Britton Lee Host Software

RSC USER'S GUIDE

(R3v5)

March 1988

Part Number 205-1575-003

This document supersedes all previous documents of the same title. 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 responsiblity 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.

IDM, Intelligent Database Language and IDL are trademarks of Britton Lee, Inc.

Unix is a trademark of AT&T Bell Laboratories.

VAX and VMS are trademarks of Digital Equipment Corporation.

MS-DOS is a trademark of Microsoft Corporation.

AOS/VS is a trademark of the Data General Corporation.

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 SQL.

Britton Lee has developed a precompiler called RSC, which handles SQL queries embedded in C. RSC translates program statements written in a mixture of SQL 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 RSC" contains a general description of RSC and covers general information for anyone writing programs in RSC.

"Programming with RSC" describes in detail how SQL commands are embedded in C code, how C expressions may be used in SQL commands, and how SQL **select** queries are handled in RSC programs.

"Advanced Programming With RSC" describes RSC support for stored commands and transactions. It also contains some examples of RSC 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 RSC source code.

1. Introduction to RSC

RSC is a precompiler which accepts a file with SQL 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.

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

The prerequisites for using RSC are a solid knowledge of the C programming language and some knowledge of the Structured Query Language (SQL). The requirement for a solid knowledge of C cannot be overemphasized. It is not feasible to learn C and RSC 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 RSC 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 SQL commands precompiled by RSC and "output file" refers to the pure C file produced by RSC to be compiled by a C compiler. The word "RSC" refers both to the precompiler and to the embedded language which it precompiles.

1.1.1. Input to RSC

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

```
1  /*
2  ** SIMPLE.RSC
3  **   This program inserts a row into "myrelation".
4  */
5
6  main()
7  {
8      INITRSC("simple");
9
10     $insert into myrelation
11            (num, name)
12            values (1, 'agatha');
13
14     exit(RS_NORM);
15 }
```

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

- The RSC source filename must have the suffix **rsc,** or no suffix. We could name the source file *simple.rsc* or *simplesrc* but not *simple.src.*
- All RSC programs must call INITRSC with the name of the executable object as its argument. INITRSC initializes IDMLIB and the RSC runtime library environment. INITRSC may be called only once by a RSC program.
- The program contains at least one embedded SQL statement introduced by a dollar sign ($). RSC would precompile a pure C program without any embedded SQL statements, but there would be little point to this exercise, since the purpose of using RSC is to write programs which contain SQL 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.rsc* with the command

rsc –d "mydatabase" simple.rsc (Unix)

rsc /dbname=mydatabase simple.rsc (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 INITRSC, but they must appear before the first executable SQL command. For example, we rewrite the program above in Section 1.1.1 as

```
1  /*
2  ** SIMPLE2.RSC -- this program inserts a row to "myrelation".
3  **
4  */
5  main()
6  {
7      RCDEVICE("host%xns");
8      RCDBNAME("mydatabase");
9      INITRSC("simple");
10
11     $insert into myrelation
12     (num, name)
13     values (1, 'agatha');
14
15     exit(RS_NORM);
16 }
```

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*.

```
1  /*
2   ** SIMPLE3.RSC -- this program inserts a row into "myrelation".
3   **
4   **        Gets database name from the command-line using crackargv.
5   **        Device name is the default in environmental variable IDMDEV.
6   */
7
8   #include  <crackargv.h>
9
10  char      *Dbase;
11
12  /* the argument list */
13  ARGLIST   Args[] =
14  {
15     FLAGPOS, FLAGSTRING, 0, CHARNULL, CHARNULL, __ &Dbase,
16     "Enter database:", CHARNULL,
17     '\0'
18  };
19
20  main(argc, argv)
21     int    argc;
22     char   *argv[];
23  {
24     INITRSC("simple");
25     crackargv(argv, Args);
26     RCDBNAME(Dbase);
27
28     $insert into myrelation
29     (num, name)
30     values (1, 'agatha');
31
32     exit(RS_NORM);
33  }
```

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

```
rsc    simple3.rsc
```

1.1.3. Output from RSC

RSC's output from *simple.rsc* 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.rsc"
/*
** SIMPLE.RSC
**      This program inserts a row into "myrelation".
*/

main()
{
        INITRSC("simple");

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

#line 12 "simple.rsc"
        exit(RS_NORM);
}

#line 15 ""after end of file on simple.rsc"

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.rsc*, the source file in Section 1.1.1, and *simple.c*, the output file in Section 1.1.3, demonstrates how RSC precompiles a syntactically correct RSC source file.

First, RSC 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 RSC 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 RSC was invoked.

