is truncated.

2.3. Retrieves

The last program used C expressions as values in an IDL **append** command. RIC also uses C identifiers to store the values of target-list elements fetched from the database server by an IDL **retrieve** command. Identifiers which will store retrieved data are prefaced with dollar signs ($) when they are declared and when they are used in the body of the **retrieve** command:

```
$short        num;
$char         lname[16];

$range of a is author;
$retrieve ($num = a.authnum, $lname = a.last)
       where . . .
```

If the name of the C identifier is identical with the name of the attribute on the database server, an explicit assignment is not needed in the **retrieve** command. Below we have changed the names of the C identifiers to match those of the retrieved target-list elements to indicate clearly the relationship between the database server attributes and the C identifiers and to avoid having to make explicit assignments in the **retrieve** command. The following code implicitly assigns the retrieved data to *authnum* and *last*.

```
$short        authnum;
$char         last[16];

$range of a is author;
$retrieve (a.authnum, a.last)
       where . . .
```

There are two forms of **retrieve** command in RIC programs, the "singleton" form, which retrieves a single tuple and simply stores its results, and the "looped" form, which may retrieve more than one tuple and process retrieved data as it is fetched.

2.3.1. Singleton Retrieves

This program appends a tuple to the "title" relation and then retrieves it.

```
/*
**  SINGLE.RIC
**
**          This program appends a specific title to the "title" relation
**          and retrieves it. Demonstrates singleton retrieves.
**
*/

main()
{
    $char  title[36];
    $short onhand;
    $short docnum;

    $range of t is title;

    INITRIC("single");

    /* append a new tuple */
    $append to title
    (
            docnum = max(t.docnum) + 1,
            title = "the color purple",
            onhand = 7
    );

    /* retrieve the new tuple, implicit assignment of target elements */
    $retrieve (t.docnum, t.title, t.onhand)
            where t.docnum = max(t.docnum);

    printf("\nThe new tuple is:\n%d\t%s\t%d\n", docnum, title, onhand);

    exit(RS_NORM);
}
```

The singleton form of the **retrieve** command is useful only when a single tuple will be retrieved by the query. If more than one tuple were to satisfy the qualification, only one of them would be retrieved, and RIC cannot guarantee which one. The singleton form is most commonly used when the **retrieve** command is qualified by a unique key to the relation or by an aggregate with no **by** clause which yields a single value, as demonstrated in the example above.

2.3.2. Looped Retrieves

Looped **retrieve** commands in RIC source are precompiled into C **for** loops in the output file. C code to process the retrieved data is executed inside the **for** loop for each retrieved tuple. If no tuples are retrieved, the loop body does not execute.

```
 1  /*
 2  **  LOOPED.RIC
 3  **
 4  **          This program retrieves tuples for titles which are low in stock
 5  **          from the "title" relation. Demonstrates looped retrieves.
 6  */
 7
 8  main()
 9  {
10      $char  title[36];
11      $short onhand;
12      $short docnum;
13
14      $range of t is title;
15
16      INITRIC("looped");
17
18      printf("TITLES TO BE REORDERED\n");
19
20      /* retrieve tuples */
21      $retrieve (t.docnum, t.title, t.onhand)
22              where t.onhand < 6
23      {
24              if (onhand < 0)
25                      printf("\nPOSSIBLE ERROR IN DATABASE: ");
26
27              printf("%d\t%s\t%d\n", docnum, title, onhand);
28      }
29
30      exit(RS_NORM);
31 }
```

It is important to note that the **retrieve** command on lines 21 and 22 is not terminated by a semicolon (;). Instead, the IDL portion of the command is terminated by the left curly brace ({) which begins the body of the loop. The presence of the left curly brace ({) informs RIC that this is a looped **retrieve**. If the **retrieve** command were terminated with a semicolon (;) RIC would treat it as a singleton and generate code to retrieve one tuple. As in C, the body of the loop terminates with a right curly brace (}).

The only acceptable way to leave a **retrieve** loop prematurely (before all tuples have been fetched) is by using the C **break** statement. Never use a **return**, **goto**, or **longjump** to exit a **retrieve** loop. This point cannot be overemphasized. The precompiler cannot detect when a **retrieve** loop has been improperly exited, but such a situation will produce strange and unpredictable behavior at runtime.

Each C identifier in which retrieved data is stored is evaluated *n* plus 1 times, where *n* is the number of tuples retrieved. This is important when the evaluation of this term has a side effect, as demonstrated in the following example:

```
/*
**  GETDOCS
**
**        Retrieves the tuples and puts them in an array, docs[].
**        Returns the number of tuples retrieved.
*/

getdocs(docs)
   $int   docs[];
{
   int    n = 0;

   $range of t is title;

   $retrieve($(docs[n++]) = t. docnum)
          where t.onhand  < 8
   {
          /* just fill the array */
          continue;
   }
   /* return the number of documents retrieved */
   return (n - 1);
}
```

The function *rccount()* provides an alternative method for obtaining the number of tuples affected by the last IDL command executed. Using *rccount()* the function above would look like this:

```
/*
**  GETDOCS
**
**        Retrieves the tuples and puts them in an array, docs[].
**        Returns the number of tuples retrieved.
*/

getdocs(docs)
   $int   docs[];
{
   int    n = 0;

   $range of t is title;

   $retrieve($(docs[n++]) = t. docnum)
          where t.onhand < 8
   {
          /* just fill the array */
          continue;
   }
   /* return the number of documents retrieved */
   return (rccount());
}
```

