SIR API Overview

The *SIR/APIs* are two simple sets of five functions (one for SIRDBMS and one for SIRSQL). These APIs can be used to develop your own front-ends for SIR database management engines.

The `INIT` function initialise the system and defines two call back routines to handle the output from the API. The first of these callbacks (writeLine) handles output which would normally go to the main window, and the second (writeData) handles output from query type (`LIST`) commands. The initialise function also defines the start parameter string that you would normally use to start SIRDBMS or SIRSQL (eg: `DB=COMPANY PW=COMPANY P=...`). This initial function should be called first.

The `EXEC` function executes a DBMS or SQL command or set of commands.

The `STOP` function terminates the API, closing any open files and releasing handles. It should be called at the end of your program.

The other two functions return the `VersionNumber` (this one can be called before INIT) and the `ERROR` code. The latter can be called after any API function that has a FALSE return code.

Notes:

Your application will use SIR's DLLs, so the SIR's home directory must be included into your PATH (unless you place your executable file into that directory).

The system is not reenterable, so you must not call any of these functions before the previous call is completed. Also you cannot use both SirDBMS and SirSQL DLLs from the same application.

To compile your calls correctly under MS-Windows macro `_WIN32` must be defined. It is predefined by Visual C++. Define it in your make file for other compilers. The macro will inforce `__stdcall` calling convention for the DLL's export names. You should use sirdbms.lib or sirsql.lib import library to link your application.

To write a console application that does not have a windows interface then use a window handle (hWnd) of `(void *) 1`.

The DBMS Functions

SirDBMS_VersionNumber

```
int __StdCall SirDBMS_VersionNumber(void);
```
Returns DBMS engine's version number (40000 for 4.00.00). It's the only function you can execute before `SirDBMS_Init()`.

SirDBMS_Init

```
int __StdCall SirDBMS_Init(
    void *hWnd,
    void (__StdCall *writeLine)(const char *text),
    void (__StdCall *writeData)(const char *text),
    const char *commandLine
);
```
Must be executed before sending commands to the DBMS engine.

The first argument is the system-specific handle of the main window. It should be the current window at this moment.

DBMS engine uses the function passed as the second argument to write a line of text into the message buffer. User interface module should immediately put the text into some output window.

DBMS engine uses the function passed as the third argument to write a line of text into the data buffer when given some query-type command.

The engine doesn't add any end-of-line terminators, it just makes one call for each line of output.

The last argument is a string which contains the command line arguments (without the leading application name).

Returns `TRUE` on success, `FALSE` on failure, doesn't return if the command line activates the batch mode.

SirDBMS_Exec

```
int __StdCall SirDBMS_Exec(void *hWnd, const char *commands);
```
Executes DBMS commands.

The first argument is the system-specific handle of the current window. It is used as the owner of the engine's message boxes.

The second argument is a DBMS command[s]. Multiple lines must be separated by '\n' character.

Returns TRUE on success, FALSE on failure.

SirDBMS_Stop

```
int __StdCall SirDBMS_Stop(void *hWnd);
```
Call this before you terminate the program.

The argument is the system-specific handle of the current window. It is used as the owner of the engine's message boxes.

This will properly disconnect and close all open files.

Returns `TRUE` on success, `FALSE` on failure.

SirDBMS_ErrorCode

```
int __StdCall SirDBMS_ErrorCode(void);
```
Call this to get the error code if one of the SirDBMS_... functions returned FALSE. Error code 0 means that information is not available.

The DBMS LIST Commands

The LIST Commands are DBMS commands that send their output to the writeData callback routine.

- `LIST ATTRIBUTE LABELS`
- `LIST ATTRIBUTE VALUES`
- `LIST BUFFERLINE "buffername" [FROM start_line_number TO end_line_number]`
- `LIST CONCTDTABFILES`
- `LIST DBA`
- `LIST DEFFAMILY`
- `LIST DEFMEMBER`
- `LIST EDITORNAME`
- `LIST EDITORTYPE`
- `LIST ERRORLIMIT`
- `LIST FAMILIES`
- `LIST GLOBAL LABELS`
- `LIST GLOBAL VALUES`
- `LIST INDEXDATA tabfile.table NAMES`
- `LIST INDEXDATA tabfile.table TYPES`
- `LIST INDICES tabfile.table`
- `LIST LOADING`
- `LIST MEMBERS`
- `LIST MSGLIMIT`
- `LIST MSGLVL`
- `LIST OUTPUTFILE`
- `LIST PAGELEN`
- `LIST PAGEWID`
- `LIST PRINTBACK CALL`
- `LIST PRINTBACK COMMANDS`
- `LIST PRINTBACK FORMAT`
- `LIST PRINTBACK REMARK`
- `LIST PRINTBACK REPEAT`
- `LIST PRINTBACK USER`
- `LIST PROCFILE`
- `LIST RECORDS`
- `LIST SORTN`
- `LIST TABFILE [tabfile]`
- `LIST TABLEDATA tabfile NAMES`
- `LIST TABLEDATA tabfile TYPES`
- `LIST TABLES`
- `LIST TABLES tabfile`
- `LIST WARNLIMIT`

The SQL Functions

SirSQL_VersionNumber

```
int __StdCall SirSQL_VersionNumber(void);
```
Returns SQL engine's version number (40000 for 4.00.00). It's the only function you can execute before `SirSQL_Init()`.

SirSQL_Init

```
int __StdCall SirSQL_Init(
    void *hWnd,
    void (__StdCall *writeLine)(const char *text),
    const char *commandLine
);
```
Must be executed before sending commands to the SQL engine.

The first argument is the system-specific handle of the main window. And it should be the current window at this moment.

SQL engine uses the function passed as the second argument to write a line of text into the output buffer. User interface module might immediately add the text into the output window or use results in some other way after it gets control back. The engine doesn't add any end-of-line terminators, it just makes one call for each line of output.

The last argument is a string which contains the command line arguments (without the leading application name).

Returns `TRUE` on success, `FALSE` on failure, doesn't return if the command line activates the batch mode.

SirSQL_Exec

```
int __StdCall SirSQL_Exec(void *hWnd, const char *commands);
```
Executes SQL commands.

The first argument is the system-specific handle of the current window. It is used as the owner of the engine's message boxes.

The second argument is a SQL command[s]. Multiple lines must be separated by '\n' character.

Returns `TRUE` on success, `FALSE` on failure.

SirSQL_Stop

```
int __StdCall SirSQL_Stop(void *hWnd);
```
Call this before you terminate the program.

The argument is the system-specific handle of the current window. It is used as the owner of the engine's message boxes.

This will properly disconnect and close all open files.

Returns `TRUE` on success, `FALSE` on failure.

SirSQL_ErrorCode

```
int __StdCall SirSQL_ErrorCode(void);
```
Call this to get the error code if one of the `SirSQL_...` functions returned `FALSE`.

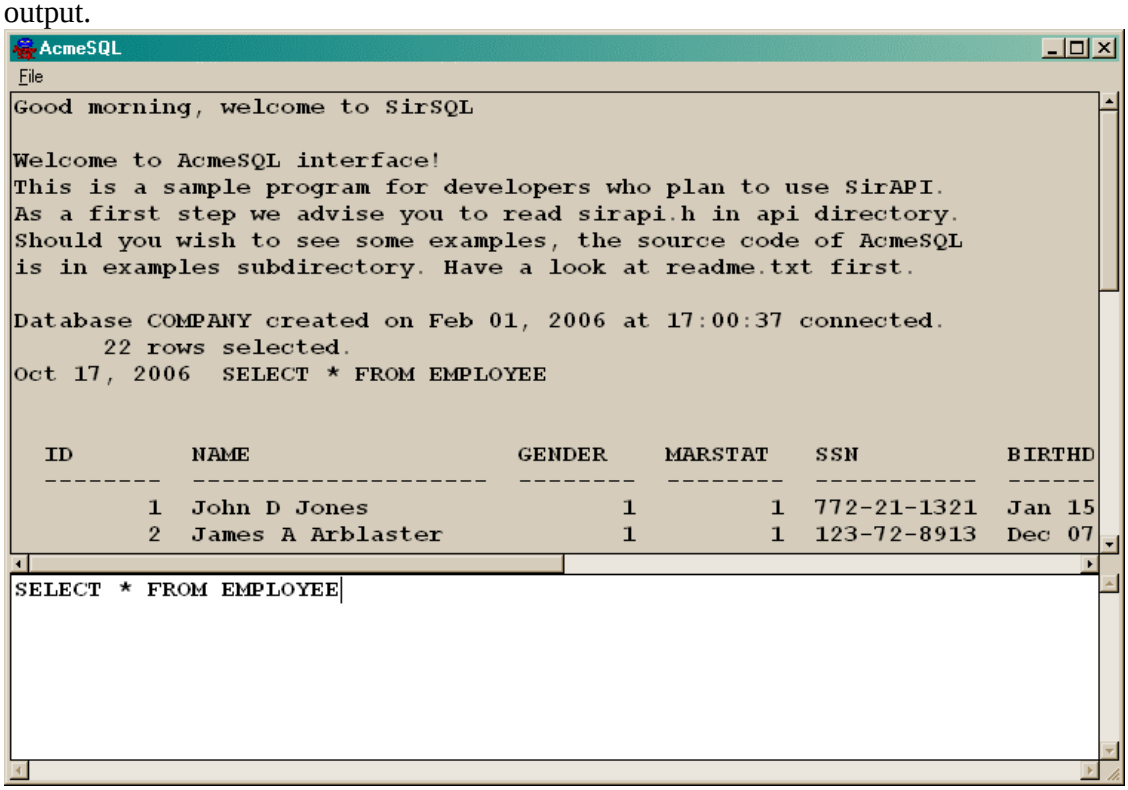
The SQL `LIST` Commands

The LIST Commands in SQL correspond to the SHOW command but send their output to the writeData callback routine.