The next line, #line 1 "simple.rsc", 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 RSC 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 RSC 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, RSC scans the file copying the input file to the output file until it reads a dollar sign ($). The dollar sign ($) is a signal to RSC that it has encountered either a SQL statement or a C identifier which is used in a SQL statement. Either situation requires special processing by RSC. In this example, RSC encounters an SQL statement in the form of the **insert** command in lines 10 through 12 of the source. RSC transforms this statement into

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

in the output file. RSC 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 table and the columns being inserted) as a series of octal bytes comprising the variable *utrees*. These bytes define the specific SQL command as a tree which can be understood by the database server. RSC 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 RSC creates from *simple.rsc* 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 RSC (1I) in the *Host Software Specification* for Unix systems and the *Command Summary* for other systems for instructions on compiling and linking a RSC 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

RSC 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*
tinyint	1	char
tinyint	1	BOOL
smallint	2	short
integer	4	long
smallfloat	4	float
float	8	double
char	variable	char[]
fixed char	variable	char[]
bcd	variable	BCDNO
bcdflt	variable	BCDNO

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

Data of the database server BINARY data type should be selected from or inserted into 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];

/* insert binary data */
$ insert into myrelation
  (bincolumn)
  values (binary ($bindata));

/* select binary data */
$ select $bindata=string(100, bincolumn)
  from myrelation;
```

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 RSC 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

RSC 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 RSC 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 RSC source, not the C compiler source, so that the RSC programmer can make corrections on the original RSC source file and then run this file through RSC again. It should never be necessary for the RSC programmer to write on the RSC 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. RSC (1I)

An indispensible reference for anyone using RSC is the entry for RSC 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 RSC program which includes

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

1.4.2. INITRC

The macros INITRSC, 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. SQL

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

1.4.5. RIC

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

2. Programming with RSC

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

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

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

create table author
(
        authnum  smallint,
        first    char(10),
        last     char(15)
)

create table authttl
(
        authnum  smallint,
        docnum   smallint
)
```

2.1. SQL in C Code

The following RSC program embeds an SQL **insert** command in C code.

```
1  /*
2  **  APPLIT.RSC
3  **
4  **          This program inserts a row into the "title" table.
5  */
6
7  main()
8  {
9      INITRSC("applit");
10
11     $insert into title
12             (docnum, title, onhand)
13             values (max(docnum) + 1, 'a flag for sunrise', 7);
14
15     exit(RS_NORM);
16 }
```

2.1.1. Delineating SQL Commands

An embedded SQL command always begins with a dollar sign ($). If there were no dollar sign, RSC 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 SQL command terminates with a semicolon (;) except for a looped **select** command which has its SQL portion delineated by a left curly brace ({). This exception is illustrated in Section 2.3.2.

An SQL command embedded in a RSC program may span more than one line. Multiple SQL 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 RSC:

```
$select last from author; $select onhand from title;

$insert into title
(docnum,
 title,
 onhand)
values
(max(docnum) + 1,
 'a flag for sunrise',
 7);

$select last, first from author
{
       printf("\n %s, %s\n", first, last);
}
```

2.1.2. Placement of SQL Commands

Embedded SQL commands may occur anywhere in a RSC source file where executable C statements may occur.

2.2. C Expressions in SQL Commands

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

```
1  /*
2  **  APPVAR.RSC
3  **
4  **          This program inserts rows to the "title" table,
5  **          getting input from the user.
6  **
7  **          Demonstrates the use of C variables in RSC 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 SQL code */
15     $char  newtitle[36];
16     $short newquan;
17     char   buf[5];
18     char   dbname[13];
19
20     INITRSC("appvar");
21     getprompt(dbname, sizeof(dbname), "Enter database name: ");
22     RCDBNAME(dbname);
23
24     /* loop on user input until user signals <RETURN> */
25     for (;;)
26     {
27           /* get user input for newtitle and newquan */
28           getprompt(newtitle, sizeof(newtitle),
29                 "Enter title or <RETURN> to quit:");
30           if (newtitle[0] == '\0')
31                 break;
32           getprompt(buf, sizeof(buf), "Enter quantity: ");
33           newquan = atos(buf);
34
35           /* insert the new row into the "title" table */
36           $insert into title
37                 (docnum, title, onhand)
38                 values (max(docnum) + 1, $newtitle, $newquan);
39     }
40
41     exit(RS_NORM);
42 }
```

2.2.1. C Declarations Used by RSC

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

On line 38, inside the **insert** command, *newtitle* and *newquan* appear prefaced with dollar signs ($) because they are embedded in SQL code. When these same variables are used in pure C code, in lines 28, 30, and 33, they are not prefaced with a dollar sign ($). The point to remember is that the dollar sign ($) must preface a C declaration only if the identifier will later appear in an SQL statement, and the dollar sign ($) must preface the identifer when it appears in an SQL statement.