2.3.3. Memory for Strings

It is the programmer's responsibily to allocate memory for retrieved strings. The following function illustrates improper usage:

```
/*
** BADFUNC -- illustrates improper usage.
*/

badfunc()
{
   $char  *last;

   /* last does not point to anything */
   $retrieve ($last = a.last)
   {
          printf("%s ",last);
   }
}
```

Memory can be allocated statically as in

```
/*
** GOODFUNC -- illustrates proper usage.
*/

goodfunc()
{
   /* allocate for the length of the attribute plus the NULL byte */
   $char  last[16];

   $retrieve ($last = a.last)
   {
          printf("%s ",last);
   }
}
```

Memory can also be allocated dynamically using the IDMLIB functions *xalloc* or *savestr*. The function *savestr* allocates enough memory to store the retrieved string and then copies it. Both functions are documented under xalloc (3I) in the *Host Software Specification* and *C Run-Time Library Reference.*

The following program builds a linked list of authors' names from the "author" relation, using *xalloc* and *savestr.*

```
/*
**  LISTAUTHS.RIC
**
**       This program builds a linked list of authors names from the
**       "author" relation. Demonstrates xalloc and savestr.
*/

#include <idmmpool.h>

    struct list
    {
            char   *name;
            struct list  *next;
    }

main()
{
    $char  inbuf[26];
    struct list  *plist = (struct list *) NULL;
    struct list  *new;
    struct list  *p;

    $range of a is author;

    INITRIC("listauths");

    $retrieve ($inbuf = concat(a.first, a.last))
    {
            /* allocate a new list element */
            new = (struct list *) xalloc(sizeof(*new), DefMpool);

            /* push the new element to the front of the list */
            new ->next = plist;
            plist = new;

            /* copy the new string into its place */
            new->name = savestr(inbuf, DefMpool);
    }

    exit(RS_NORM);
}
```

2.3.4. Order By Clauses

In interactive IDL it is possible to sort the results of a **retrieve** command using an **order by** clause with the name of the domain in which the retrieved data is stored as in

retrieve (last = a.last) order by last

and

retrieve (name = concat(a.first, a.last)) order by name

In RIC, the object on which retrieved data is ordered may not be the domain in which retrieved data is stored. The RIC code required to achieve the functionality of the commands above is

```
$retrieve ($last = a.last) order by a.last;
```

and

```
$retrieve ($name = concat(a.first, a.last))
      order by concat(a.first, a.last);
```

2.3.5. Nested Retrieves

It is possible to nest an IDL **retrieve** command inside of another IDL **retrieve** command.

```
/*
**  NEST.RIC
**
**        Displays all the books in the "title" relation and
**        for each book displays all its authors.
**        Demonstrates nested retrieves.
*/

main()
{
   $char  name[16];
   $char  title[36];
   $short docnum;

   $range of t is title;
   $range of a is author;
   $range of l is authttl;

   INITRIC("nest");

   /* outer loop retrieves and displays titles */
   $retrieve (t.title, t.docnum)
   {
          printf("%s\n", title);

          /* inner loop retrieves and displays authors for each title */
          $retrieve ($name = a.last)
                 where a.authnum = l.authnum and $docnum = l.docnum
          {
                 printf("\t%s\n", name);
          }
          printf("\n");
   }

   exit(RS_NORM);
}
```

The issue of nesting other types of queries, such as updates, inside a **retrieve** loop is discussed in the following chapter in Section 3.2.3.

2.4. Canceling IDL Commands

At times it may be desirable to cancel all IDL activity on the database server. To do this, use the special IDL command, **cancel** prefaced by a dollar sign. This command is only available in embedded languages; it does not exist in interactive IDL. The **cancel** command cancels all database server activity on the current dbin and any related dbins. If the **cancel** occurs inside a **retrieve** loop, the C **break** command must be used to exit the loop. If the **retrieve** loop is nested, there must be an explicit **break** command for every level of nesting outside of the loop in which the **cancel** command occurs.

The previous example, *nest.ric*, could be re-written as follows to cancel all database server activity if an invalid author name is retrieved.

```
/*
**  CANCEL.RIC
**
**        Displays all the books in the "title" relation and
**        for each book displays all its authors.
**        If an author name not beginning with an alphabetic character
**        is retrieved, cancels all database activity and breaks from
**        the inner and outer retrieve loops. Demonstrates $cancel.
*/

#define   NOTALPHA      !(c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z')

main()
{
   $char  name[16];
   $char  title[36];
   $short docnum;
   char   c;
   BOOL   cancelled = FALSE;

   $range of t is title;
   $range of a is author;
   $range of l is authttl;

   INITRIC("cancel");

   /* outer loop retrieves and displays titles */
   $retrieve (t.title, t.docnum)
   {
          printf("%s\n", title);

          /* inner loop retrieves and displays authors for each title */
          $retrieve ($name = a.last)
                 where a.authnum = l.authnum and $docnum = l.docnum
          {
                 c = name[0];
                 if (NOTALPHA)
                 {
                        cancelled = TRUE;
                        $cancel;      /* both retrieves */
                        break;        /* from inner loop */
                 }
                 else
                        printf("\t%s\n", name);
          }
          if (cancelled)
                 break;        /* from outer loop */
          else
                 printf("\n");
   }

   exit(RS_NORM);
}
```

3. Advanced Programming with RIC

This chapter describes methods for making RIC programs more powerful

- by accessing IDL's stored commands.
- by taking advantage of IDL's support for transactions
- by invoking IDMLIB's exception handling system for error conditions

Transactions are described in the *IDL Reference Manual* pages for **begin transaction**, **end transaction**, and **abort transaction**. Stored commands are described under **define** and **execute**.