Example Programs

ACME SQL

ACME SQL is a simple SQL interface that lets you enter SQL commands and view the output.




```
//*********************************************************************
*
//* AcmeSQL - example of using SirAPI (see readme.txt)
*
//*
*
//* This software is provided as is, without any explicit or
*
//* implied warranties. Use it at your own risk.
*
//*********************************************************************
*

#define STRICT
#include <windows.h>
#include <stdio.h>

#ifndef _WIN32
#define _WIN32
```

```
#endif

#include <sirapi.h>
#include "acmesql.rh"

const char *AppName = "AcmeSQL";
const char *IniFile = "acmesql.ini";

const int MinEngineVer = 41000;

int InpWndRatio = 30;
int AutoSave = FALSE;

HINSTANCE hInst;

HWND hFrame;
HWND hInpWnd;
HWND hOutWnd;

HFONT hFont;

char *GetInputText() {

    int length = GetWindowTextLength(hInpWnd);
    if (length == 0)
        return NULL;

    char *buffer = new char[length+1];

    length = GetWindowText(hInpWnd, buffer, length+1);

    int Empty = TRUE;
    for (int i=0; i<length; i++) if (!isspace(buffer[i])) {
        Empty = FALSE;
        break;
    }
    if (Empty) {
        delete buffer;
        return NULL;
    }

    return buffer;
}

void SetInputText(const char *text) {
    SetWindowText(hInpWnd, text);
}

void SetOutputText(const char *text) {
    SetWindowText(hOutWnd, text);
}

void AddToOutputWindow(const char *text) {

    int length = GetWindowTextLength(hOutWnd);
    int addlen = strlen(text);
```

```
    char *tmp = new char[addlen+2+1];
    memcpy(tmp, text, addlen);
    memcpy(tmp+addlen, "\r\n", 3);

    SendMessage(hOutWnd, EM_SETSEL, length, length);
    SendMessage(hOutWnd, EM_REPLACESEL, FALSE, (LPARAM)tmp);

    delete tmp;
}

void __StdCall OutputHandler(const char *text) {
    AddToOutputWindow(text);
}

int InitEngine(HWND hWnd, const char *CommandLine) {

    EnableWindow(hWnd, FALSE);

    int status = SirSQL_Init(hWnd, OutputHandler, CommandLine);
    if (status == FALSE)
        MessageBox(
            hWnd,
            "Cannot initialize the SQL engine", AppName,
            MB_OK | MB_ICONERROR);

    EnableWindow(hWnd, TRUE);

    return status;
}

int ShutdownEngine(HWND hWnd) {

    EnableWindow(hWnd, FALSE);

    int status = SirSQL_Stop(hWnd);
    if (status == FALSE)
        MessageBox(
            hWnd,
            "Cannot shutdown the SQL engine correctly", AppName,
            MB_OK | MB_ICONERROR);

    EnableWindow(hWnd, TRUE);

    return status;
}

int Execute(HWND hWnd, const char *command) {

    EnableWindow(hWnd, FALSE);

    SetCursor(LoadCursor(NULL, IDC_WAIT));

    int status = SirSQL_Exec(hWnd, command);
    if (status == FALSE)
        Beep(500, 100);

    SetCursor(LoadCursor(NULL, IDC_ARROW));
```

```
    EnableWindow(hWnd, TRUE);

    return status;
}

LRESULT MainWnd_OnCreate(HWND hWnd, char *CmdLine) {

    hFrame = hWnd;

    int dllVersion = SirSQL_VersionNumber();
    if (dllVersion < MinEngineVer) {
        char buffer[1024];
        sprintf(
            buffer,
            "Wrong dynamic link library\n\n"
            "The SQL engine from sirsql.dll version %d.%02d.%02d\n\n"
            "You must use SQL engine version %d.%02d.%02d or later",
            dllVersion/10000, dllVersion/100%100, dllVersion%100,
            MinEngineVer/10000, MinEngineVer/100%100,
MinEngineVer%100);
        MessageBox(hWnd, buffer, AppName, MB_OK | MB_ICONERROR);
        return -1;
    }

    int fontSize = GetPrivateProfileInt(AppName, "FSize", 0, IniFile);
    HDC hDC = GetDC(hWnd);
    int fontHeight = -MulDiv(fontSize, GetDeviceCaps(hDC, LOGPIXELSY),
72);
    ReleaseDC(hWnd, hDC);
    hFont = CreateFont(
        fontHeight, 0, 0, 0, FW_BOLD, 0, 0, 0, ANSI_CHARSET,
        OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, PROOF_QUALITY,
        FIXED_PITCH, "Courier New");
    if (hFont == NULL)
        hFont = CreateFont(
            0, 0, 0, 0, FW_BOLD, 0, 0, 0, ANSI_CHARSET,
            OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, PROOF_QUALITY,
            FIXED_PITCH, "Courier New");

    InpWndRatio = GetPrivateProfileInt(AppName, "InpWndRatio", 30,
IniFile);
    if (InpWndRatio < 15 || InpWndRatio > 75)
        InpWndRatio = 30;

    hInpWnd = CreateWindowEx(
        0, "EDIT", NULL,
        WS_CHILD | WS_VISIBLE | WS_VSCROLL | WS_HSCROLL | WS_BORDER |
        ES_LEFT | ES_MULTILINE | ES_AUTOVSCROLL | ES_AUTOHSCROLL |
        ES_WANTRETURN,
        0, 0, 0, 0, hWnd, (HMENU)10030, hInst, NULL);
    if (hFont != NULL)
        SendMessage(hInpWnd, WM_SETFONT, (WPARAM)hFont,
MAKELPARAM(TRUE,0));
    SendMessage(hInpWnd, EM_LIMITTEXT, 0, 0);

    hOutWnd = CreateWindowEx(
```

```
        0, "EDIT", NULL,
        WS_CHILD | WS_VISIBLE | WS_VSCROLL | WS_HSCROLL | WS_BORDER |
        ES_LEFT | ES_MULTILINE | ES_AUTOVSCROLL | ES_AUTOHSCROLL |
        ES_NOHIDESEL | ES_READONLY,
        0, 0, 0, 0, hWnd, (HMENU)10031, hInst, NULL);
    if (hFont != NULL)
        SendMessage(hOutWnd, WM_SETFONT, (WPARAM)hFont,
MAKELPARAM(TRUE,0));
    SendMessage(hOutWnd, EM_LIMITTEXT, 0, 0);

    AutoSave = GetPrivateProfileInt(AppName, "AutoSave", FALSE,
IniFile);

    if (!InitEngine(hWnd, CmdLine))
        return -1;

    AddToOutputWindow("");
    AddToOutputWindow("Welcome to AcmeSQL interface!");
    AddToOutputWindow("This is a sample program for developers who plan
to use SirAPI.");
    AddToOutputWindow("As a first step we advise you to read sirapi.h
in api directory.");
    AddToOutputWindow("Should you wish to see some examples, the source
code of AcmeSQL");
    AddToOutputWindow("is in examples subdirectory. Have a look at
readme.txt first.");
    AddToOutputWindow("");

    return 0;
}

LRESULT MainWnd_OnClose(HWND hWnd) {

    if (AutoSave)
        Execute(hFrame, "SAVE");

    ShutdownEngine(hWnd);

    DestroyWindow(hWnd);

    if (hFont != NULL)
        DeleteObject(hFont);

    return 0;
}

LRESULT MainWnd_OnSize(HWND , int w, int h) {

    int InpWndH = h*InpWndRatio/100;
    int OutWndH = h-InpWndH;

    MoveWindow(hOutWnd, 0, 0      , w, OutWndH, TRUE);
    MoveWindow(hInpWnd, 0, OutWndH, w, InpWndH, TRUE);

    return 0;
}
```

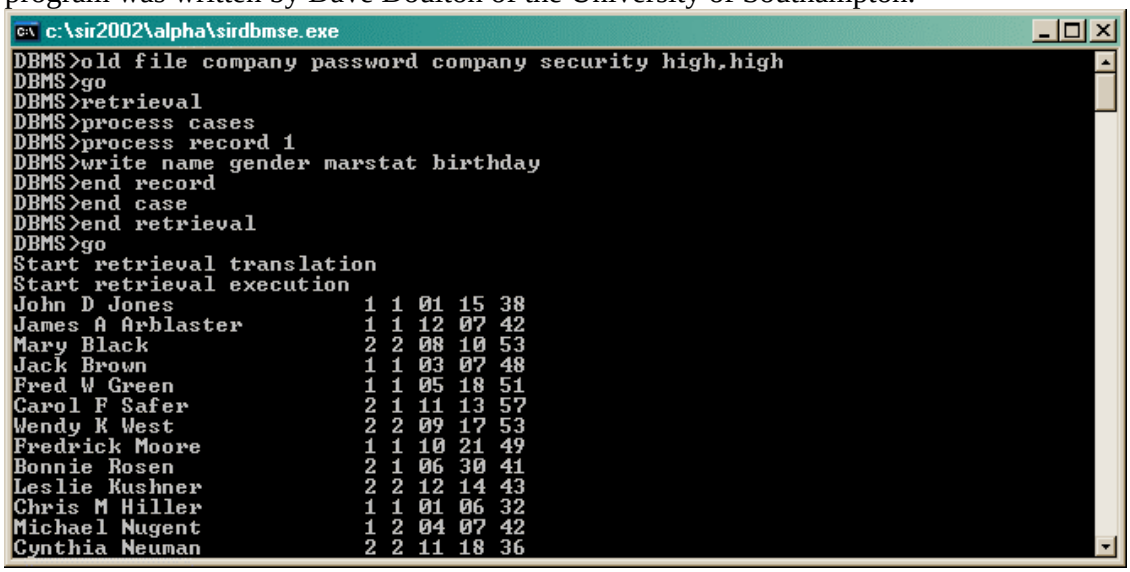
```
LRESULT MainWnd_OnSetFocus(HWND) {
    SetFocus(hInpWnd);
    return 0;
}

LRESULT MainWnd_OnCommand(HWND hWnd, WORD cmd, WORD NCode, HWND
hControl) {

    switch (cmd) {
        case CMD_EXECUTE: {
            char *cmd = GetInputText();
            if (cmd != NULL) {
                int len = strlen(cmd);
                int n = 0;
                for (int i = 0; i <= len; i++) {
                    if (cmd[i] == '\r') {
                        n++;
                    } else {
                        if (n != 0) {
                            cmd[i-n] = cmd[i];
                        }
                    }
                }
                if (Execute(hFrame, cmd))
                    SetInputText("");
                SetFocus(hInpWnd);
                delete cmd;
            }
            break;
        }
        case CMD_EXIT:
            SendMessage(hWnd, WM_CLOSE, 0, 0);
            break;
        default: ;
    }

    if (NCode == EN_MAXTEXT || NCode == EN_ERRSPACE) {
        Beep(500, 100);
        if (hControl == hOutWnd) {
            SetOutputText("");
        }
    }

    return 0;
}

LRESULT MainWnd_OnPaint(HWND hWnd) {
    HDC hDC;
    PAINTSTRUCT ps;
    hDC = BeginPaint(hWnd, &ps);
    EndPaint(hWnd, &ps);
    return 0;
}

LRESULT CALLBACK MainWnd_WndProc(
    HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam
) {
```

```
    typedef struct {
        short sz;
        void *p;
    } UNALIGNED *UP;

    switch (msg) {
        case WM_CREATE:
            return MainWnd_OnCreate(
                hWnd,
                (char *)((UP)(((LPCREATESTRUCT)lParam)-
>lpCreateParams))->p);
        case WM_CLOSE:
            return MainWnd_OnClose(hWnd);
        case WM_SIZE:
            return MainWnd_OnSize(hWnd, LOWORD(lParam),
HIWORD(lParam));
        case WM_SETFOCUS:
            return MainWnd_OnSetFocus(hWnd);
        case WM_COMMAND:
            return MainWnd_OnCommand(
                hWnd, LOWORD(wParam), HIWORD(wParam), (HWND)lParam);
        case WM_PAINT:
            return MainWnd_OnPaint(hWnd);
        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
        default: return DefWindowProc(hWnd, msg, wParam, lParam);
    }
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE, LPSTR lpCmdLine,
int) {

    hInst = hInstance;

    WNDCLASS wc;
    wc.style         = CS_VREDRAW | CS_HREDRAW | CS_DBLCLKS;
    wc.lpfnWndProc   = MainWnd_WndProc;
    wc.cbClsExtra    = 0;
    wc.cbWndExtra    = 0;
    wc.hInstance     = hInstance;
    wc.hIcon         = LoadIcon(hInstance,
MAKEINTRESOURCE(IDR_MAINICON));
    wc.hCursor       = LoadCursor(0, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName  = MAKEINTRESOURCE(IDR_MAINMENU);
    wc.lpszClassName = "MainWnd";
    if (RegisterClass(&wc) == 0)
        return 0;

    struct {
        short sz;
        void *p;
    } CreateWindowParam = {sizeof(void *), lpCmdLine};

    HWND hWnd = CreateWindowEx(
        WS_EX_APPWINDOW,
```

```
        wc.lpszClassName, AppName,
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
        HWND_DESKTOP, NULL, hInstance, &CreateWindowParam);
    if (hWnd == NULL)
        return 0;

    ShowWindow(hWnd, SW_SHOWDEFAULT);

    HACCEL hAccel = LoadAccelerators(
        hInstance, MAKEINTRESOURCE(IDR_ACCELTABLE));

    MSG msg;
    while (GetMessage(&msg, 0, 0, 0) == TRUE) {
        if (!TranslateAccelerator(hWnd, hAccel, &msg)) {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }

    DestroyIcon(wc.hIcon);

    return msg.wParam;
}
```

SIRDBMSE

SIRDBMSE is a SIRDBMS console application which lets you enter DBMS commands and program and run them in a command window or telnet connection. Enter commands at the `DBMS>` prompt and run them by entering `go`. Enter `exit` to end the program. This program was written by Dave Doulton of the University of Southampton.




```
#include <string.h>
#include <stdio.h>
#include "sirapi.h"
```

```
void OutputHandler(const char *text) {
        if(text[0] == ' ')
        {
    printf("%s\n",text+1);
        }
        else
        {
    printf("%s\n",text);
        }

}

void DisplayHandler(const char *text) {
        if(text[0] == ' ')
        {
    printf("%s\n",text+1);
        }
        else
        {
    printf("%s\n",text);
        }
}


int main(int argc,char *argv[]) {
int  i;
int res;
char cmdLine[1024] = "";
char cmd[1024]= "";
char ocmd[1024]= "";
char line[200]="";
   for (i = 1; i < argc; i++) {
        if (i > 1) strcat(cmdLine, " ");
        strcat(cmdLine, argv[i]);
    }

    res=SirDBMS_Init((void *) 1, (void *) OutputHandler, (void *)
DisplayHandler, cmdLine);
    if (res != 1)
    {
      res=SirDBMS_ErrorCode();
      printf("error code %ld\n",res);
    }
    printf("DBMS>");
    while (gets(line) != NULL)
    {
        if(strcmp(line,"go") == 0)
        {
          res=SirDBMS_Exec((void *) 1,cmd);
          strcpy(ocmd,cmd);
          strcpy(cmd,"");
          printf("DBMS>");
        }
        else
        {
          if(strcmp(line,"rep") ==0)
          {
```

```
            res=SirDBMS_Exec((void *) 1,ocmd);
            strcpy(cmd,"");
            printf("DBMS>");
          }
          else
          {
            if(strcmp(line,"exit") ==0)
            {
              strcpy(cmd,"");
              break;
            }
            else
            {
              strcat(cmd,line);
          strcat(cmd,"\n");
              printf("DBMS>");
            }
          }
        }
    res=SirDBMS_Exec((void *) 1,cmd);
    res=SirDBMS_Stop((void *) 1);
    return res;
}
```

Compilation & Linkage

The header file (sirapi.h) and link library files (sirdbms.lib and sirsql.lib) are found in the API subdirectory of the SIR installation directory. These files must be used for compiling and linking the API program. Example makefiles for a couple of compilers are also included in this subdirectory.