C identifiers used in SQL 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 SQL 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 SQL expression. If the C expression is a simple C variable name, it need not be enclosed in parentheses when it is used in the SQL statement:

```
$short newquan;

/* no parentheses necessary - newquan is a simple variable name */
$update title set onhand = $newquan;
```

If the C expression is more complex than a simple variable name, the expression must be enclosed in parentheses in the SQL 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 */
$update title set onhand = $((newquan + oldquan) / num) . . .
```

RSC 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**.

RSC can handle all C expressions, including casts such as

```
$update title set docnum = $((short) 3.7) . . .
```

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

2.3. Selects

The last program used C expressions as values in an SQL **insert** command. RSC also uses C identifiers to store the values of target-list elements fetched from the database server by an SQL **select** 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 **select** command:

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

$select $num = authnum, $lname = last from author
       where . . .
```

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

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

$select authnum, last from author
       where . . .
```

There are two forms of **select** command in RSC programs, the "singleton" form, which selects a single row and simply stores its results, and the "looped" form, which may select more than one row and process selected data as it is fetched.

2.3.1. Singleton Selects

This program inserts a row into the "title" table and then selects it.

```
1  /*
2  **  SINGLE.RSC
3  **
4  **          This program inserts a specific title to the "title" table
5  **          and selects it. Demonstrates singleton selects.
6  **
7  */
8
9  main()
10 {
11     $char  title[36];
12     $short onhand;
13     $short docnum;
14
15     INITRSC("single");
16
17     /* insert a new row */
18     $insert into title
19            (docnum, title, onhand)
20            values (max(title.docnum) + 1, 'the color purple', 7);
21
22     /* select the new row, implicit assignment of target elements */
23     $select docnum, title, onhand from title
24            where docnum =
25                   (select max(docnum) from title);
26
27     printf("\nThe new row is:\n%d\t%s\t%d\n", docnum, title, onhand);
28
29     exit(RS_NORM);
30 }
```

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

2.3.2. Looped Selects

Looped **select** commands in RSC source are precompiled into C **for** loops in the output file. C code to process the selected data is executed inside the **for** loop for each selected row. If no rows are selected, the loop body does not execute.

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

It is important to note that the **select** command on lines 19 and 20 is not terminated by a semicolon (;). Instead, the SQL 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 RSC that this is a looped **select**. If the **select** command were terminated with a semicolon (;) RSC would treat it as a singleton and generate code to select one row. As in C, the body of the loop terminates with a right curly brace (}).

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

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

```
1   /*
2   **  GETDOCS
3   **
4   **        Selects the rows and puts them in an array, docs[].
5   **        Returns the number of rows selected.
6   */
7
8   getdocs(docs)
9      $int   docs[];
10  {
11     int    n = 0;
12
13     $select $(docs[n++]) = docnum from title
14            where onhand < 8
15     {
16            /* just fill the array */
17            continue;
18     }
19     return (n - 1);
20  }
```

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

```
1   /*
2   **  GETDOCS
3   **
4   **        Selects the rows and puts them in an array, docs[].
5   **        Returns the number of rows selected.
6   */
7
8   getdocs(docs)
9      $int   docs[];
10  {
11     int    n = 0;
12
13     $select $(docs[n++]) = docnum from title
14            where onhand < 8
15     {
16            /* just fill the array */
17            continue;
18     }
19     return (rccount());
20  }
```

2.3.3. Memory for Strings

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

```
1   /*
2   ** BADFUNC -- illustrates improper usage
3   */
4
5   badfunc()
6   {
7      $char  *last;
8
9      /* last does not point to anything */
10     $select $last from author
11     {
12           printf("%s ",last);
13     }
14  }
```

Memory can be allocated statically as in

```
1   /*
2   ** GOODFUNC -- illustrates proper usage
3   */
4
5   goodfunc()
6   {
7      /* allocate for the length of the column plus the NULL byte */
8      $char  last[16];
9
10     $select $last from author
11     {
12           printf("%s ",last);
13     }
14  }
```

Memory can also be allocated dynamically using the IDMLIB functions *xalloc* or *savestr*. The function *savestr* allocates enough memory to store the selected 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" table, using *xalloc* and *savestr*.

```
1   /*
2   **  LISTAUTHS.RSC
3   **
4   **        This program builds a linked list of authors names from the
5   **        "author" table.  Demonstrates xalloc and savestr.
6   */
7
8   #include <idmmpool.h>
9
10      struct list
11      {
12              char    *name;
13              struct list  *next;
14      }
15
16  main()
17  {
18      $char  inbuf[26];
19      struct list  *plist = (struct list *) NULL;
20      struct list  *new;
21      struct list  *p;
22
23
24
25      INITRSC("listauths");
26
27      $select $inbuf = concat(first, last) from author
28      {
29              /* allocate a new list element */
30              new = (struct list *) xalloc(sizeof(*new), DefMpool);
31
32              /* push the new element to the front of the list */
33              new ->next = plist;
34              plist = new;
35
36              /* copy the new string into its place */
37              new->name = savestr(inbuf, DefMpool);
38      }
39
40      exit(RS_NORM);
41  }
```

2.3.4. Nested Selects

It is possible to nest an SQL **select** command inside of another SQL **select** command.