In addition to the schema used for the examples in Chapter 2, the examples in this chapter assume the existence of the following stored commands in the "books" database:

```
define addtitle
        range of t is title
        append to title
        (
                docnum = max(t.docnum) + 1,
                title = $t,
                onhand = $q
        )
end define
```

```
define newtitle
        range of t is title
        append to title
        (
                docnum = max(t.docnum) + 1,
                title = $t,
                onhand = $q
        )
        retrieve (t.docnum, t.title, t.onhand)
                where t.docnum = max(t.docnum)
end define
```

```
define newauth
        range of a is author
        append to author
        (
                authnum = max(a.authnum) + 1,
                first = $f,
                last = $l
        )
        retrieve (a.authnum, a.first, a.last)
                where a.authnum = max(a.authnum)
end define
```

```
define addlink
        append to authttl
        (
                authnum = $a,
                docnum = $d
        )
end define
```

```
define getstuff
        range of t is title
        retrieve (t.docnum, t.onhand)
        range of a is author
        retrieve (a.first, a.last)
end define
```

3.1. Stored Commands

Stored commands may be defined and executed from a RIC program. Parameters passed to the stored command may be literal values or C identifiers flagged with dollar signs. The following program executes the stored command "addtitle".

```
/*
**  STORED.RIC
**
**  This program appends a tuple to the "title" relation
**  using the stored command "addtitle". Parameters passed
**  to the stored command are C identifiers fetched from the
**  command-line with crackargv().
*/

#include <crackargv.h>

$static  short  Quan;
$static  char   *Title;

/* the argument list */
ARGLIST  Args[] =
{
  't', FLAGSTRING, 0, CHARNULL, CHARNULL, __ &Title, "Title: ", CHARNULL,
  'q', FLAGSHORT, 0, CHARNULL, CHARNULL, __ &Quan, "Quantity: ", CHARNULL,
  '\0'
};

main(argc, argv)
    int  argc;
    char *argv[];
{
    INITRIC("stored");
    crackargv(argv, Args);

    $execute addtitle with q = $Quan, t = $Title;
    exit(RS_NORM);
}
```

3.1.1. Obtaining Retrieved Data from Stored Commands

If the stored command contains one or more **retrieve** commands, the **execute** command must have a block structure delineated by curly braces ({ }). The statements between the curly braces consist of one or more special **obtain** commands used to bind retrieved data to C program variables. This is a special command which is only available in embedded IDL; there is no **obtain** command in interactive IDL. The only kind of $-flagged statements allowed within the **execute** block are **obtain statements**.

There must be one **obtain** command flagged with a dollar sign for **each retrieve** command in the stored command.

The syntax of an **obtain** command is

> **obtain** (expr [,expr. . .])

Each element in the comma-separated list of C expressions must evaluate to an object of a type compatible with the targets returned by the **retrieve.** The table in Section 1.2 maps database server types to C types.

The first attribute returned by the **retrieve** command is assigned to the object indicated by the first expression passed to **obtain**; the second attribute returned by the **retrieve** is assigned to the object indicated by the second expression passed to **obtain**, etc.

The **obtain** command is terminated either by a pair of curly braces ({ }), which may enclose a list of executable statements processing the retrieved data, or by a semicolon (;), if the only processing of retrieved data is assignment to the designated C identifiers. If the **obtain** command is terminated by a semicolon (;), the command still loops for every tuple retrieved; it does not simply fetch the first tuple as in a singleton **retrieve**.

When the curly braces are used, the executable statements in the **obtain** block must be C statements, not $-flagged IDL statements. The precompiler checks for $-flagged IDL statements in the body of the **obtain** loop, but it is the programmer's responsibility to make certain that no function called from the **obtain** loop contains any IDL statements. This code elicits an error message at precompile time

```
$obtain ($num, $name)
{
        if (num = badnum)
              $delete r where r.num = $badnum;
        else
              printf("%d, %s", num, name);
}
```

but the following code elicits no error message, although it will cause unpredictable behavior at runtime.

```
$obtain ($num, $name)
{
        if (num = badnum)
                func(badnum);
        else
                printf("%d, %s", num, name);
}

func(badnum)
$int   badnum;
{
        $delete r where r.num = $badnum;
        return()
}
```

The following program uses the **obtain** command to bind data retrieved from the stored command "getstuff" into arrays of shorts and character strings. Because the counters are incremented on the final pass through the obtain loop for which all of the tuples have already been obtained, the counters must be decremented at the end of the loop if their values are to be used subsequently in the program.

```
/*
**  GETDATA.RIC
**
**  This program executes the stored command "getstuff".
*/

#define MAXITEM  15
#define MAXNAME  20

main()
{
    $short docnums[MAXITEM], quantities[MAXITEM];
    $char  fnames[MAXITEM][MAXNAME], lnames[MAXITEM][MAXNAME];
    int    i,j,k;

    INITRIC("getdata");
    i = j = 0;

    $execute getstuff
    {
            /* fill the arrays */
            $obtain($(docnums[i]), $(quantities[i++]));
            $obtain($(fnames[j]), $(lnames[j++]));
    }
    /* decrement the counters */
    j--; i--;

    /* print the arrays */
    printf("\ndocnum\tquantity");
    for (k = 0; k < i; k++)
            printf("\n%d\t%d", docnums[k], quantities[k]);

    printf("\nauthors");
    for (k = 0; k < j; k++)
            printf("\n%s\t%s", fnames[k], lnames[k]);

    exit(RS_NORM);
}
```