To run a SIRAPI application the SIR dynamic libraries (eg. sirdbms.dll sirmdr.dll etc) must be in the application's directory or PATH.

General Comments

HOST is a set of FORTRAN routines that enable the SIR/DBMS user to interface directly with a database. These routines provide all the capabilities of a DBMS RETRIEVAL or RETRIEVAL UPDATE. However, schema definition, schema modification, batch data input, retrieval procedures (such as SPSS SAVE FILE or CONDESCRIPTIVE), utility routines and the graphical user interface are not available via HOST. These operations must be performed with DBMS.

HOST is designed for use in those situations in which features are required that are not available in DBMS. It provides a direct interface between a database and a user-written program. For this reason, HOST is particularly appropriate when the user has an existing application package that requires the inclusion of facilities for retrieval of information from a database. HOST also provides facilities that give the SIR/DBMS user the capability to write a program that can process input from many terminals. The user's program controls the operation of the terminals and uses HOST to process one or more databases simultaneously, a feature not available in DBMS.

The HOST package (like any subroutine package) must be used correctly and carefully to avoid damage to the database. Unlike DBMS, the HOST routines cannot protect the database from programmer error, accidental or deliberate. For example, if a HOST memory resident table is modified in any way by the user-written routines during a run, then invalid data, pointers, counts, and other erroneous information may be written to the database. This invalid information may not be detected by either HOST or DBMS until later.

We recommend that applications using HOST be tested thoroughly with a duplicate of the database before the programs are put into production.

Introduction

HOST consists of a set of FORTRAN callable functions written in ANSI standard FORTRAN. The user writes a main program to call the HOST routines which perform the requested retrieval functions. These routines are compiled by the standard FORTRAN compiler for each machine on which SIR/DBMS is available and are maintained as a standard library of FORTRAN routines on each machine.

See additional documentation and examples in the API subdirectory of the SIR installation directory.

There may be two HOST libraries available at your site. One of them, referred to as HOST or regular HOST, is intended for a single user environment. The other one, referred to as concurrent HOST, is intended to work in a multiple-user environment, in conjunction with the MASTER module. The user interface with these two versions is almost identical, such that almost no programming effort is required to switch between them. The small differences are documented below in *Multiple User Support with HOST*.

The HOST routines can be called by a main program written in a language other than FORTRAN (such as PL/1 or C), provided that FORTRAN functions can be called from that language.

If the language can call FORTRAN functions, but cannot test their return values, the user can use a common area called HERROR to get this value. The first variable of the HERROR common block is a R*8 containing the name of the last HOST function called (except for ZCALL which does not touch the variable). Note that this variable does not have the CHARACTER*8 FORTRAN type. The next variable of the HERROR common block is an I*4 which will contain the return code from the last called HOST function. The next variable is an I*4 which is reserved for future use. The length of the HERROR common area is the length of a R*8 plus the length of two I*4's.

The routines provided allow the HOST user to perform all the operations available in DBMS retrieval. Some DBMS commands are provided in a single HOST function call while others have been divided into several function calls.

Every HOST routine is a FORTRAN function which returns a value to the calling application program. If a function executes its task successfully, then it will return a requested value or the value zero or a positive status value to the calling program. If a function encounters an error or problem in performing its task, however, it will return a negative number and not perform the task. The absolute value of the returned value will be the error code detected. A list of all current error codes are listed in Environment appendix A. Some of the negative returned codes do not represent real errors, but some specific conditions such as "no more records found" or "a missing value has been stored".

Writing Programs with HOST

In order to use the routines effectively, the HOST programmer should have experience in writing DBMS Retrieval programs. We recommend that new HOST users write their retrievals in the DBMS Retrieval language and then translate them into HOST function calls. Once the user gains proficiency with HOST, this technique can be dispensed with.

As illustrated by the examples, a HOST program generally has many more statements than its DBMS counterpart. This is partly because each DBMS Retrieval command usually translates into several HOST function calls. HOST programs are also include statements that check the error codes returned by the HOST functions. It is very important to check the HOST error codes. If errors are ignored, the results of subsequent function calls will be unpredictable. If this occurs during a database modification run, the database could be damaged.

To help the programmer write code that performs all of the required returned values testing, a new function, ZCALL, has been introduced. ZCALL helps the programmer write programs that are easy to understand and maintain.

Initialisation

The HOST system is initialised by calling ZSTART. It must be the first HOST call in the user's program. If concurrent HOST is used, ZLOGIN should be called immediately after ZSTART.

One or more calls that open the databases to be used during the run follow. There can be more than one of these calls because, unlike DBMS, HOST allows simultaneous access to more than one database. The HOST routines that open databases are ZORDB (open a random database), and ZOSDB (open a sequential database).

These calls also allow the user to specify default read and write security passwords for each database. The defaults can be overridden by calling ZSECUR.

The Retrieval Stack

A HOST retrieval stack is an internal table containing one entry for each CIR and data record currently being processed. The entries are called blocks. The ordinal of a block in the retrieval stack is referred to as the level of the block. Each block is completely nested within the preceding blocks similar to the nesting in a DBMS retrieval program. In HOST, nesting is extended to include the nesting of databases as well as cases and records.

For example, consider the following schematic DBMS Retrieval with four levels of nesting:

```
RETRIEVAL
. PROCESS CASES ....
.   PROCESS REC 7....
.     CASE IS....
.       RECORD IS 9 ....
.       END RECORD IS
.     END CASE IS
.   END PROCESS REC
. END PROCESS CASES
END RETRIEVAL
```

Within the first level of nesting (PROCESS CASES block), the corresponding HOST retrieval stack would consist of

```
level 1 - CIR
```

Within the fourth level of nesting (RECORD IS block), the HOST retrieval stack would contain

```
level 1 - CIR
level 2 - record type 7
level 3 - CIR
level 4 - record type 9
```

Within a DBMS Retrieval block, the user can only refer to CIR variables or record variables that belong to the block. HOST, on the other hand, allows the user's program to access data at any level from any higher level in the retrieval stack. In the previous example, the user can refer to the CIR variables of level 1 from within levels 2, 3 or 4. Similarly one can refer to record type 7 variables from within levels 3 or 4. This is made possible by the retrieval stack and by the variable descriptors discussed in the next section.

Although the retrieval stack allows the user full access to all preceding levels, there are some operations that can only be performed on a level-by-level basis with HOST. Among these operations are record deletion, getting the next record and termination of record processing. For example, in order to terminate the CASE IS block (level 3 in the previous example) from within the RECORD IS block (level 4), the user must first terminate the RECORD IS block and then the CASE IS block.

In version 2.2, the routines that create blocks (`ZCCNT`, `ZCCNTD`, `ZCGDMY`, `ZCIS`, `ZCISD`, `ZCSAM`, `ZCSAMD`, `ZRCNT`, `ZRCNTD`, `ZRCNTL`, `ZRGDMD`, `ZRGDML`, `ZRGDMY`, `ZRIS`, `ZRISD`, `ZRISL`, `ZRSAM`, `ZRSAMD` and `ZRSAML`) return the stack level of the created block rather than 0 (as they did in previous releases). The level can be used as the fourth piece of a variable descriptor. The level of a case block can also be used as the last argument of `ZRCNTL`, `ZRGDML`, `ZRISL` and `ZRSAML` to make a "record block" belong to a "case block" other than the closest above it. For example, the following structure is possible now:

level 1 - CIR
level 2 - CIR
level 3 - record type 5 "belonging" to the CIR on level 1
level 4 - record type 2 "belonging" to the CIR on level 2

Variable Descriptors

In HOST, database variables are accessed by variable descriptor rather than by name. The purpose of the descriptor is to uniquely identify a variable across all levels of the retrieval stack.

A variable descriptor contains four pieces of identifying information about a variable:

- the database number (assigned by HOST in order of opening)
- the record type number (the CIR is type zero)
- the variable number (assigned automatically in same order as LIST SCHEMA)
- the retrieval stack level

Note: The value of the database number should be left undefined. Opening and closing databases could cause a different sequence of values to be assigned.

Variable descriptors are created in HOST by the routines, ZDESC, ZDESCD, ZDESCM and ZSDESC. Routine ZDESCB breaks a descriptor in its 4 components. ZDESCD requires an explicit database name while ZDESC assumes the database is the one currently in use.

The recommended programming technique is to call ZDESCD at the beginning of the program for each variable to be referenced.

To illustrate, suppose we plan to retrieve data from two databases, CLIENT and PROSPECT. We want to retrieve the variables NAME, PHONE, and BILLED from record type 1 in the CLIENT database. We also want to retrieve NAME and PHONE from the CIR and SALESREP and CLDATE from record type 3 in the PROSPECT database.

The calls to ZDESCD would be placed at the beginning of the HOST program and would look like this in FORTRAN:

```
IERR = ZDESCD (CNAME,  'CLIENT'  , 1,'NAME  '  , 0)
IERR = ZDESCD (CPHONE, 'CLIENT'  , 1,'PHONE'   , 0)
IERR = ZDESCD (CBILLD, 'CLIENT'  , 1,'BILLED'  , 0)
IERR = ZDESCD (PNAME,  'PROSPECT', 0,'NAME'    , 0)
IERR = ZDESCD (PPHONE, 'PROSPECT', 0,'PHONE'   , 0)
IERR = ZDESCD (PSLSRP, 'PROSPECT', 3,'SALESREP', 0)
IERR = ZDESCD (PDATE,  'PROSPECT', 3,'CLDATE'  , 0)
```

The FORTRAN variables CNAME, CPHONE, CBILLD, etc. contain the variable descriptors for the corresponding database variables NAME, PHONE, BILLED, etc. Once the descriptors are defined, they are used in all subsequent HOST calls. For example, to retrieve the value of the CIR variable NAME in the PROSPECT database, the appropriate HOST call would be

```
IERR = ZRCTST (PNAME, PRONAM, 25)
```

`ZRCTST` transfers 25 characters of the CIR variable `NAME` (whose descriptor is stored in `PNAME`) into the `FORTRAN` character variable `PRONAM`.

Case and Record Processing

Case and record processing have been extended in HOST so that cases can be processed with the AFTER, FROM, THRU, UNTIL, and WITH clauses and records can be processed with the COUNT and SAMPLE options.

Case and record processing in HOST involves several steps:

1. Call the appropriate case or record-initialisation routine.
2. If necessary, call one or more key creation and key definition routines.
3. Use ZCNEXT or ZRNEXT as the first statement of the case or record-processing loop. These routines get a case or record from the database and make it available for use.
4. To loop through several cases or records, return to the start of the loop, ZCNEXT or ZRNEXT. These routines return a negative error code (-4xxx) when there are no more cases or records left.
5. Call ZCEXIT or ZREXIT when the loop is terminated.