```
1   /*
2   **  NEST.RSC
3   **
4   **        Displays all the books in the "title" table and
5   **        for each book displays all its authors.
6   **        Demonstrates nested selects.
7   */
8
9   main()
10  {
11     $char  name[16];
12     $char  title[36];
13     $short docnum;
14
15     INITRSC("nest");
16
17     /* outer loop selects and displays titles */
18     $select title, docnum from title
19     {
20           printf("%s\n", title);
21
22           /* inner loop selects and displays authors for each title */
23           $select $name = last from author
24                 where author.authnum = authttl.authnum and
25                 $docnum = authttl.docnum
26           {
37                 printf("\t%s\n", name);
28           }
29           printf("\n");
30     }
31
32     exit(RS_NORM);
33  }
```

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

2.4. Canceling SQL Commands

At times it may be desirable to cancel all SQL activity on the database server. To do this, use the special SQL command, **cancel** prefaced by a dollar sign. This command is only available in embedded languages; it does not exist in interactive SQL. The **cancel** command cancels all database server activity on the current dbin and any related dbins. If the **cancel** occurs inside a **select** loop, the C **break** command must be used to exit the loop. If the **select** 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.rsc*, could be re-written as follows to cancel all database server activity if an invalid author name is selected.

```
1   /*
2   **  CANCEL.RSC
3   **
4   **      Displays all the books in the "title" table and
5   **      for each book displays all its authors.
6   **      If an author name not beginning with an alphabetic character
7   **      is selected, cancels the inner and outer selects.
8   **      Demonstrates $cancel.
9   */
10
11  #define   NOTALPHA      !(c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z')
12  main()
13  {
14     $char  name[16];
15     $char  title[36];
16     $short docnum;
17     char   c;
18     BOOL   cancelled = FALSE;
19
20     INITRSC("cancel");
21
22     /* outer loop selects and displays titles */
23     $select title, docnum from title
24     {
25           printf("%s\n", title);
26
27           /* inner loop selects and displays authors for each title */
28           $select $name = last from author
29                 where author.authnum = authttl.authnum and
30                 $docnum = authttl.docnum
31           {
32                 c = name[0];
33                 if (NOTALPHA)
34                 {
35                       cancelled = TRUE;
36                       $cancel;
37                       break;        /* from inner loop */
38                 }
39                 else
40                       printf("\t%s\n", name);
41           }
42           if (cancelled)
43                 break;        /* from outer loop */
44           else
45                 printf("\n");
46     }
47
48     exit(RS_NORM);
49  }
```

3. Advanced Programming with RSC

This chapter describes methods for making RSC programs more powerful

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

Transactions are described in the *SQL Reference Manual* pages for **set autocommit off**, **commit work**, and **rollback work**. Stored commands are described under **store** and **start**.

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:

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

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

```
store newauth
        insert into author
        (authnum, first, last)
        values
        (max(author.authnum) + 1, &f, &l)
        select authnum, first, last from author
                where authnum =
                        (select max(author.authnum))
end store



store addlink
        insert into authttl
        (authnum, docnum)
        values
        (&a, &d)
end store



store getstuff
        select docnum, onhand from title
        select first, last from author
end store
```

3.1. Stored Commands

Stored commands may be defined and executed from a RSC 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".

```
1  /*
2  **  STORED.RSC
3  **
4  **  This program inserts a row into the "title" table
5  **  using the stored command "addtitle". Parameters passed
6  **  to the stored command are C identifiers fetched from the
7  **  command-line with crackargv().
8  */
9
10 #include <crackargv.h>
11
12 $static  short  Quan;
13 $static  char   *Title;
14
15 /* the argument list */
16 ARGLIST  Args[] =
17 {
18    't', FLAGSTRING, 0, CHARNULL, CHARNULL, __ &Title, "Title: ", CHARNULL,
19    'q', FLAGSHORT, 0, CHARNULL, CHARNULL, __ &Quan, "Quantity: ", CHARNULL,
20    '\0'
21 };
22
23 main(argc, argv)
24     int  argc;
25     char *argv[];
26 {
27     INITRSC("stored");
28      crackargv(argv, Args);
29
30     $start addtitle(q = $Quan, t = $Title);
31     exit(RS_NORM);
32 }
```

3.1.1. Obtaining Selected Data from Stored Commands

If the stored command contains one or more **select** commands, the **start** 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 selected data to C program variables. This is a special command which is only available in embedded SQL; there is no **obtain** command in interactive SQL. The only kind of $-flagged statements allowed within the **start** block are **obtain** statements.

There must be one **obtain** command flagged with a dollar sign for each **select** 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 **select.** The table in Section 1.2 maps database server types to C types.

The first column returned by the **select** command is assigned to the object indicated by the first expression passed to **obtain**; the second column returned by the **select** 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 selected data, or by a semicolon (;), if the only processing of selected data is assignment to the designated C identifiers. If the **obtain** command is terminated by a semicolon (;), the command still loops for every row selected; it does not simply fetch the first row as in a singleton **select**.

When the curly braces are used, the executable statements in the **obtain** block must be C statements, not $-flagged SQL statements. The precompiler checks for $-flagged SQL 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 SQL statements. This code elicits an error message at precompile time