The next program uses the **obtain** command to bind data retrieved by the stored commands "newtitle" and "newauthor". The **obtain** command is followed by two **printf** statements enclosed in curly braces ({ }). These statements are executed one time for each tuple obtained.

```
/*
**  NEWBOOK.RIC
**
**          This program enters new books into the "books" database.
**
**          Retrieves the title from the "title" relation.
**          If it is there, adjusts "onhand" attribute.
**          If it is not there, adds appropriate tuple to the "title",
**          "author", and "authttl" relations, using the stored commands
**          "newtitle", "newauth", and "addlink".
**
**          Demonstrates use of the obtain command to execute stored
**          commands containing retrieves from a RIC program.
*/

main()
{
    $char  title[36];
    $char  lname[16];
    $char  fname[11];
    $short onhand;
    $short docnum;
    $short authnum;
    $char  newname[36];
    $short newquan;
    char   buf[5];


    INITRIC("newbook");
    $range of t is title;
    $range of a is author;

    getprompt(newname, sizeof(newname), "Enter title:");
    getprompt(buf, sizeof(buf), "Enter quantity: ");
    newquan = (short) atos(buf);

    /* retrieve the docnum from the "title" relation */
    $retrieve (t.docnum)
          where t.title = $newname;

    /* if the title is already in the database */
    if (docnum != '\0')
    {
          $replace t (onhand = t.onhand + $newquan)
                where t.docnum = $docnum;
          $retrieve (t.docnum, t.title, t.onhand)
                where t.title = $newname;

          printf("The new tuple is:\n%d\t%s\t%d\n", docnum, title, onhand);
          exit(RS_NORM);
    }
    else
    /* add the book to the database */
    {
          /* invoke stored command "newtitle" */
          $execute newtitle with q = $newquan, t = $newname
          {
                /* bind retrieved data to designated C identifiers */
                $obtain($docnum, $title, $onhand)
```

```
                    {
                            printf("\nThe new row is:\n");
                            printf("%d\t%s\t%d", docnum, title, onhand);
                    }
            }

            /* retrieve the author from the "author" relation */
            getprompt(fname, sizeof(fname), "Enter author's first name: ");
            getprompt(lname, sizeof(lname), "Enter author's last name: ");
            $retrieve (a.authnum)
                    where a.first = $fname and a.last = $lname;

            /* if the author is not in the database */
            if (authnum == '\0')
            {
                    /* invoke stored command "newauth" */
                    $execute newauth with f = $fname, l = $lname
                    {
                            /* bind the new author number with authnum) */
                            $obtain($authnum, $fname, $lname)
                            {
                                    printf("\nThe new row is:\n");
                                    printf("%d\t%s\t%d", docnum, title, onhand");
                            }
                    }
            }

            /*
            **  add a tuple to the "authttl" relation  using stored
            **  command "addlink"
            */
            $execute addlink with a = $authnum, d = $docnum;
    }

    exit(RS_NORM);
}
```

3.2. Transactions

A transaction is a sequence of one or more IDL commands which are executed as though they were a single command. Transactions are used to ensure consistency in a database.

None of the commands comprising the transaction may alter the schema of the database. An attempt to execute a command such as **create** or **destroy** inside a transaction will cause an exception to be raised. The only commands which can be used in a transaction in a RIC program are:

- **abort transaction**
- **append**
- **begin transaction**
- **delete**
- **end transaction**
- **replace**
- **retrieve**
- **sync**

The **begin transaction** and **end transaction** commands which delineate a transaction must pair up within a function. A **begin transaction** with no corresponding **end transaction** in the same function will elicit an error message from the precompiler.

C code as well as IDL queries may be inserted between the **begin transaction** and the **end transaction** in a RIC program.

The following program demonstrates the use of a simple transaction in a RIC program. Tuples are appended to three relations as a new book is entered in the database. If a transaction were not used in an application such as this and the system went down in the middle of execution, for example after the title was entered but before the author, the entry for that book would be incomplete. The use of a transaction ensures that entries are made in all three relations or not at all.

```
/*
** TRANS.RIC
**
**       This program enters a new book into the "books" database.
**
**       Adds a tuple to the "title" and "authttl" relations and may
**       add a tuple to the "author" relation.
**
**       Demonstrates RIC support for transactions.
**       Aborts if user tries to add a duplicate title.
**       Gets title and author from the command-line using the IDMLIB
**       function, crackargv.
*/

#include <crackargv.h>

   /* variables for command-line arguments */
   $short Quan;
   $char  *Title;
   $char  *Fname;
   $char  *Lname;

   /* the argument list */
   ARGLIST       Args[] =
   {
          FLAGPOS, FLAGSHORT, 0, CHARNULL, CHARNULL, __ &Quan,
                 "Quantity:", CHARNULL,
          FLAGPOS, FLAGSTRING, 0, CHARNULL, CHARNULL, __ &Title,
                 "Title:", CHARNULL,
          FLAGPOS, FLAGSTRING, 0, CHARNULL, CHARNULL, __ &Fname,
                 "Author's first name:", CHARNULL,
          FLAGPOS, FLAGSTRING, 0, CHARNULL, CHARNULL, __ &Lname,
                 "Author's last name:", CHARNULL,
          '\0'
    };

main(argc, argv)
   int    argc;
   char   **argv;
{
   $short authnum;
   $short docnum;

   $range of t is title; $range of a is author; $range of l is authttl;

   INITRIC("trans");

   /* get values from command-line for Quantity, Title, Fname, Lname */
   crackargv(argv, Args);

   $begin transaction;
   $append to title
   (
          docnum = max(t.docnum) + 1,
          title = $Title,
          onhand = $Quan
   );

   $retrieve ($docnum = t.docnum)
```

```
           where t.title = $Title
    {
           ;
    }
    /* check for duplicates */
    if (rccount() > 1)
    {
           printf("\nThat title is already in the database.");
           printf("\nAborting this transaction");
           /* control passes to statement after "end transaction" */
           $abort transaction;
    }

    /* check if the author is in the "author" relation */
    $retrieve ($authnum = a.authnum)
           where a.first = $Fname and a.last = $Lname;

    /* if the author is not in the relation */
    if (authnum == '\0')
    {
           $append to author
           (
                  authnum = max(a.authnum) + 1,
                  first = $Fname,
                  last = $Lname
           );
           $retrieve (a.authnum)
                  where a.authnum = max(a.authnum);
    }

    $append to authttl
    (
           authnum = $authnum,
           docnum = $docnum
    );

    $end transaction;
    exit(RS_NORM);
}
```