To illustrate these concepts further, consider the following DBMS Retrieval commands:

```
.  PROCESS CASES ALL
.    WRITE CASEID NAME ADDRESS
.  END PROCESS CASES
```

The corresponding HOST program would look like this in FORTRAN:

```
      IERR = ZCCNT (-1, 1, 1)
100   IERR = ZCNEXT (0)
      IF (IERR .EQ. -4001 .OR. IERR .EQ. -4002) GO TO 200

....    (retrieve and print CASEID, NAME and ADDRESS)

      GO TO 100
200   IERR = ZCEXIT (DUM)
```

Errors -4001 and -4002 indicate that there are no more cases left to be processed.

Now suppose we want to run the same retrieval, but only for those cases whose case-ids lie in the range 5000 to 5999. We must add calls to the HOST routines that create and define the key values for the cases. The HOST program would then be written as follows:

```
      IERR = ZCCNT (-1, 1, 1)
      IERR = ZFROM (0)
      IERR = ZINTKY (5000)
      IERR = ZTHRU (0)
      IERR = ZINTKY (5999)
100   IERR = ZCNEXT (0)
```

```
      IF (IERR .EQ. -4001 .OR. IERR .EQ. -4002) GO TO 200

.....  (retrieve and print CASEID, NAME and ADDRESS)

      GO TO 100
200   IERR = ZCEXIT (0)
```

Note that the key creation (`ZFROM`, `ZTHRU`) and key definition (`ZINTKY`) routines are placed immediately after the initialisation routine (`ZCCNT`) and before the actual start of the loop (`ZCNEXT`). Record processing in `HOST` is analogous to case-processing. For example, the following `PROCESS REC` loop:

```
.  PROCESS REC 3, WITH (1982, 100)
.    WRITE 'DEPT YTD TOTAL IS', YTDSALE
.  END PROCESS REC
```

might look like this in `FORTRAN`:

```
      IERR = ZRCNT (3, -1, 1, 1)
      IERR = ZWITH (0)
      IERR = ZINTKY (1982)
      IERR = ZINTKY (100)
100   IERR = ZRNEXT (0)
      IF (IERR .EQ. -4001 .OR. IERR.EQ. -4002) GO TO 200

..... (retrieve and print value of YTDSALE)

      GO TO 100
200   CALL ZREXIT (0)
```

Caseless Database Processing

If a database is caseless, some of the `HOST` routines cannot be used when accessing that database. If you attempt to use these routines with a caseless database, the routines will return an error code of -2053. No case block should ever be created. The following is a list of the routines that can only be used with "case" databases:

```
ZCCNT  ZCCNTD ZCDEL  ZCEXIT ZCFIND ZCGDMY
ZCIS   ZCISD  ZCLOCK ZCNEXT ZCRDMY ZCREST
ZCSAM  ZCSAMD ZCWRIT ZNOR   ZNORD
```

If you want to find out if a database is caseless or not, your program should call `ZNSIDS` with the second argument 0. `ZNSIDS` returns the number of sort ids in the database for the specified record type, which is 0 for a caseless database. (Actually it returns the number of sort ids in the CIR for the caseless database).

`ZNCASE` also returns the number of cases in the specified database, and it will always return 0 for a caseless database. However, it cannot be used as a test for case structure in a database since it will return 0 for any empty database, and any caseless database regardless of the number of records.

Multiple Database Mode in HOST

As previously stated, HOST allows the user to access data from several databases simultaneously. This capability is not available in DBMS Retrieval.

A HOST program is in multiple-database mode when it contains more than one call to the database initialisation routines (ZORDB, ZOSDB). Note that multiple-database mode remains in effect even if all of the databases have been closed.

In multiple database mode, the routines that initialise case and record processing also establish the identity of the current database. These routines are described in the section called *Database Switching*.

All of the records processed within a case loop are assumed to belong to the current database.

In *Variable Descriptors*, we saw that variable descriptors allow the user to access data at higher levels of the retrieval stack from within lower levels. This concept also applies to stack levels from different databases.

For example, suppose the retrieval stack contains the following levels:

```
level 1 - database DB1, CIR
level 2 - database DB1, record type 5
level 3 - database DB2, CIR
level 4 - database DB2, record type 8
```

and we have defined the descriptor for the database variable, AGE, from record type 5 in DB1 with the call:

```
      IERR = ZDESCD (XAGE,'DB1     ',5, 'AGE     ',0)
```

Then, the database variable AGE can be retrieved at any level of the retrieval stack (except level 1) by using its descriptor stored in XAGE.

If the stack contains two process rec 1 blocks, then stack level 0 indicates the innermost process rec.

Multiple User Support with HOST

As mentioned in the Introduction, in addition to providing access to databases as a stand-alone program, HOST also provides access to the databases by multiple users. To use concurrent HOST, a MASTER process must be running and the user program must login to that MASTER. To do this, a typical application program will call ZLOGIN immediately after calling ZSTART. If the user program does not call ZLOGIN with the name of the MASTER explicitly, the system will do it, using the name of the default MASTER. A given application program can be logged-in to only one MASTER at a time. For regular HOST programs, the calling of ZLOGIN is optional.

To determine from within a program which of the two versions of HOST is used, the function ZVERS can be called. It will return 0 for regular HOST and 1 for concurrent HOST.

HOST 2.2 Lock Types

In HOST 2.2, (both regular and concurrent) there are 6 lock types defined:

```
Lock-type number       Lock type
0 or 14                protected read
1 or 16                exclusive
11                     null
12                     concurrent read
13                     concurrent write
15                     protected write
```

The locks ensure data integrity. They prevent other programs from updating records that you are updating. Several HOST programs feature the option to set and test the locks.

Routine Formal Parameters

When using the `HOST` routines great care must be taken to insure that the formal parameters match in both number and type. TABLE 1 below indicates the various argument types used by HOST. See the documentation in the API subdirectory for information and examples for the exact declarations and representations needed for each data type.

TABLE 1

ABBREV	TYPE	DESCRIPTION
B*n	BYTE	This is a CHARACTER*N structure as defined by the FORTRAN 77 compiler on your machine. It must be of the proper size to hold the character strings to be passed to or from it.
D*8	DESCRIPTOR	This is an (8 byte) area containing a set of values that uniquely describe a data variable (i.e. variable number, record type number, database number, stack location). This data type may be stored as a REAL*8 on some machines; in this case care should be taken in transferring descriptors because it is not a REAL*8 value but an 8 byte bit pattern and no floating operations such as normalisation should be performed.
I*4	INTEGER	This is I*4 as defined in Machine Specifics documentation.(It might be 4 or 8 bytes long.)
N*8	NAME	This is a CHARACTER*8 created by a FORTRAN 77 compiler.
R*n	REAL	This is R*4 or R*8 as defined in your Machine Dependencies documentation.

Overview of the HOST Subroutines

A list of the standard routines available within HOST are listed below. Certain machines have additional routines available usually dealing with additional type conversions and other utilities to ease the use of the HOST package. See the documentation in the API subdirectory for more information on these routines.

Note that all of the HOST functions are of type INTEGER*4 regardless of the first letter of the names. Declare them as type INTEGER*4 in each routine that calls them. Also, the types of the function arguments do not correspond to standard FORTRAN usage (i.e. first letter I through N is type INTEGER, otherwise REAL).

As already stated, these functions may return 0, a required value, a positive status value, or a negative error code. In Chapter 3, the return values for the functions will be documented only if they have a specific significance. If they are not documented, the assumption is that the function will return a positive or zero value for success and a negative value for failure.

Initialisation Routines

ZSTART Initialises the HOST system.

ZLOGIN login the current process into a specified MASTER.

ZSECUR Specifies security passwords for current stream.

ZORDB Initialises a specified random format database for use.

ZOSDB Initialises a specified sequential format database for use.

Control Routines

ZUSER Allows the switching of the current retrieval stack from one stream to another.

ZCALL Call HOST functions and test their return code.

Termination Routines

ZENDDB Terminates the use of a specified database. It closes and return the files for use by other jobs. In order to use the database again it must be reinitialised.

ZCLEAR Terminates all levels in the retrieval stack of a specified stream.

ZEND Terminates the HOST run. It cleans up tables and checks that the databases have been properly closed and all internal operations are completed. If any databases are still open, ZEND attempts to close them.

Case Processing Routines

ZCCNT Initialises a "PROCESS CASE" block with either the "ALL" or "COUNT" option.

ZCCNTD Initialises a "PROCESS CASE" block with either the "ALL" or "COUNT" option. It also provides the capability of changing databases.

ZCGDMY Initialise a case-processing block of an undefined type (dummy).It also provides the capability of changing databases.

ZCIS Initialises a "CASE IS" block.

ZCISD Initialises a "CASE IS" block. It also provides the capability of changing databases.

ZCSAM Initialises a "PROCESS CASE" block with a "SAMPLE" option.

ZCSAMD Initialises a "PROCESS CASE" block with a "SAMPLE" option. It also provides the capability of changing databases.

ZCREST Restores the common variables in the innermost case processing block.

ZCRDMY Terminates the current case-processing level and resets it to a dummy.

ZCFIND Finds a case, after a dummy case-processing block was initialised and some CIR variables specified.

ZCFRST Gets the first case in a case-processing block.

ZCLAST Gets the last case in a case-processing block.

ZCNEXT Gets the next case in a case-processing block.

ZCPREV Gets the previous case in a case-processing block.

ZCLOCK Returns the lock type of a case-processing block.

ZCDEL Deletes the current case in the innermost case-processing block.

ZCWRIT Writes the current innermost CIR to the database as a permanent change.

`ZCEXIT` Leaves the current case-processing block and pops the retrieval stack back one level of retrieval nesting.

Record Processing Routines

`ZRCNT` Initialises a "`PROCESS RECORD`" block with either the "`ALL`" or "`COUNT`" option.

`ZRCNTD` Initialises a "`PROCESS RECORD`" block with either the "`ALL`" or "`COUNT`" option. It also provides the capability of changing databases.

`ZRCNTL` Initialises a "`PROCESS RECORD`" block with either the "`ALL`" or "`COUNT`" option, belonging to a CIR on a specified level.

`ZRGDMY` Initialise a record-processing block of an undefined type (dummy). It also provides the capability of changing databases.

`ZRGDMD` Initialise a record-processing block of an undefined type (dummy). It also provides the capability of changing databases.

`ZRGDML` Initialises a record-processing of an undefined type (dummy) block, belonging to a CIR on a specified level.

`ZRIS` Initialises a "`RECORD IS`" block.

`ZRISD` Initialises a "`RECORD IS`" block. It also provides the capability of changing databases.

`ZRISL` Initialises a "`RECORD IS`" block belonging to a CIR on a specified level.

`ZRSAM` Initialises a "`PROCESS RECORD`" block with a "`SAMPLE`" option.

`ZRSAMD` Initialises a "`PROCESS RECORD`" block with a "`SAMPLE`" option. It also provides the capability of changing databases.

`ZRSAML` Initialises a "`PROCESS RECORD`" block with a "`SAMPLE`" option, belonging to a CIR on a specified level.

`ZRREST` Restores the record variables in the innermost record processing block.

`ZRRDMY` Terminates the current record-processing level and resets it to dummy.

`ZRFIND` Finds a record, after a dummy record-processing block was initialised and some record variables specified.

`ZRFRST` Gets the first case in a record-processing block.

`ZRLAST` Gets the last case in a record-processing block.

`ZRNEXT` Gets the next case in a record-processing block.

`ZRPREV` Gets the previous case in a record-processing block.

`ZRLOCK` Returns the lock type of a record-processing block.

`ZRDEL` Deletes the current record in the innermost record-processing block.

`ZRWRIT` Writes the current innermost record to the database as a permanent change.

`ZREXIT` Leaves the current record-block and pops the retrieval stack one level of retrieval nesting.

Key Creation Routines

`ZAFTER` Initialises the creation of a key for either case or record processing levels. It starts up a key of the form of the "`AFTER`" keyword on the `PROCESS` statement in `DBMS`.

`ZBEGIN` Initialises the creation of a key for either case or record processing levels. It starts up a key of an yet undefined form.

`ZFROM` Initialises the creation of a key for either case or record processing levels. It starts up a key of the form of the "`FROM`" keyword on the `PROCESS` statement in `DBMS`.