```
$obtain $num, $name
{       if (num = badnum)
                $delete from rel where 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 from rel where num = $badnum;
        return()
}
```

The following program uses the **obtain** command to bind data selected 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 rows 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.

```
1  /*
2  **  GETDATA.RSC
3  **
4  **  This program executes the stored command "getstuff".
5  */
6
7  #define MAXITEM   15
8  #define MAXNAME   20
9
10 main()
11 {
12     $short docnums[MAXITEM], quantities[MAXITEM];
13     $char  fnames[MAXITEM][MAXNAME], lnames[MAXITEM][MAXNAME];
14     int    i,j,k;
15
16     INITRSC("getdata");
17     i = j = 0;
18
19     $start getstuff
20     {
21           /* fill the arrays */
22           $obtain($(docnums[i]), $(quantities[i++]));
23           $obtain($(fnames[j]), $(lnames[j++]));
24     }
25     /* decrement the counters */
26     j--; i--;
27
28     /* print the arrays */
29     printf("\ndocnum\tquantity");
30     for (k = 0; k < i; k++)
31           printf("\n%d\t%d", docnums[k], quantities[k]);
32
33     printf("\nauthors");
34     for (k = 0; k < j; k++)
35           printf("\n%s\t%s", fnames[k], lnames[k]);
36
37     exit(RS_NORM);
38 }
```

The next program uses the **obtain** command to bind data selected 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 row obtained.

```
1  /*
2  **  NEWBOOK.RSC
3  **
4  **          This program enters new books into the "books" database.
5  **
6  **          Selects the title from the "title" table.
7  **          If it is there, adjusts "onhand" column.
8  **          If it is not there, adds appropriate row to the "title",
9  **          "author", and "authttl" tables, using the stored commands
10 **          "newtitle", "newauth", and "addlink".
11 **
12 **          Demonstrates use of the obtain command to execute stored
13 **          commands containing selects from a RSC program.
14 */
15
16 main()
17 {
18     $char  title[36];
19     $char  lname[16];
20     $char  fname[11];
21     $short onhand;
22     $short docnum;
23     $short authnum;
24     $char  newname[36];
25     $short newquan;
26     char   buf[5];
27
28     INITRSC("newbook");
29
30     getprompt(newname, sizeof(newname), "Enter title:");
31     getprompt(buf, sizeof(buf), "Enter quantity: ");
32     newquan = (short) atos(buf);
33
34     /* select the docnum from the "title" table */
35     $select docnum from title
36            where title = $newname;
37
38     /* if the title is already in the database */
39     if (docnum != '\0')
40     {
41            $update title set onhand = titles.onhand + $newquan
42                   where title.docnum = $docnum;
43            $select docnum, title, quan from title
44                   where title = $newname;
45
46            printf("The new row is:\n%d\t%s\t%d\n", docnum, title, onhand);
47            exit(RS_NORM);
48     }
49     else
```