3.2.1. Nested Transactions

A transaction is nested if a **begin transaction** is executed inside of a transaction. Nested transactions can occur inside a single function or across function calls. In the outline below, the transaction in g() is nested when g() is called by f(), but not when g() is called by e().

```
f()
{
        $begin transaction;
        /* stuff */
        g();
        $end transaction;
        return();
}

e()
{
        /* stuff */
        g();
        return();
}

g()
{
        $begin transaction;
        /* IDL queries */
        $end transaction;

        /* more stuff */
        return();
}
```

Since the database server does not commit the transaction until the **end transaction** corresponding to the first **begin transaction** is executed, which in this case is the **end transaction** in f(), the transaction in g() does not have any real meaning when g() is called by f().

When f() calls g(), an **abort transaction** in g() transfers control to the statement following the **end transaction** in f(). When e(), calls g(), an **abort transaction** in g() transfers control to the statement following the **end transaction** in g().

The following program demonstrates nested transactions.

```
/*
** NESTTRANS.RIC
**
**          This program displays all the books in the "title" relation
**          and for each book displays all its authors.
**
**          Demonstrates nested transactions.
**          Functionally like NEST.RIC, except commands are transactions.
**          If a title is found with no associated author,
**          an error message is displayed and all processing is
**          halted by the "abort transaction".
*/

main()
{
    $char  title[36];
    $short docnum;
    void   getauths();

    $range of t is title;

    INITRIC("nesttrans");

    /* outer loop to retrieve and display titles */
    $begin transaction;

    $retrieve (t.title, t.docnum) order by t.docnum
    {
            printf("%s\n", title);
            getauths(docnum, title);
            printf("\n");
    }

    /*
    ** commit everything from the above retrieve
    ** and the one in getauths
    */

    $end transaction;

    exit(RS_NORM);
}

/*
**  GETAUTHS -- retrieve and display the authors of the title
**  passed in docnum.
*/

void
getauths(docnum, title)
    $short docnum;
    $char  *title;
{
    $char  name[16];

    $range of a is author;
    $range of l is authttl;

    /* this is a nested transaction when called from main */
60     $begin transaction;
61     $retrieve ($name = a.last)
62             where l.authnum = a.authnum and l.docnum = $docnum
63     {
64             printf("\t%s\n", name);
65     }
66     /* if the name is not there */
67     if (!rccount())
68     {
69             printf("\nNo entry for an author of %s.", title);
70             printf("\nAborting this transaction so user can");
71             printf("\nmodify the 'author' and/or 'authttl' relations.");
72
73             /* transfers control to after "end transaction" in main */
74             $abort transaction;
75     }
76
77     /* other IDL commands could go here. */
79     $end transaction;
80  }
```

3.2.2. New Transactions

In the program above, all processing halts after execution of the **abort transaction** on line 74 because the inner transaction in *getauths()* is nested in the outer transaction in *main()*. If the transaction aborts, control passes to the statement following the **end transaction** in *main()*, which is **exit**().

It may be desirable for some applications to ensure that a **begin transaction** command initiates a new transaction rather than a nested one. To accomplish this, use the command **begin new transaction**.

To modify *nesttrans.ric* so that a new transaction is initiated in *getauths()* change the **begin transaction** on line 60 to **begin new transaction**. The result of this modification is that the transaction in *getauths()* is no longer nested in the transaction in *main*. When the **abort transaction** in *getauths()* is executed, control passes to the statement following the **end transaction** in *getauths()* rather than the one in *main()*. In this case, the function returns to line 31 in *main()* and the program continues processing.

Consider the outline

```
f(x)
        BOOL    x;
{
        BOOL    y;

        $begin transaction;
        y = TRUE;
        g(y);

        if (x)
                $abort transaction;

        /* this transaction is committed here */
        $end transaction;
        return(TRUE);
}

g(y)
        BOOL    y;
{
        $begin new transaction;
        if (y)
                $abort transaction;

        /* this transaction is committed here */
        $end transaction;

        /* control transfers to here on an abort transaction */
        return(TRUE);
}
```

An **abort transaction** in f() will not abort anything that was done in g(), because the transactions in f() and g() are unrelated as far as the database server is concerned. It is as though the two transactions were invoked by two separate programs. Unrelated transactions must be used with caution, however, because they can lead to self-inflicted infinite waits as illustrated below:

```
$range of t is title with fulllock;

f()
{
        $begin transaction;

        /* this replace locks the whole relation */
        $replace t (title = "untitled");
        g();
        $end transaction;
}

g()
{
        $begin new transaction;
        $replace t (title = "UNIX SYSTEMS")
              where t.title = "ARCHAIC OS THEORY";
        $end transaction;
}
```

Since the transaction in g() needs to access some data which the transaction in f() has locked, g() must wait for the transaction in f() to complete. Function f() however, is waiting for g() to return before committing its transaction and releasing its locks. Because the database server cannot detect the dependency between f() and g(), the deadlock is not signaled and the program goes into an infinite wait.