`ZTHRU` Initialises the creation of a key for either case or record processing levels. It starts up a key of the form of the "`THRU`" keyword on the `PROCESS` statement in `DBMS`.

`ZUNTIL` Initialises the creation of a key for either case or record processing levels. It starts up a key of the form of the "`UNTIL`" keyword on the `PROCESS` statement in `DBMS`.

`ZWITH` Initialises the creation of a key for either case or record processing levels. It starts up a key of the form of the "`WITH`" keyword on the `PROCESS` statement in `DBMS`.

Key Definition Routines

`ZDTTKY` Moves a date string into the next sort-id position of the current key.

`ZFPTKY` Moves a real value into the next sort-id position of the current key.

`ZINTKY` Moves an integer value into the next sort-id position of the current key.

`ZSTTKY` Moves a character string into the next sort-id position of the current key.

`ZTMTKY` Moves a time string into the next sort-id position of the current key.

Variable Modification Routines

`ZBLTRC` Stores a blank missing value in a variable.

`ZDTTRC` Moves a date string into a specified variable descriptor.

`ZFPTRC` Moves a real value into a specified variable descriptor.

`ZINTRC` Moves an integer value into a specified variable descriptor.

`ZMSTRC` Transfers a missing/undefined value to a specified variable descriptor.

`ZRCTRC` Transfers the value from one variable descriptor to another variable descriptor.

`ZSTTRC` Moves a string value into a specified variable descriptor.

`ZTMTRC` Moves a time string into a specified variable descriptor.

Variable Retrieval Routines

`ZRCTDT` Returns the value of a specified date variable descriptor.

`ZRCTFP` Returns the value of a specified real variable descriptor.

`ZRCTIN` Returns the value of a specified integer variable descriptor.

`ZRCTST` Returns the value of a specified string variable descriptor.

`ZRCTTM` Returns the value of a specified time variable descriptor.

General Routines

`ZVERS` Returns the version and revision numbers of HOST.

`ZOPEN` Returns an indication of whether a specified database is currently available for use or not.

`ZUPLEV` Returns the current update level of a database.

`ZSECLV` Returns the current security levels for a specified database.

`ZLCKRT` Sets/returns the record type lock for a specific record type.

`ZNCASE` Returns the number of cases in the database.

`ZNRECS` Returns the number of records of a specific type that are currently in the database.

`ZNEW` Returns an indication of whether the last CIR/record accessed was a new CIR/record or an existing CIR/record.

`ZNOR` Returns the number of records of a specific type that are currently in the innermost CIR.

`ZNORD` Returns the number of records of a specific type that are currently in the innermost CIR of a specified database.

`ZRNAMD` Returns the record name for a record number.

`ZRNUM` Returns the record number for a record name.

`ZRNUMD` Returns the record number for a record name in a specified database.

`ZNVARS` Returns the number of variables in a specific record type.

`ZNSIDS` Returns the number of sort ids in the key of a record of a specific type.

`ZVARLB` Returns the variable label of a variable specified by descriptor.

`ZVNAME` Returns the variable name for a variable specified by descriptor.

`ZVTYPE` Returns the storage type of the variable specified by descriptor.

`ZLABLN` Returns the value label for a value of a specified numeric variable descriptor.

`ZLABLS` Returns the value label for a value of a specified string variable descriptor.

`ZMSLAB` Returns the value label for a missing value.

`ZDESC` Builds the variable descriptor from a variable description.

`ZDESCD` Builds the variable descriptor from a variable description for a variable in a different database.

`ZSDESC` Returns the variable descriptor corresponding to the nth sortid of the specified record type.

`ZDESCB` Breaks a descriptor in its four components.

ZDESCM Makes a descriptor out of its four components.

ZEXIT Performs the equivalent of ZCEXIT for a case-processing block and of ZREXIT for a record-processing block.

Utility Routines

ZATTR Perform equivalent to DBMS ATTRIBUTE command.

ZCACHE Activates/deactivates the caching and sets/reads various caching parameters.

ZOPT Set or return various system options.

ZERMSG Returns a text string describing a specific numbered error. Optionally it can also file the error message in the job log.

ZDTXIN Converts a date string into a date integer.

ZINXDT Converts a date integer into a date string.

ZTMXIN Converts a time string into a time integer.

ZINXTM Converts a time integer into a time string.

ZTIME Returns the current date and time as julian integers.

Advanced Key Definition Routines

ZDTXKY Moves a date string into the next sort-id position of the current key.

ZFPXKY Moves a real value into the next sort-id position of the current key.

ZINXKY Moves an integer value into the next sort-id position of the current key.

ZSTXKY Moves a character string into the next sort-id position of the current key.

ZTMXKY Moves a time string into the next sort-id position of the current key.

Advanced Data Modification Routines

ZDTXRC Moves a date string into a specified variable descriptor.

ZFPXRC Moves a real value into a specified variable descriptor.

ZINXRC Moves an integer value into a specified variable descriptor.

`ZSTXRC` Moves a string value into a specified variable descriptor.

`ZTMXRC` Moves a time string into a specified variable descriptor.

Advanced Data Retrieval Routines

`ZRCXDT` Returns the value of a specified date variable descriptor.

`ZRCXFP` Returns the value of a specified real variable descriptor.

`ZRCXIN` Returns the value of a specified integer variable descriptor.

`ZRCXST` Returns the value of a specified string variable descriptor.

`ZRCXTM` Returns the value of a specified time variable descriptor.

Database Switching

In a multiple database environment, it is important to know which database is being used at any given time. This database is referred to as the "current database". While most of the routines refer to the current database, some routines can switch databases, making another database the current one. The following list describes how the `HOST` routines manipulate the databases. (The term "specified" refers to an item in the argument list of the subroutine):

`ZCCNTD`, `ZCGDMY`, `ZCISD`, `ZCSAMD`, `ZRCNTD`, `ZRGDMD`, `ZRISD`, `ZRSAMD` explicitly make the specified database be the current database.

`ZRCNTL`, `ZRGDML`, `ZRISL`, `ZRSAML` make the database at the specified level be the current database.

`ZEXIT`, `ZCEXIT`, `ZEXIT` pop back 1 level from the stack and make the database at the new level be the current database.

`ZORDB`, `ZOSDB` do not switch databases. However, if there is no current database, the specified database is made current.

`ZENDDB` does not switch databases. However, if the specified database is the current one, the "next" open database is made current.

ZAFTER

```
ZAFTER, Start Creation of an AFTER Key
ARGS:   DUMMY
DESC:   ZAFTER starts the creation of a DBMS key. It creates
        an "AFTER" key.  That is, a key which will be used
        to select all CIR/record's whose key comes after the
        key that is currently being defined.  It is normally
        called after one of the case/record level
        initialisation routines in order to initialise the
        key selection options for case or record loops.
        Following a call to ZAFTER, key definitions routines
        are called to enter values into the key one at a
        time.  For a case key only one call is made to enter
        the value of the case id. For record keys, the case
        id is assumed to be the same as the last CIR
        retrieved or restored. Therefore, only the record
        sort ids have to be inserted into the key.  The
        sort-ids must be entered in the order of their
        appearance in the key being created.
ENTRY:  DUMMY (I * 4 ) Dummy argument needed to make a
        syntactically correct FORTRAN function. Should
        always be 0.
EXIT:   None.
```

ZATTR

```
ZATTR,  Equivalence Long and Short Filenames
ARGS:   FILENM,STRING,STRLEN
DESC:   ZATTR performs the equivalent of the ATTRIBUTE
        command in DBMS.  Currently it allows the
        equivalencing of a FILENAME (short 1-8 character
        name) and a long (quoted filename).  After ZATTR is
        called use of the short name will reference the file
        specified by the long name.
ENTRY:  FILENM (N * 8) Short name or "ldi" specification of
                       ATTRIBUTE command.
        STRING (B * n) Long name or string appearing within
                       quote marks of DSN = subparameter of
                       ATTRIBUTE command (without quotes).
        STRLEN (I * 4) Number of characters in argument STRING.
EXIT:   None.
RETURN: The location of the entry in the ATTRIBUTE table
        (small positive integer).
```

ZBEGIN

```
ZBEGIN, Start Creation of Starting Key
ARGS:   DUMMY
DESC:   ZBEGIN is called after a new level in a retrieval
        block is started, and prior to retrieving the first
        CIR/record.  It allows the user to start creating a
        key which will be used as the first key to be
        processed at the level.
ENTRY:  DUMMY (I * 4 ) Dummy argument needed to make a
                       syntactically correct FORTRAN
                       function. Should always be 0.
EXIT:   None.
```

ZBLTRC

```
ZBLTRC, Store Blank in Variable if Blank Is Missing or Undefined
ARGS:   VDESC
DESC:   ZBLTRC stores a blank into a variable.  If the blank
        is a missing value, then it is stored as the
        respective missing value.  For other values, an
        error is returned.
ENTRY:  VDESC (D * 8) The descriptor for the variable to be set
                      to blank.
EXIT:   None.
```

ZCACHE

```
ZCACHE, Set and Retrieve Cache System Control Parameters
ARGS:   FNC,VAL
DESC:   ZCACHE is used to set and retrieve cache system-control
        parameters.
ENTRY:  FNC (I * 4)  The control function code.
        VAL (I * 4)  The new value or dummy.
EXIT:   None.
RETURN: Negative if error, if not as described below:
        FNC   VAL     Return code   Description
         1     0            0      Disable cache
         1     1            0      Enable cache
         2     n            0      Set table space = n
         3     n            0      Set number of buffers= n
         5     0            0      Set "write when" mode
         5     1            0      Set "write thru" mode
        11     x            0      Return for cache disabled
        11     x            1      Return for cache enabled
        12     x            n      Return table space
        13     x            n      Return number of buffers
        15     x            0      Return for "write when" mode
        15     x            1      Return for "write thru" mode
```

ZCALL

```
ZCALL,  Call HOST Function and Check Its Return Value
ARGS:   ZFUNC, PRTFL, ENDFL, LBLNO, CODE1, CODE2
DESC:   ZCALL enables the programmer to write a more concise
        and easier to maintain program by performing the
        testing of the return codes and the eventual error
        processing. First, ZCALL calls ZFUNC.  If ZFUNC
        returns a positive value, zero or a negative value
        that is between CODE1 and CODE2, ZCALL returns
        immediately.  If ZCALL does not return immediately,
        the following action is taken: PRTFL is 0, nothing
        is printed in the log file.  PRTFL is 1, a brief
        message is printed that indicates the subroutine
        name, the error code, and the statement label LBLNO.
        If it is 2, the short message is printed followed by
        the complete error message on the next line.  If
        ENDFL is 0, ZCALL returns. If it is -1, the full
        HOST system is shutdown but the subroutine returns.
        If it is -2, the full system is shutdown and the
        subroutine exits directly.  If it is n(positive
        integer), the current stream (user) is shutdown, the
        system switches to user n and the subroutine
        returns.
ENTRY:  ZFUNC (I * 4)  Z function to call, with all its arguments.
        PRTFL (I * 4)  Print flag, controls if and how a message is
                       printed.
        ENDFL (I * 4)  End flag, controls the flow of control after
                       an error has occurred
        LBLNO (I * 4)  Statement label number.
        CODE1 (I * 4)  Error code (Negative 4 digit number or 0).
        CODE2 (I * 4)  Error code (Negative 4 digit number or 0).
EXIT:   None.
RETURN: Return value of ZFUNC.
```

ZCCNT

```
ZCCNT,  Initialise COUNT Case-processing Block for Current Database
ARGS:   TCASES,INDEX,START
DESC:   ZCCNT is the first of a series of routines that can
        be called in order to create "PROCESS CASE" block
        with the COUNT option. The purpose of this call is
        to initialise the retrieval stack with the required
        information.  The arguments correspond to the
        arguments on the COUNT = clause of the PROCESS CASE
        command in RETRIEVAL.  ZCCNT can also be used to
        process ALL cases by setting INDEX and START to 1
        and TCASES to -1.
ENTRY:  TCASES(I * 4) Total number of cases to be processed.
        INDEX( I * 4) Every INDEX the case will be processed.
        START (I * 4) Starting with the START th case in the
                      database.
EXIT:   None.
RETURN: Stack level of the newly created block, negative if
        error.
```