```
50      /* add the book to the database */
51      {
52              /* invoke stored command "newtitle" */
53              $start newtitle(q = $newquan, t = $newname)
54              {
55                      /* bind selected data to designated C identifiers */
56                      $obtain $docnum, $title, $onhand
57                      {
58                              printf("\nThe new row is:\n");
59                              printf("%d\t%s\t%d\n", docnum, title, onhand);
60                      }
61              }
62
63              /* select the author from the "author" table */
64              getprompt(fname, sizeof(fname), "Enter author's first name: ");
65              getprompt(lname, sizeof(lname), "Enter author's last name: ");
66              $select authnum from author
67                      where first = $fname and last = $lname;
68
69              /* if the author is not in the database */
70              if (authnum == '\0')
71              {
72                      /* invoke stored command "newauth" */
73                      $start newauth(f = $fname, l = $lname)
74                      {
75                              /* bind the new author number with authnum) */
76                              $obtain $authnum, $fname, $lname
77                              {
78                                      printf("\nThe new row is:\n");
79                                      printf("%d\t%s\t%s\n", authnum, fname, lname");
80                              }
81                      }
82              }
83
84              /*
85              **  add a row to the "authttl" table  using stored
86              **  command "addlink"
87              */
88              $start addlink(a = $authnum, d = $docnum);
89      }
90
91      exit(RS_NORM);
92  }
```

3.2. Transactions

A transaction is a sequence of one or more SQL 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 table** or **drop** inside a transaction will cause an exception to be raised.

In interactive SQL, a transaction begins when the user executes the **set autocommit off** command, or after a **commit work** or **rollback work** command is issued. In interactive SQL, a transaction ends when a **commit work** or **rollback work** command is issued or when a **set autocommit on** command is executed.

The embedded SQL used by RSC does not use these statements. The precompiler does not accept the commands **set autocommit**, **commit work**, and **rollback work**, and gives an error message if it encounters them. Instead, a **begin transaction** command initiates a transaction, an **end transaction** command terminates a transaction, and an **abort transaction** rolls back a transaction, nullifying the effects of its constituent commands. At execution time there is no transaction automatically in effect between the execution of an **end transaction** or **abort transaction** and the execution of the next **begin transaction**.

The only commands which can be used in a transaction in a RSC program are:

- **abort transaction**
- **insert**
- **begin transaction**
- **drop**
- **end transaction**
- **update**
- **select**
- **sync**
- **delete**

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 SQL queries may be inserted between the **begin transaction** and the **end transaction** in a RSC program.

The following program demonstrates the use of a simple transaction in a RSC program. Rows are inserted into three tables 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 tables or not at all.

```
1   /*
2   ** TRANS.RSC
3   **
4   **      This program enters a new book into the "books" database.
5   **
6   **      Adds a row to the "title" and "authttl" tables and may
7   **      add a row to the "author" table.
8   **
9   **      Demonstrates RSC support for transactions.
10  **      Aborts if user tries to add a duplicate title.
11  **      Gets title and author from the command-line using the IDMLIB
12  **      function, crackargv.
13  */
14
15  #include <crackargv.h>
16
17     /* variables for command-line arguments */
18     $short Quan;
19     $char  *Title;
20     $char  *Fname;
21     $char  *Lname;
22
23     /* the argument list */
24     ARGLIST      Args[] =
25     {
26           FLAGPOS, FLAGSHORT, 0, CHARNULL, CHARNULL, __ &Quan,
27                 "Quantity:", CHARNULL,
28           FLAGPOS, FLAGSTRING, 0, CHARNULL, CHARNULL, __ &Title,
29                 "Title:", CHARNULL,
30           FLAGPOS, FLAGSTRING, 0, CHARNULL, CHARNULL, __ &Fname,
31                 "Author's first name:", CHARNULL,
32           FLAGPOS, FLAGSTRING, 0, CHARNULL, CHARNULL, __ &Lname,
33                 "Author's last name:", CHARNULL,
34           '\0'
35      };
36
37  main(argc, argv)
38     int    argc;
39     char   **argv;
40  {
41     $short authnum;
42     $short docnum;
43
44     INITRSC("trans");
45
46     /* get values from command-line for Quantity, Title, Fname, Lname */
47     crackargv(argv, Args);
48
49     $begin transaction;
50     $insert into title
51           (docnum, title, onhand)
52           values
53           (
54                 max(title.docnum) + 1,
55                 $Title,
56                 $Quan
57           );
```

```
58      $select docnum from title
59              where title = $Title and docnum
60      {
61              ;
62      }
63      /* check for duplicates */
64      if (rccount() > 1)
65      {
66              printf("\nThat title is already in the database.");
67              printf("\nAborting this transaction");
68              /* control passes to statement after "end transaction" */
69              $abort transaction;
70      }
71
72      /* check if the author is in the "author" table */
73      $select authnum from author
74              where first = $Fname and last = $Lname;
75
76      /* if the author is not in the table */
77      if (authnum == '\0')
78      {
79              $insert into author
80              (authnum, last, first)
81              values
82              (
83                      max(author.authnum) + 1,
84                      $Fname,
85                      $Lname
86              );
87              $select authnum from author
88                      where authnum =
89                              (select max(author.authnum));
90      }
91
92      $insert into authttl
93              (authnum, docnum)
94              values ($authnum, $docnum);
95
96      $end transaction;
97      exit(RS_NORM);
98  }
```

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;
        /* SQL 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.