3.2.3. Nested Operations in Retrieve Loops

Special issues arise when a program performs an update inside a **retrieve** loop. The update cannot simply be nested in the retrieve loop as in *nesttrans.ric* because of the way retrieve loops are implemented; when two operations are so closely related (running in the same dbin), the IDMLIB software expects all the data from the database server to be fetched before initiating a new database query. On the other hand, if the update is begun in a new transaction, as described in section 3.2.2, the retrieve transaction and the update transaction are completely unrelated; if a deadlock occurs in this situation, there is no way for the database server to manage a backout because it does not know which operations are related. What is needed is the establishment of a relationship between the nested transactions, so that communication between them is possible.

This is accomplished with the **begin nest *n* transaction** command where *n* represents the depth of nesting up to a maximum of 7. The outline below represents two levels of nesting inside an outer retrieve loop:

```
func()
{
        $begin nest 2 transaction;

        /* outer retrieve loop */
        $retrieve . . .
        {
                /* inner retrieve loop - first level of nesting */
                $retrieve . . .
                {

                        /* update - second level of nesting */
                        $replace . . .
                }
        }

        $end transaction;
}
```

The next program, *reexam.ric*, demonstrates the use of a single level of nesting to execute a **replace** command inside a **retrieve** loop. As each tuple in the "titles" relation is retrieved, the user is prompted to update the value of the "onhand" attribute in that tuple.

3.3. Deadlock Backout and Recovery

If several applications are simultaneously accessing and updating the same data, deadlock can occur. Deadlock can also occur within a single application which uses the **begin new transaction** or **begin nest *n* transaction** constructions.

When the database server detects that a deadlock has occurred, it selects one or more participants, backs out all the work already done by their queries, and signals the host that a backout is occurring by raising two exceptions: "E:IDM:E55" and "W:IDM.DONE.XABORT".

By default, RIC generates code which catches the exceptions and attempts to restart the query immediately.

If you wish your RIC program to take some alternate or additional action when a deadlock/backout is detected, the program should set a handler to ignore "E:IDM:E55" and another to manage "W:IDM.DONE.XABORT". For more information about setting an exception handler, consult Chapter 3 of the *Idmlib User's Guide.*

If the query being backed out is a transaction, the exception handler must be set after the **begin transaction** and removed after the **end transaction**. If **new** transactions are used, the exception handler must be reset after each **begin new transaction** and removed after each **end transaction**.

The next program sets an exception handler called *handler()* which displays a message and re-raises the exception.

```
/*
**  REEXAM.RIC -- runs an update inside a retrieve loop.
**
**        Retrieves each tuple in the "title" relation.
**        Allows the user to update the "onhand" attribute of
**        the retrieved tuple.
**
**        Demonstrates "begin nest <n> transaction".
**        Provides one level of nesting - one parent and one child.
**        The "begin nest <n> transaction" is necessary when there is
**        an update inside a retrieve loop.
**
**        Sets an exception handler to display a message and re-raise
**        the exception to restart the transaction when W:IDM.DONE.XABORT
**        is raised. Uses canned handler excignore for E:IDM:E55.
*/

main()
{
   $char  title[36];
   $short docnum;
   $short onhand;
   $short newstock;
   char   buf[5];
   int    handler();

   $range of t is title;

   INITRIC("reexam");
   $begin nest 1 transaction;

   /* set exception handlers for deadlocks */
   exchandle("E:IDM.E55", excignore);
   exchandle("W:IDM.DONE.XABORT", handler);
   printf("\nDOCNUM     TITLE         ONHAND\n");

   /* run retrieve on parent dbin */
   $retrieve (t.docnum, t.title, t.onhand)
   {
          printf("%d\t%s\t%d\n", docnum, title, onhand);

          /* get input for the update */
          getprompt(buf, sizeof(buf),
                 "Enter new stock or <RETURN> to quit: ");
          if (buf[0] == '\0')
                 break;
          newstock = atos(buf);

          if (newstock == 0)
                 printf("\nNo change in # copies of %s.\n", title);
          else if (newstock > 0)
                 printf("\nAdding %d copies of %s.\n", newstock, title);
          else
                 printf("\nSubtracting %d copies of %s.\n",
                        newstock * -1,  title);

```

```
            /* run replace on nested child dbin */
            $replace t (onhand = t.onhand + $newstock)
                   where t.docnum = $docnum and t.title = $title;
    }

    $end transaction;

    /* remove the exception handlers */
    exchandle("E:IDM.E55", FUNCNULL);
    exchandle("W:IDM.DONE.XABORT", FUNCNULL);

    exit (RS_NORM);
}

/*
**  HANDLER
**
**          Displays message before attempting to restart the query.
*/

handler(excv)
    char **excv;
{

    fprintf(stderr, "\nYour query has deadlocked with another");
    fprintf(stderr, "application and your work is being backed out.");
    fprintf(stderr, "\nAn attempt is being made to restart the query.");

    /* call RIC's exception handler to attempt restart */
    excvraise(excv);
    return (0);
}
```

You may wish for a RIC program to catch all exceptions, not just "E:IDM.55", and "W:IDM.DONE.XABORT". The following statement declares *myhandler()* to handle all exceptions of severity Error.

```
exchandle("E:*", myhandler);
```

Appendix A: Stored Programs

All of the query trees which RIC builds from IDL code are normally stored in the executable program which is created when the output of RIC is compiled. When the program is executed, the trees are sent to the database server for processing.