ZCCNTD

```
ZCCNTD, Set Current Database and Initialise COUNT Case-processing Block
ARGS:   TCASES,INDEX,START,DBNAME
DESC:   ZCCNTD is called to make DBNAME the current database
        and to start creating a "PROCESS CASE" block with
        the COUNT option.  The purpose of this call is to
        initialise the retrieval stack with the required
        information.  The arguments correspond to the
        arguments on the COUNT = clause of the PROCESS CASE
        command in RETRIEVAL. the database.  ZCCNTD can also
        be used to process ALL cases by setting INDEX and
        START to 1 and TCASES to -1.
ENTRY:  TCASES(I * 4) Total number of cases to be processed.
        INDEX (I * 4) Every INDEX th case will be processed.
        START (I * 4) Starting with the START the case in
                      the database.
        DBNAME(N * 8) The database name to be processed by
                      this case loop.
EXIT:   None.
RETURN: Stack level of the newly created block, negative if error.
```

ZCDEL

```
ZCDEL,  Delete Current Case from Database
ARGS:   DUMMY
DESC:   ZCDEL is called after a CIR is retrieved to delete
        the current CIR and all its associated data records.
ENTRY:  DUMMY (I * 4) Dummy argument needed to make a
                      syntactically correct FORTRAN
                      function. Should always be 0.
EXIT:   None.
```

ZCEXIT

```
ZCEXIT, Terminate Case Processing Level
ARGS:   OLDSEED
DESC:   ZCEXIT is called to terminate a case level and pop
        back one level in the retrieval stack.
ENTRY:  None.
EXIT:   OLDSEED(I * 4) The current value of the seed is
                       returned to this argument if this was a
                       PROCESS loop with the SAMPLE option.
                       Otherwise the value is undefined.
```

ZCFIND

```
ZCFIND, Find Case with Given Key
ARGS:   OPT, LFLAG, DIRECT, BEGIN
DESC:   ZCFIND finds an existent case or creates a new case
        with a previously specified key.  It is called after
        routine ZCGDMY or ZCRDMY has created a dummy block
        and after putting values into the caseid variable in
        CIR and ( if OPT is 1 ) in other common variables.
        When ZCFIND is executed, the value of the caseid is
        used to create the case key.  If the caseid variable
        is undefined, then the dummy block is converted into
        a PROCESS CASE ALL block and the next or previous
        CIR is read.  Otherwise, the dummy block is
        converted into a CASE IS block and: a)  if OPT is 1,
        then the undefined values are updated to the values
        read from the found CIR or b) if OPT is 2 the full
        current CIR is replaced by the read CIR.
ENTRY:  OPT (I * 4)   Controls the update/replace of the CIR.
        LFLAG (I * 4)  Lock flag.
        DIRECT (I 4)  If PROCESS CASE then use 1 to get the
                      next case and -1 to get the previous
                      case.
        BEGIN (I * 4)  Should be 0 to set the range to all
                      cases (starting with the first case in
                      the database) and I to convert to a
                      PROCESS CASES ALL block (if caseid is
                      undefined) or CASE IS block (if
                      caseid is defined).
EXIT:   None.
RETURN: -4001 if CIR not found;
        -3026 if CIR is found but is incompatible locked;
        +0003 if CIR is found and has a compatible lock;
        +0004 if CIR is found and available for CASE IS;
        +0000 if CIR is found and available for PROCESS
              CASE;
        negative if error.
```

ZCFRST

```
ZCFRST, Start-up Case Level Block and Get First CIR
ARGS:   LFLAG
DESC:   ZCFRST is called after case-block initialisation and
        all key creation, and after all key-definition
        routines.  It starts-up the case block and gets the
        first CIR that meets all of the selection options
        previously specified.  No other case-level function
        can be called until this block is successfully
        executed.  After ZCFRST is executed, no further key
        definitions may be made.  The system checks that the
        current lock flag is compatible with LFLAG and if it
        is, LFLAG becomes the new lock.
ENTRY:  LFLAG(I * 4)            The lock flag.
EXIT:   None.
```

ZCGDMY

```
ZCGDMY, Initialise Dummy Case-level Processing
ARGS:   DBNAME
DESC:   ZCGDMY starts up a case block without accessing the
        database.  After ZCGDMY has been called and values
        put in the CIR, ZCFIND should be called to create
        the key and find the CIR.  If DBNAME is not all
        blanks, it is made the current database.
ENTRY:  DBNAME ( N * 8 ) Database name (all blanks means use
                         current database
EXIT:   None.
RETURN: Stack level of the newly created block, negative if error.
```

ZCIS

```
ZCIS,   Initialise CASE IS Block for Current Database
ARGS:   NEW,OLD
DESC:   ZCIS is the first of a series of routines that can
        be called in order to create "CASE IS" block.  The
        purpose of this call is to initialise the retrieval
        stack with the required information.
ENTRY:  NEW (I * 4) It is 1 if a new case can be created by
                    this level, otherwise it is 0. The
                    database must have been opened for
                    update to allow a value of 1.
        OLD (I * 4) It is 1 if an old case can be accessed
                    by this level, otherwise it is 0.
EXIT:   None.
RETURN: Stack level of the newly created block, negative if error.
```

ZCISD

```
ZCISD,  Set Current Database and Initialise CASE IS Block
ARGS:   NEW,OLD,DBNAME
DESC:   ZCISD is called to make DBNAME the current database
        and to start creating a "CASE IS" block.  The
        purpose of this call is to initialise the retrieval
        stack with the required information.
ENTRY:  NEW (I * 4) It is 1 if a new case can be created by
                    this level, otherwise it is 0. The
                    database must have been opened for
                    update to allow a value of 1.
        OLD (I * 4) It is I if an old case can be accessed
                    by this level, otherwise it is 0.
        DBNAME(N*8) The database name to be processed by the
                    level being created.
EXIT:   None.
RETURN: Stack location of the new block, negative if error.
```

ZCLAST

```
ZCLAST, Start Case-level Block and Get Last CIR
ARGS:   LFLAG
DESC:   ZCLAST is called after case-block initialisation and
        all key creation, and after all key-definition
        routines.  It starts-up the case block and gets the
        last CIR that meets all of the selection options
        previously specified.  No other case-level function
        can be called until this block is successfully
        executed.  After ZCLAST is executed, no further key
        definitions may be made.  The system checks that the
        current lock flag is compatible with LFLAG and if it
        is, LFLAG becomes the new lock.
ENTRY:  LFLAG (I * 4)   The lock flag.
EXIT:   None.
```

ZCLEAR

```
ZCLEAR, Clear Retrieval Stack for a Stream
ARGS:   USERNO
DESC:   ZCLEAR calls ZREXIT and ZCEXIT as often as needed
        and in the proper order, in order to clear all
        levels in the retrieval stack for the specified
        user.  ZCLEAR should be used to clear the stack when
        the current position is unknown.
ENTRY:  USERNO(I * 4) The stream number whose retrieval stack
        should be cleared.
EXIT:   None.
```

ZCLOCK

```
ZCLOCK, Return The Lock Type of the Case Level from
        Execution Stack
ARGS:   LFLAG
DESC:   ZCLOCK returns the lock type of the case level to
        which the innermost block in the execution stack
        belongs.
ENTRY:  None.
EXIT:   LFLAG (I * 4)   Lock type.
RETURN: 0 if the level is not write-locked. 1 if the level is write-
locked.
```

ZCNEXT

```
ZCNEXT, Get Next Case for Current Level
ARGS:   LFLAG
DESC:   ZCNEXT is called after case-block initialisation and
        all key creation, and after all key-definition
        routines.  It starts-up the case block and gets the
        next CIR that meets all of the selection options
        previously specified.  No other case-level function
        can be called until this block is successfully
        executed.  After ZCNF-XT is executed, no further key
        definitions may be made.  The system checks that the
        current lock flag is compatible with LFLAG and if it
        is, LFLAG becomes the new lock.
ENTRY:  LFLAG (I * 4)   The lock flag.
EXIT:   None.
```

ZCPREV

```
ZCPREV, Start-up Case Level Block and Get Previous CIR
ARGS:   LFLAG
DESC:   ZCPREV is called after case-block initialisation and
        all key creation, and after all key definition
        routines.  It starts-up the case block and gets the
        previous CIR that meets all of the selection options
        previously specified.  No other case-level function
        can be called until this block is successfully
        executed.  After ZCPREV is executed, no further key
        definitions may be made.  The system checks that the
        current lock flag is compatible with LFLAG and if it
        is, LFLAG becomes the new lock.
ENTRY:  LFIAG (I * 4)   The lock flag.
EXIT:   None.
```

ZCRDMY

```
ZCRDMY, Terminate Case-processing Level and Reset to
Dummy
ARGS:   DUMMY
DESC:   ZCRDMY terminates the processing of the current CIR,
        rewrites it to the database if necessary, and then
        resets the block to dummy.
ENTRY:  DUMMY (I * 4 ) Dummy argument needed to make a
                       syntactically correct FORTRAN
                       function. Should always be 0.
EXIT:   None.
```

ZCREST

```
ZCREST, Restore CIR from Database and Reset Lock Type
ARGS:   LFLAG
DESC:   ZCREST is called to replace the values of the common
        variables in the retrieval stack with the values of
        the common variables in the database.  Potentially,
        it can change the lock type of the CIR.
ENTRY:  LFLAG (I * 4)     The lock flag.
EXIT:   None.
```

ZCSAM

```
ZCSAM,  Initialise SAMPLE Case-processing Block
ARGS:   SAMPLE,SEED
DESC:   ZCSAM is the first of a series of routines that can
        be called in order to create "PROCESS CASE" block
        with the SAMPLE option.
ENTRY:  SAMPLE(R * 4) Sample size (SAMPLE).
        SEED (I * 4) Starting seed for random-number
                     generator. Same seed always produces the
                     same random selection sequence. Any odd
                     value can be used for the seed. See
                     routine ZCEXIT for obtaining the value
                     of the seed after the loop is finished.
EXIT:   None.
RETURN: Stack location of the new block, negative if error.
```

ZCSAMD

```
ZCSAMD, Set Current Database and Initialise a SAMPLE
        Case-processing Block
ARGS:   SAMPLE,SEED,DBNAME
DESC:   ZCSAMD is called to make DBNAME the current database
        and to start creating a "PROCESS CASE" block with
        the SAMPLE option.  The purpose of this call is to
        initialise the retrieval stack with the required
        information.
ENTRY:  SAMPLE(R * 4) Sample size (SAMPLE).
        SEED (I * 4)  Starting seed for random-number
                      generator. Same seed always produces
                      the same random selection sequence.
                      Any odd value can be used for the
                      seed. See routine ZCEXIT for obtaining
                      the value of the seed after the loop
                      is finished.
        DBNAME(N * 8) The database name to be processed by
                      this case block (all blanks means use
                      current database).
EXIT:   None.
RETURN: Stack location of the new block, negative if error.
```

ZCWRIT

```
ZCWRIT, Replace Modified CIR on Database and Lock of
        Block
ARGS:   LFLAG
DESC:   ZCNWIT is called to replace the database version of
        the CIR with the current version in the retrieval
        stack.  This process is performed automatically by
        the HOST routines when the next CIR is retrieved or
        the case level is terminated.  However the user may
        want to write the modified CIR to the database and
        change the lock for further processing.
ENTRY:  LFLAG (I * 4)     Lock flag.
EXIT:   None.
```

ZDESC

```
ZDESC,  Create Variable Descriptor
ARGS:   VDESC,RECTYP,VRNAME,LEVEL
DESC:   ZDESC builds-up a variable's descriptor. (Refer to
        the Machine Specifics documentation for
        additional information on descriptors.) ZDESC can be
        called once prior to entering a loop which
        references a variable.  This avoids the necessity of
        creating the variable descriptor for each loop
        iteration.  Unlike ZDESCD, the database name is not
        required.
ENTRY:  RECTYP(I * 4) The record type number to which the
                      variable belongs. Common variables are
                      indicated by setting this argument to
                      zero.
        VRNAME(N*8) The variable name.
        LEVEL (I * 4) Level in the retrieval stack where the
                      CIR/data record can be found. A zero
                      value indicates that when the descriptor
                      is used, the system should start with
                      the innermost level in the retrieval
                      stack and search outwards for the first
                      level which matches the database and
                      record type specified within the
                      descriptor. A negative value indicates
                      that the record is LEVEL levels out from
                      the innermost level. A positive value
                      indicates that the record is LEVEL
                      levels deep in the retrieval stack.
EXIT:   VDESC (D*8) Contains the descriptor for the variable
                    specified by the other arguments.
```