```
1   /*
2   ** NESTTRANS.RSC
3   **
4   **        This program displays all the books in the "title" table
5   **        and for each book displays all its authors.
6   **
7   **        Demonstrates nested transactions.
8   **        Functionally like NEST.RSC, except commands are transactions.
9   **        If a title is found with no associated author,
10  **        an error message is displayed and all processing is
11  **        halted by the "abort transaction".
12  */
13
14  main()
15  {
16     $char  title[36];
17     $short docnum;
18     void   getauths();
19
20     INITRSC("nesttrans");
21
22     /* outer loop to select and display titles */
23     $begin transaction;
24
25     $select title, docnum from title order by docnum
26     {
27            printf("%s\n", title);
28            getauths(docnum, title);
29            printf("\n");
30     }
31
32     /*
33     ** commit everything from the above select
34     ** and the one in getauths
35     */
36
37     $end transaction;
38
39     exit(RS_NORM);
40  }
41
42  /*
43  **  GETAUTHS -- select and display the authors of the title
44  **  passed in docnum.
45  */
46
47  void
48  getauths(docnum, title)
49     $short docnum;
50     $char  *title;
51  {
52     $char  name[16];
53
54     /* this is a nested transaction when called from main */
55     $begin transaction;
56     $select $name = last from author
57            where authttl.authnum = author.authnum and
58            authttl.docnum = $docnum;
59     {
```

```
60              printf("\t%s\n", name);
61      }
62      /* if the name is not there */
63      if (!rccount())
64      {
65              printf("\nNo entry for an author of %s.", title);
66              printf("\nAborting this transaction so user can");
67              printf("\nmodify the 'author' and/or 'authttl' relations.");
68
69              /* transfers control to statement after end transaction in main */
70              $abort transaction;
71      }
72
73      /* other SQL commands could go here. */
74      $end transaction;
75  }
```

3.2.2. New Transactions

In the program above, all processing halts after execution of the **abort transaction** on line 70 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.rsc* so that a new transaction is initiated in *getauths()* change the **begin transaction** on line 55 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 29 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:

```
f()
{
        $begin transaction;

        /* this update locks the whole relation */
        $update title set title = 'untitled';
        g();
        $end transaction;
}

g()
{
        $begin new transaction;
        $update title set title = 'UNIX SYSTEMS'
                where 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 Select Loops

Special issues arise when a program performs an update inside a **select** loop. The update cannot simply be nested in the select loop as in *nesttrans.rsc* because of the way select 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 select 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 select loop:

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

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

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

        $end transaction;
}
```

The next program, *reexam.rsc*, demonstrates the use of a single level of nesting to execute an **update** command inside a **select** loop. As each row in the "titles" table is selected, the user is prompted to update the value of the "onhand" column in that row.

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, RSC generates code which catches the exceptions and attempts to restart the query immediately.

If you wish your RSC 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.

```
1   /*
2   **  REEXAM.RSC -- runs an update inside a select loop.
3   **
4   **        Selects each row in the "title" table.
5   **        Allows the user to update the "onhand" column of
6   **        the selected row.
7   **
8   **        Demonstrates "begin nest <n> transaction".
9   **        Provides one level of nesting - one parent and one child.
10  **        The "begin nest <n> transaction" is necessary when there is
11  **        an update inside a select loop.
12  **
13  **        Sets an exception handler to display a message and re-raise
14  **        the exception to restart the transaction when W:IDM.DONE.XABORT
15  **        is raised. Uses canned handler excignore for E:IDM:E55.
16  */
17
18  main()
19  {
20     $char  title[36];
21     $short docnum;
22     $short onhand;
23     $short newstock;
24     char   buf[5];
25     int    handler();
26
27
28     INITRSC("reexam");
29
30     $begin nest 1 transaction;
31
32     /* set exception handlers for deadlocks */
33     exchandle("E:IDM.E55", excignore);
34     exchandle("W:IDM.DONE.XABORT", handler);
35     printf("\nDOCNUM     TITLE          ONHAND\n");
36
37     /* run select on parent dbin */
38     $select docnum, title, onhand from title
39     {
40           printf("%d\t%s\t%d\n", docnum, title, onhand);
41
42           /* get input for the update */
43           getprompt(buf, sizeof(buf),
44                 "Enter new stock or <RETURN> to quit: ");
45           if (buf[0] == '\0')
46                 break;
47           newstock = atos(buf);
48
49           if (newstock == 0)
50                 printf("\nNo change in # copies of %s.\n", title);
51           else if (newstock > 0)
52                 printf("\nAdding %d copies of %s.\n", newstock, title);
53           else
54                 printf("\nSubtracting %d copies of %s.\n",
55                       newstock * -1,  title);
56
```