The trees representing the database queries may instead be stored on the database server as a stored program.

$$\begin{tikzcd} \textit{prog.ric} \arrow[r, "\text{RIC precompiler}"] \arrow[d, "\text{store program}"'] & \textit{prog.c} \arrow[r, "\text{C compiler}"] & \textit{prog} \\ \text{myqueries} & & \end{tikzcd}$$

Queries may be stored on the database server only if the RIC program satisfies the following requirements:

- The program operates on a single database server in a single database which can be specified at precompilation time.
- The schema of the database is unchanging, meaning that objects in the database are not created, destroyed or otherwise altered structurally during execution of the program.
- The program uses no dynamic range variables.

When a program is in the development stage, it is generally desirable not to store the trees on the database server because precompilation time is longer, since the precompiler must communicate with the database server. But when a production version of a RIC program is being precompiled, it is often preferable to store the queries on the database server, because execution may be faster and the size of the executable program is smaller.

To precompile a RIC program so that the trees are stored on the database server, use the **–n** or **/progname** *progname* option. The *progname* is the unique name under which the collection of database queries in that program are to be stored.

When the **–n** or **/progname** *progname* option is used, the **–d** or **/dbname** *database* option must also be used.

The database server destroys any stored programs in the specified database previously stored under ***progname*** before precompiling the source file and creating the stored program.

The following command-line instructs RIC to precompile *myprog.ric*, storing the query trees on the database server as a stored program in the "books" database named "myqueries".

```
ric –n "myqueries" –d "books"  myprog.ric (Unix)

ric /progname = myqueries  /dbname = books myprog.ric        (other)
```

If a RIC program contains several modules residing in different source files, the stored programs for each module should be associated with a unique name. The following command-lines instruct RIC to precompile a program residing in two source files, and to store the queries in *main.ric* in a stored program called "mainqueries" and the queries in *functions.ric* in a stored program called "funcqueries". Both "mainqueries" and "funcqueries" are stored in the "books" database:

UNIX:

```
ric –n "mainqueries" –d "books"  main.ric
ric –n "funcqueries" –d "books"  functions.ric
cc –o prog main.c functions.c –lidmlib
```

VMS:

```
ric /progname=mainqueries  /dbname=books main.ric
ric /progname=funcqueries  /dbname=books functions.ric
DEFINE/USER VAXC$INCLUDE IDM_DIR
CC main, functions
LINK/EXE=PROG.EXE MAIN,FUNCTIONS,IDMLIB/OPT
```

PC/MS-DOS

```
ric /progname=mainqueries  /dbname=books main.ric
ric /progname=funcqueries  /dbname=books functions.ric
msc /AL /Gs main.c, main.obj;
msc /AL /Gs functions.c functions.obj;
link /STACK:10000 main.obj + functions.obj, prog,, idmlib;
```

AOS/VS

```
ric  –n "mainqueries"  –d "books"  main.ric
ric  –n "funcqueries"  –d "books"  functions.ric
cc main :IDM:include/search
cc functions :IDM:include/search
ccl/o=prog/tasks=4 main.ob functions.ob :IDM:LIB:rclib.lb &
       :IDM:LIB:IDMLIB.lb :IDM:LIB:ITPUSR.lb
```

Appendix B: RIC and the C Preprocessor

The C preprocessor is the first pass of the C compiler. It processes lines beginning with a score (#), such as *#define*, *#include*, and *#line*. On some operating systems, including many flavors of Unix, it is possible to invoke the C compiler's preprocessor separately from other passes of the compiler. It may be desirable to run the preprocessor on a RIC source file before sending it through RIC to obtain **struct** definitions and **typedefs** given in a header file so that these definitions may be applied to C variables which are known to RIC.

This technique should be used with caution, though many users can simply use the most straightforward pipeline their Unix systems and shell allow. Most C preprocessors require their input files to have a suffix **.c** and will not preprocess a file with the suffix **.ric** or no suffix.

Flagging Statements for RIC Header Files

A special problem arises when a RIC source file is run through the C preprocessor before being run through RIC, if a header file is to be *#included* by both the RIC programs and pure C programs. For RIC, declarations in the header file must be prefaced with a dollar sign ($) so they will be noticed by RIC, but if declarations in a C program are prefaced with a dollar sign ($), they will elicit syntax errors from the C compiler.

To overcome these contradictory requirements, all header files which must be acceptable to RIC and the C compiler should begin with the line

```
#include <rcflag.h>
```

In addition, any declarations in the header file which may need to be interpreted by RIC should be flagged with the word "RCFLAG" instead of a dollar sign ($).

For example, the following header file called *example.h*

```
/*
**  EXAMPLE.H - header file to be used by RIC and a C compiler.
*/

#include <rcflag.h>

RCFLAG    int    Num;
```

would be included in the RIC source file by the following two lines

```
#define      RCFLAG  $
#include     "example.h"
```

Now *Num* will be flagged with a dollar sign ($) in the source file processed by RIC but not in any other files.