ZDESCB

```
ZDESCB, Break Descriptor Into Four Integers
ARGS:   VDESC,DBNUM,RECTYP,VARNUM,LEVEL
DESC:   ZDESCB breaks a descriptor into 4 integers.
ENTRY: VDESC (I * 4) Contains the descriptor.
EXIT:   DBNUM (I * 4) The database number in HOST system.
        RECTYP(I * 4) The record-type number.
        VRNUM (I * 4) The variable number in record.
        LEVEL (I * 4) The stack level.
```

ZDESCD

```
ZDESCD, Create Variable Descriptor
ARGS:   VDESC,DBNAME,RECTYP,VRNAME,LEVEL
DESC:   ZDESCD looks up a common or record variable name and
        returns the variable's descriptor.  Refer to the
        Machine Specifics documentation for additional
        information on descriptors.  ZDESCD can be called
        once prior to entering a loop which references a
        variable.  This avoids the necessity of creating the
        variable descriptor for each loop iteration.
ENTRY:  DBNAME(N*8) The database in which the variable
                    resides.
        RECTYP(I * 4) The record type number to which the vari-
                    able belongs.  Common variables are indi-
                    cated by setting this argument to zero.
        VRNAME(N*8) The variable name.
        LEVEL (I * 4) This indicates where the CIR/data record
                    can be found.  A zero value indicates
                    that when the descriptor is used, the
                    system should start with the innermost
                    level in the retrieval stack and search
                    outwards for the first level which
                    matches the database and record type
                    specified within the descriptor.  A
                    negative value indicates that the record
                    is LEVEL levels out from the innermost
                    level A positive value indicates that
                    the record is LEVEL levels deep in the
                    retrieval stack.
EXIT:   VDESC (D 8) Contains the descriptor for the variable
                    specified by the other arguments.
```

ZDESCM

```
ZDESCM, Make Descriptor out of Four Integers
ARGS:   VDESC,DBNUM,RECTYP,VARNUM,LEVEL
DESC:   ZDESCM makes a descriptor from 4 integers.
ENTRY:  DBNUM (I * 4) The database number in HOST system.
        RECTYP(I * 4) The record type number. It cannot be 0
                      for a caseless database.
        VRNUM (I * 4) The variable number in record.
        LEVEL (I * 4) The stack level.
EXIT:   VDESC (I * 4) Contains the descriptor for the variable
                      specified by the other arguments.
```

ZDETAL

```
ZDETAL, Find Location of File Control Block of DETAIL File
ARGS:   DUMMY
DESC:   ZDETAL returns the location in table ZERO of the
        file control block for the detail file for the
        current database.
ENTRY:  DUMMY(I * 4) Dummy argument.
EXIT:   None.
RETURN: Location of DETAIL file FCB.
```

ZDTTKY

```
ZDTTKY, Enter a Date String into a Key
ARGS:   DATEST,LENGTH,DATEMP
DESC:   ZDTRKY is called after one of the key initialisation
        routines in order to insert the value of a date
        string into the next location of the key currently
        being defined.
ENTRY:  DATEST (B * n) Date string to insert into key.
        LENGTH (I * 4) Number of characters in strings DATEST
                       and DATEMP.
        DATEMP (B * n) String containing format for decoding
                       the date specified in DATEST.  Legal
                       characters are I(ignore), Y(year),
                       M(month), or D(day).  For
                       example 'MMIDDIYY'
EXIT:   None.
```

ZDTTRC

```
ZDTTRC, Move Date into CIR/Record
ARGS:   DATEST,LENGTH,DATEMP,VDESC
DESC:   ZDTTRC transfers the value of a date string into a
        CIR or data record.
ENTRY:  DATEST (B * n) Date string to transfer to record.
        LENGTH (I * 4) Number of characters in strings DATEST
                       and DATEMP.
        DATEMP (B * n) String containing format for decoding
                       the date specified in DATEST.  Legal
                       characters are I(ignore), Y(year),
                       M(month), or D(day).
        VDESC (D * 8)  Variable descriptor of variable to receive
                       value.
EXIT:   None.
```

ZDTXIN

```
ZDTXIN, Convert Date String into Date Integer
ARGS:   DATEST,ORDINAL,DATEMP,LENGTH
DESC:   ZDTXIN converts a date encoded as a character string
        into a julian integer value.
ENTRY:  DATEST (B * n)  Date string to convert.
        ORDINAL (I * 4) First character in string DATEST to
                        use.
        DATEMP(B * n)   String containing format for
                        decoding the date specified in
                        DATEST.  Legal characters are
                        I(ignore), Y(year), M(month), or
                        D(day).  For example 'MMIDDIYY'
        LENGTH(I * 4)     Number of characters in strings
                        specified above.
EXIT:   None.
```

ZDTXKY

```
ZDTXKY, Enter Date String into Key
ARGS:   DATEST,ORDINAL,LENGTH,DATEMP
DESC:   ZDTXKY is called after one of the key initialisation
        routines in order to insert the value of a date
        string into the next location of the key currently
        being defined.
ENTRY:  DATEST  (B * n) Date string to insert into key.
        ORDINAL (I * 4) Starting byte number in area DATA to
                        transfer the value from.  For a
                        simple variable this value is 1.
        LENGTH (I * 4)  Number of characters in strings
                        specified above.
        DATEMP (B * n)  String containing format for
                        decoding the date specified in
                        DATEST.  Legal characters are
                        I(ignore), Y(year), M(month), or D
                        (day).  For example 'MMIDDIYY'
EXIT:   None.
```

ZDTXRC

```
ZDTXRC, Move Date into CIR/Record
ARGS:   DATEST,ORDINAL,LENGTH,DATEMP,VDESC
DESC:   ZDTXRC transfers the value of a date string into a
        CIR or data record.
ENTRY:  DATEST (B * n)  Date string to transfer to record.
        ORDINAL (I * 4) Starting byte number in area DATA to
                        transfer the value from. For a
                        simple variable this value is 1.
        LENGTH (I * 4)  Number of characters in strings
                        specified above.
        DATEMP (B * n)  String  containing format for
                        decoding the date specified in
                        DATEST. Legal characters are
                        I(ignore), Y(year), M(month), or
                        D(day).
        VDESC (D * 8)   Variable descriptor of variable to
                        receive value.
EXIT:   None.
```

ZEND

```
ZEND,   Terminate Processing of HOST
ARGS:   TSUSED
DESC:   ZEND terminates a HOST run.  It must be called after
        using ZENDDB to close all the databases still open
        for the run. ZEND ensures that all information
        related to the databases are properly handled.  All
        internal tables are cleared and any scratch files
        used are closed and returned.
ENTRY:  None.
EXIT:   TSUSED (I * 4)  The amount of table space actually used
      in the current job is returned by HOST via
      this argument. If the value is negative then
      during the run some tables had to be
      swapped to a scratch disk file in order to
      continue processing. In this case addition-
      al memory should be allocated to reduce
      the swapping time the next time the
      program is used.
```

ZENDDB

```
ZENDDB, Terminate the Use of Database
ARGS:   DBNAME
DESC:   ZENDDB terminates the use of a specified database.
        The files are updated and closed.  The space
        associated with each database is not freed until
        ZEND is called.  ZENDDB must be called for each open
        database prior to calling ZEND.  Prior to calling
        ZENDDB, all streams must have terminated their use
        of the database, otherwise the call fails and
        returns a fatal error.  Failure to close a database
        can cause destruction of the database.
ENTRY:  DBNAME(N * 8) Name of the database. This is the same
                      name that appeared on the database
                      initialisation routine.
EXIT:   None.
```

ZERMSG

```
ZERMSG, Error Description Routine
ARGS:   ERRNUM,BUFFER,BUFLEN,LOGFLG,RNAME
DESC:   ZERMSG converts an error code ERRNUM into coded text
        that can be printed.
ENTRY:  ERRNUM (I * 4) Error code to be converted.
        BUFLEN (I * 4) Maximum number of characters to
                       transfer to error text buffer BUFFER.
        LOGFLG (I * 4) Insert error message in log flag. If
                       the value of this argument is 0 then
                       no message is placed in the system
                       log file.  If the value of this
                       argument is I then the message is
                       placed in the system log file and
                       also transferred to the BUFFER array.
        RNAME  (N * 8) Calling routine name to be included
                       in error message placed in system log
                       file.
EXIT:   BUFFER (B * n) Will contain up to BUFLEN characters
                       describing the error code ERRNUM.
RETURN: Number of characters in the returned message,
        negative if error.
```

ZEXIT

```
ZEXIT,  Exit One Process Level Regardless of Type
ARGS:   OSEED
DESC:   ZEXIT exits the block at the lowest level in the
        current stack, regardless its type(case or record).
ENTRY:  None.
EXIT:   OSEED(I * 4) If this was a process sample level,
                     then the current seed value is returned
                     here so that it can be used on the next
                     call for the next random-number
                     generation.
```

ZFPTKY

```
ZFPTKY, Enter Real Value into Key
ARGS:   DATA
DESC:   ZFPTKY is called after one of the key initialisation
        routines in order to insert a real value into the
        next location of the key currently being defined.
ENTRY:  DATA (R * 4) Real value to insert into key.
EXIT:   None.
```

ZFPTRC

```
ZFPTRC, Move Real into CIR/Record
ARGS:   DATA,VDESC
DESC:   ZFPTRC transfers a real value into a CIR or data
        record.
ENTRY:  DATA (R * 4)  Real value to transfer to record.
        VDESC (D * 8) Variable descriptor of variable to
                      receive value.
EXIT:   None.
```

ZFPXKY

```
ZFPXKY, Enter Real Value into Key
ARGS:   DATA,ORDINAL,LENGTH
DESC:   ZFPXKY is called after one of the key initialisation
        routines in order to insert a real value into the
        next location of the key currently being defined.
ENTRY:  DATA (R * n)    Real value to insert into key.
        ORDINAL (I * 4) Starting byte number in area DATA to
                        transfer the value from. For a
                        simple variable this value is 1.
        LENGTH (I * 4)  Number of bytes in value.
EXIT:   None.
```

ZFPXRC

```
ZFPXRC, Move Real into CIR/Record
ARGS:   DATA,ORDINAL,LENGTH,VDESC
DESC:   ZFPXRC transfers a real value into a CIR or data
        record.
ENTRY:  DATA (R * n)    Real value to transfer to record.
        ORDINAL (I * 4) Starting byte number in area DATA to
                        transfer the value from. For a
                        simple variable this value is 1.
        LENGTH(I * 4)   Number of bytes in value.
        VDESC (D * 8)   Variable descriptor of variable to
                        receive value.
EXIT:   None.
```

ZFROM

```
ZFROM,  Start Creation of FROM Key
ARGS:   DUMMY
DESC:   ZFROM starts the creation of a key.  It creates a
        "FROM" key.  That is, a key which is used to select
        all CIR/record's whose key matches or comes after
        the key that is currently being defined.  It is
        normally called after one of the case/record level
        initialisation routines in order to initialise the
        key selection options for case or record loops.
        Following a call to ZFROM, other routines are called
        to enter values into the key one at a time.  For a
        case key only one call is made to enter the value of
        the case id.  For record keys, the case id is
        assumed to be the same as the last CIR retrieved or
        restored.  Therefore, only the record sort ids have
        to be inserted into the key.  The sort-ids must be
        entered in the order of their appearance in the key
        being created.
ENTRY:  DUMMY(I * 4) Dummy argument to make routine a
                     syntactically correct FORTRAN function.
                     Should always be 0.
EXIT:   None.
```

ZINTKY

```
ZINTKY, Enter Integer Value Into Key
ARGS:   DATA
DESC:   ZINTKY is called after one of the key initialisation
        routines in order to insert an integer value into
        the next location of the key currently being
        defined.
ENTRY:  DATA(I * 4)  Integer value to insert into key.
EXIT:   None.
```