```
57            /* run update on nested child dbin */
58            $update title set onhand = onhand + $newstock
59                  where docnum = $docnum and title = $title;
60      }
61
62      $end transaction;
63
64      /* remove the exception handlers */
65      exchandle("E:IDM:E55", FUNCNULL);
66      exchandle("W:IDM:DONE:XABORT", FUNCNULL);
67
68      exit (RS_NORM);
69  }
70
71  /*
72  **  HANDLER
73  **
74  **        Displays message before attempting to restart the query.
75  */
76
77  handler(excv)
78      char **excv;
79  {
80
81      fprintf(stderr, "\nYour query has deadlocked with another");
82      fprintf(stderr, "application and your work is being backed out.");
83      fprintf(stderr, "\nAn attempt is being made to restart the query.");
84
85      /* call RSC's exception handler to attempt restart */
86      excvraise(excv);
87      return (0);
88  }
```

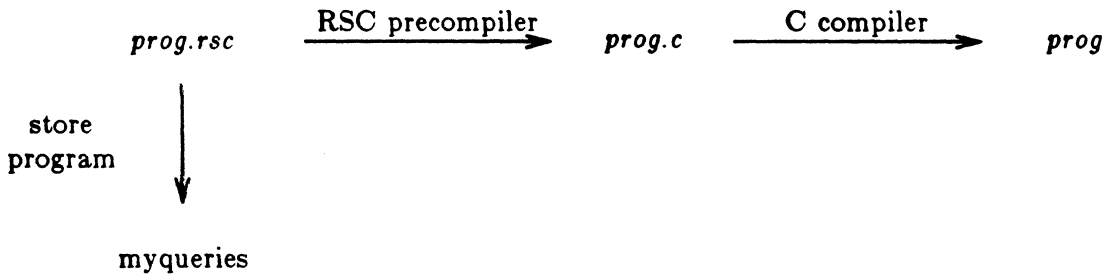
You may wish for a RSC program to catch all exceptions, not just "E:IDM.55", and "W:IDM:DONE.XABORT". The following statement declares *myhandler()* to handle all exceptions of severity Error.

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

Appendix A: Stored Programs

All of the query trees which RSC builds from SQL code are normally stored in the executable program which is created when the output of RSC 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.



Queries may be stored on the database server only if the RSC 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.

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 RSC 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 RSC 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 RSC to precompile *myprog.rsc*, storing the query trees on the database server as a stored program in the "books" database named "myqueries".

rsc –n "myqueries" –d "books" myprog.rsc (Unix)

rsc /progname = myqueries /dbname = books myprog.rsc (other)

If a RSC 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 RSC to precompile a program residing in two source files, and to store the queries in *main.rsc* in a stored program called "mainqueries" and the queries in *functions.rsc* in a stored program called "funcqueries". Both "mainqueries" and "funcqueries" are stored in the "books" database:

UNIX:

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

VMS:

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

PC/MS-DOS

```
rsc /progname=mainqueries  /dbname=books main.rsc
rsc /progname=funcqueries  /dbname=books functions.rsc
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

```
rsc  -n "mainqueries"  -d "books"  main.rsc
rsc  -n "funcqueries"  -d "books"  functions.rsc
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: RSC 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 RSC source file before sending it through RSC to obtain **struct** definitions and **typedef**s given in a header file so that these definitions may be applied to C variables which are known to RSC.

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 **.rsc** or no suffix.

Flagging Statements for RSC Header Files

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

To overcome these contradictory requirements, all header files which must be acceptable to RSC 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 RSC should be flagged with the word "RCFLAG" instead of a dollar sign ($).

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

```
1   /*
2   **  EXAMPLE.H - header file to be used by RSC and a C compiler.
3   */
4
5   #include <rcflag.h>
6
7   RCFLAG    int    Num;
```

would be included in the RSC 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 RSC but not in any other files.

The #line Directive

A directive that reads

```
#line  3 "file.rsc"
```

instructs the C compiler to consider the next line of text to be the third line read from a file named *file.rsc*, regardless of the name of the file it is actually reading or the number of lines it has actually read. RSC writes many of these directives on the output file it produces to allow error messages from the C compiler to reference lines in the RSC source. RSC 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. RSC 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.rsc"

(2) # line 3 "file.rsc"

(3) # 3 "file.rsc"

(4) # line 3 file.rsc

(5) # line 3 file.rsc

(6) # 3 file.rsc

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

If the **-l** command-line option is passed to RSC 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 RSC output which was run through the C preprocessor when it was RSC 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 RSC, quotation marks will not enclose the filename in the *#line* directives. This is required by some C compilers.

Appendix C: Portability of RSC 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 RSC program is precompiled, each SQL 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 RSC 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 RSC source must be precompiled again on the destination host.
- If a RSC 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 RSC 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 RSC source.

With Stored Programs

If all of the SQL 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 RSC program is ported.

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

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

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

(3) The SQL statements may only be among the following: **select**, **insert**, **update**, **drop**, **begin transaction**, **end transaction**, or **abort transaction**.

If conditions 2 and 3 do not exist, but the RSC program is compiled with the **–n** or **/progname** option, the precompiler will create stored programs for the allowable SQL 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 RSC 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 RSC 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 RSC 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:

```
rsc –n "VAXmyqueries" –d "books" myprog.rsc
```

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

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

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 RSC 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 **select**, **insert**, **update**, **drop**, **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 SQL 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