The #line Directive

A directive that reads

```
#line  3 "file.ric"
```

instructs the C compiler to consider the next line of text to be the third line read from a file named *file.ric*, regardless of the name of the file it is actually reading or the number of lines it has actually read. RIC writes many of these directives on the output file it produces to allow error messages from the C compiler to reference lines in the RIC source. RIC also reads and interprets any *#line* directives in its source file and uses them to formulate its conception of how lines in the input file are to be referred to in error messages.

Unfortunately, there is a lack of unanimity among C compilers, and even among various phases of a single C compiler, concerning the precise syntax of *#line* directives. RIC reads and interprets all the following forms identically. Any non-NULL string of blanks or tabs can be substituted for the blanks in these examples.

(1) # line 3 "file.ric"

(2) # line 3 "file.ric"

(3) # 3 "file.ric"

(4) # line 3 file.ric

(5) # line 3 file.ric

(6) # 3 file.ric

By default, any *#line* directive emitted by RIC ressembles example 1 above. This includes directives read from the source file, which RIC always interprets and rewrites. This is the form preferred by most C compilers.

If the –l command-line option is passed to RIC the word *line* does not appear in the output directive, producing a line ressembling example 3. This is the form demanded by the parsing phase of the Unix compiler *cc*. On some systems it is possible to pass RIC output which was run through the C preprocessor when it was RIC source directly to the parsing phase of *cc*, avoiding a second useless pass through the preprocessor. This feature is not always documented.

If the **–q** command-line option to RIC, quotation marks will not enclose the filename in the *#line* directives. This is required by some C compilers.

Appendix C: Portability of RIC Programs

Without Stored Programs

The pure C code produced by the precompiler is not totally portable between different types of hardware. This is because, when a RIC program is precompiled, each IDL query is represented as a series of octal bytes (see the variable *utrees* in the output file shown in section 1.1.3). These bytes are defined in the C program in the order in which they appear in storage on the host machine on which the program is precompiled. Some machines store integers most-significant byte first; others store them least-significant byte first. Thus, different *utrees* are created on different hardware.

The implications of this for portability are as follows:

- If a RIC program which has been precompiled on a host that represents integers most-significant byte first is being ported to a host that represents integers least-significant byte first, the RIC source must be precompiled again on the destination host.
- If a RIC program which has been precompiled on a host that represents integers least-significant byte first is being ported to a host that represents integers most-significant byte first, the RIC source must be precompiled again on the destination host.
- If the original and destination hosts represent integers in the same order, it is not necessary to re-precompile the RIC source.

With Stored Programs

If all of the IDL queries are compiled into stored programs, the above problem is avoided. In this case, both the stored program(s) and the C code produced by the precompiler are machine-independent and a new precompilation is not necessary when the RIC program is ported.

However, in order for all of the IDL queries to be compiled as stored programs, the following conditions must exist:

(1) The RIC program must have been precompiled with the **–n** or **/progname** option.

(2) No IDL statement may contain a C expression as the first argument to the functions *bcd(), bcdflt(), bcdflt(), fbcdflt(), string(), fstring(), fchar(),* or *char().* No IDL statement may contain a C expression as either of the first two arguments to the functions *bcdfixed(), substr()* or *substring().*

(3) The IDL statements may only be among the following: **range**, **retrieve**, **append**, **replace**, **delete**, **begin transaction**, **end transaction**, or **abort transaction**.

(4) No IDL statement may use dynamic range variables.

If conditions 2, 3, and 4 do not exist, but the RIC program is compiled with the **-n** or **/progname** option, the precompiler will create stored programs for the allowable IDL statements and machine-dependent *utrees* for the rest. The user is not notified when this occurs; the only way to ascertain that both *utrees* and stored programs have been created is to examine the pure C output from the precompiler.

With Stored Programs and Utrees

If precompilation of a RIC program creates a combination of stored programs and machine-dependent C code, and the program is to be ported, the machine-dependent portion must be precompiled according to the guidelines mentioned above in the section dealing with RIC programs which do not contain stored programs. If a new precompilation is necessary, the programmer may choose to handle it in one of two ways:

(1) Precompile the RIC program on the destination machine using the **-n** or **/progname** option, but give the argument to this option a different name for every host on which the program is precompiled. For example, the following command precompiles a program on a VAX running Unix:

```
ric -n "VAXmyqueries" -d "books" myprog.ric
```

To port the program to an IBM PC, one could transfer the source to a PC and precompile it as follows:

```
ric /progname = "PCmyqueries" /dbname = books myprog.ric
```

This is necessary because, if the name of the stored program is not changed, the original stored program from the original precompilation will be overwritten and given a new internal identifier. This will make it impossible for the original RIC program to retrieve its stored program.

This solution causes identical stored programs to be stored on the database server under different internal identification numbers.

(2) Where feasible, a preferable solution is to write the code in such a way that all queries which become stored programs, that is, those using the commands **range**, **retrieve**, **append**, **replace**, **delete**, **begin transaction**, **end transaction**, or **abort transaction**, and not using C expressions as arguments in the functions listed above, are contained in one precompiler source file. The IDL commands which produce machine-dependent code go in a second source file. The first file would be precompiled once on the original host and then its output transferred to all the destination hosts where it would be compiled. The second file would be precompiled and compiled on all of the destination hosts. This solution results in a single stored program on the database server with individual machine-dependent modules on the various hosts.

Index of Terms