ZINTRC

```
ZINTRC, Move Integer into CIR/record
ARGS:   DATA,VDESC
DESC:   ZINTRC transfers an integer value into a CIR or data
        record.
ENTRY:  DATA (I * 4)  Integer value to transfer to record.
        VDESC (D * 8) Descriptor for variable to modify.
EXIT:   None.
```

ZINXDT

```
ZINXDT, Convert Integer into Date String
ARGS:   IDAYS,DATEST,ORDINAL,LENGTH,DATEMP
DESC:   ZINXDT converts an integer value into a date string
        according to a specified format.
ENTRY:  IDAYS (I * 4)   The integer value to convert.
        ORDINAL (I * 4) Starting byte number in area DATEST
                        to transfer the date string to.
        LENGTH (I * 4)  Number of characters in strings
                        specified above.
        DATEMP (B * n)  String containing format for
                        decoding the date specified in
                        DATEST.  Legal characters are
                        I(ignore), Y(year), M(month), or
                        D(day).  For example 'MMIDDIYY'
EXIT:   DATEST (B * n)  The area in which to place the date
                        string created.  Starting at
                        position ORDINAL as specified above.
```

ZINXKY

```
ZINXKY, Enter Integer Value Into Key
ARGS:   DATA,ORDINAL,LENGTH
DESC:   ZINXKY is called after one of the key initialisation
        routines in order to insert an integer value into
        the next location of the key currently being defined.
ENTRY:  DATA (I * n)   Integer value to insert into key.
        ORDINAL(I * 4) Starting byte number in area DATA to
                       transfer the value from. For a simple
                       variable this value is 1.
        LENGTH(I * 4)  Number of bytes in value.
EXIT:   None.
```

ZINXRC

```
ZINXRC, Move Integer into CIR/Record
ARGS:   DATA,ORDINAL,LENGTH,VDESC
DESC:   ZINXRC transfers an integer value into a CIR or data
        record.
ENTRY:  DATA    (I * n) Integer value to transfer to record.
        ORDINAL (I * 4) Starting byte number in area DATA to
                        transfer the value from. For a
                        simple variable this value is 1.
        LENGTH  (I * 4) Number of bytes in value.
        VDESC   (D * 8) Descriptor for variable to modify.
EXIT:   None.
```

ZINXTM

```
ZINXTM, Convert Integer into Time String
ARGS:   ITIME,TIMSTR,ORDINAL,LENGTH,TIMMAP
DESC:   ZINXTM converts an integer value into a time string
        according to a specified time format.
ENTRY:  ITIME   (I * 4)  The integer to convert.
        ORDINAL (I * 4)  Starting byte number in area TIMSTR to
                         transfer the time string to.
        LENGTH  (I * 4)  Number of characters in strings above.
        TIMMAP  (B * n)  String containing the decoding
                         format for the time string
                         contained in TIMEST. Legal values
                         are I(ignore), H(hour), M(minute),
                         S(second). For example 'HHMMSS'
EXIT: TIMSTR (B * n)     The area to receive the time
                         string. The location the string is
                         placed in this area is dependent on
                         ORDINAL.
```

ZLABEL

```
ZLABEL, Get Value Label for Current Value of a Variable
ARGS:   VDESC,STRING,LENGTH
DESC:   ZLABEL gets the value label for the current value of
        a variable.  It is equivalent to the VALLAB
        function.
ENTRY:  VDESC (D * 8)  The variable descriptor.
        LENGTH (I * 4) Number of characters of value label
                       to retrieve.
EXIT:   STRING (B * n) String area in which the value label
                       will be placed.
RETURN: Number of characters actually transferred, negative
        if error.
```

ZLABLN

```
ZLABLN, Get Value Label for Numeric Variable
ARGS:   VDESC,VALUE,STRING,LENGTH
DESC:   ZLABLN searches the database for the value label
        associated with the specified variable and the
        numeric value.
ENTRY:  VDESC (D * 8) The variable descriptor.
        VALUE (R * 8) The specific value whose label is to
                      be returned.
        LENGTH(I * 4) Number of characters of value label to
                      retrieve.
EXIT:   STRING (B * n) Area in which the value label will be
                       placed.
RETURN: Number of characters actually transferred, negative if error.
```

ZLABLS

```
ZLABLS, Get Value Label for String Variable
ARGS:   VDESC,VALUE,LENGTHI,STRING,LENGTH2
DESC:   ZLABILS searches the database for the value label
        associated with the specified variable and the
        string value.
ENTRY:  VDESC (D * 8)  The variable descriptor.
        VALUE (B * n)  The specific value whose label is to
                       be returned.
        LENGTH1(I * 4) Number of characters in the value.
        LENGTH2(I * 4) Number of characters of value label
                       to retrieve.
EXIT:   STRING (B * n) Area in which the value label is to
                       be placed.
RETURN: Number of characters actually transferred, negative if error.
```

ZLCKRT

```
ZLCKRT, Set or Return the Record Type Lock
ARGS:   DBNAME,RECTYP,LFLAG
DESC:   ZLCKRT sets or returns the lock for a specified
        record type.  It works only for concurrent HOST.
        The routine can be invoked from a regular HOST
        program but it does not have any effect, nor does it
        return any significant value.
ENTRY:  DBNAME(D * 8) The database name to which the record
                      type belongs.
        RECTYP(B * n) If ZLCKRT is invoked to set a record
                      type lock, RECTYP is the record type
                      number (0 means CIR). If it is
                      invoked to return a record type lock,
                      RECTYP should be set to a negative
                      value that is the -1 minus the record
                      type ( i.e. -1 for CIR, - 2 for
                      rectype 1, etc.)
        LFLAG (I * 4) Value to set the lock record type to (
                      if RECTYP is not negative).
EXIT:   LFLAG (B * n) Current value of the lock flag for the
                      rectype specified by a negative value
                      of RECTYP.
```

ZLOGIN

```
ZLOGIN, Initialisation of Master Link
ARGS:   MDSN,MLEN,SDSN,SLEN
DESC:   ZLOGIN specifies the MASTER and SLAVE names (DSNS)
        and logs the slave (current process) into the
        MASTER.  It must be the next routine called after
        ZSTART.  If it is not called, the default master and
        slave names are used.  For non-concurrent HOST,
        ZLOGIN is not operational.
ENTRY:  MDSN (B * n)  The MASTER DSN.
        MLEN (I * 4)  The length of the MDSN.
        SDSN (B * n)  The SLAVE DSN.
        SLEN (I * 4)  The length of the SDSN.
EXIT:   None.
```

ZMSLAB

```
ZMSLAB, Return Value Label for Missing Value
ARGS:   VDESC,VALUE,STRING,LENGTH
DESC:   ZMSLAB retrieves the value label for a specified
        missing value of a certain variable.
ENTRY:  VDESC (D * 8)  The descriptor for the variable.
        VALUE (I * 4)  The missing value whose label is
                       being returned. (0, 1, 2, 3)
        LENGTH(I * 4)  Maximum number of characters to
                       return.
EXIT:   STRING (B * n) String area that will have the label
                       placed in it.
RETURN: The number of characters actually transferred
        negative if error.
```

ZMSTRC

```
ZMSTRC, Transfer Missing Value to Variable
ARGS:   NUMBER,VDESC
DESC:   ZMSTRC sets the value of a specified variable to
        either undefined or one of the 3 missing values.
ENTRY:  NUMBER(I * 4) The value 0 if the variable is to be
                      set to undefined or the value 1-3 for
                      missing value 1-3.
        VDESC (D * 8) The descriptor of the destination
                      variable.
EXIT:   None.
```

ZNCASE

```
ZNCASE, Get Number of Cases in Database
ARGS:   DBNAME
DESC:   ZNCASE returns the number of cases currently in the
        database.
ENTRY:  DBNAME (N * 8) The database name.
EXIT:   None.
RETURN: The number of cases in the database, negative if error.
```

ZNEW

```
ZNEW,   Check if New CIR record was Created
ARGS:   DUMMY
DESC:   ZNEW returns an indication of whether the
        CIR/record in the lowest level of the retrieval
        stack was just created or if it existed prior to
        this reference.
ENTRY:  DUMMY (I * 4 ) Dummy argument needed to make a
                       syntactically correct FORTRAN
                       function. Should always be 0.
EXIT:   None.
RETURN: 0 if CIR/record previously existed, 1 if CIR/record
        was just created, negative if error.
```

ZNOR

```
ZNOR,   Get The Number of Records in Current Case
ARGS:   RECTYP
DESC:   ZNOR returns the number of records of the specified
        type within the current case.  This function
        corresponds to the COUNT function in DBMS.  ZNOR
        differs from ZNORD only in that the CIR
        corresponding to the innermost block in retrieval
        stack is used, regardless of database being used.
        ZNOR cannot be used on a caseless database.
ENTRY:  RECTYP(I * 4) The record type. If it is zero, then
                      it returns the total number of records
                      in the case.
EXIT:   None.
RETURN: The number of records of the specified type in the
        current case, negative if error.
```

ZNORD

```
ZNORD,  Get Number of Records in Current Case for Specified Database
ARGS:   DBNAME,RECTYP
DESC:   ZNORD returns the number of records of the specified
        type in the current case for the specified database.
        This function corresponds to the COUNT function in
        DBMS.
ENTRY:  DBNAME (N * 8) The database name.  ZNORD searches
                       backwards from the current level for
                       a block "belonging" to the specified
                       database and uses its CIR to return
                       the value from.  ZNORD cannot be used
                       on a caseless database.
        RECTYP (I * 4) The record type.  If it is zero, then
                       it returns the total number of
                       records in the case.
EXIT:   None.
RETURN: The number of records of the specified type in the
        current case, negative if error.
```

ZNRECS

```
ZNRECS, Get Number of Records of Type
ARGS:   DBNAME,RECTYP
DESC:   ZNRECS returns the number of records of a specified
        type currently in the database.
ENTRY:  DBNAME (N * 8) The database name.
        RECTYP (I * 4) The record type number, 0 for CIR.
EXIT:   None.
RETURN: The number or records of the specified type,
        negative if error.
```

ZNSIDS

```
ZNSIDS, Get Number of Sort-ids in Record Type
ARGS:   DBNAME,RECTYP
DESC:   ZNSIDS returns the number of sort-ids in the key for
        the specified record type.
ENTRY:  DBNAME (N * 8) The database name.
        RECTYP (I * 4) The record type number.
EXIT:   None.
RETURN: The number of sort-ids in the specified record type,
        negative if errors.
```

ZNVARS

```
ZNVARS, Get the Number of Variables in Record
ARGS:   DBNAME,RECTYP
DESC:   ZNVARS returns the number of variables in a
        specified record type or the CIR.  It can be used to
        determine if a database is caseless or not.
ENTRY:  DBNAME (N * 8) The database name.
        RECTYP (I * 4) The record type number.  Use zero for
                       the CIR.  If RECTYP is 0 and the
                       database is caseless, the function
                       returns 0.
EXIT:   None.
RETURN: The number of variables in the specified record
        type, zero for the number of variables in the CIR of
        a caseless database, negative if error.
```

ZOPEN

```
ZOPEN,  Determine if Database is Available
ARGS:   DBNAME
DESC:   ZOPEN is called to determine if a specified database
        is currently open for use or not.
ENTRY:  DBNAME ( N * 8 ) The database name to check.
EXIT:   None.
RETURN: Internal database number ( small positive integer ),
        negative if error.
```

ZOPT

```
ZOPT,   Set or Return Various System Options
ARGS:   OPT
DESC:   ZOPT allows the user to select, deselect or return
        various options.
ENTRY:  OPT (I * 4)  The sum of the options to select if
                     positive; -1 to return the current
                     settings.
  OPT             Description
  0*1 or 1*1      allow or disallow the storage of valid
                  values
  0*2 or 1*2      allow or disallow the storage of undefined
                  values
  0*4 or 1*4      allow or disallow the storage of missing
                  values
  0*8 or 1*8      allow or disallow backward search In the
                  stack when a "from" descriptor level Is 0
  0*16 or 1*16    allow or disallow backward search in the
                  stack when a "to" descriptor level is 0
EXIT:   None.
RETURN: 0 if OPT is between 0 and 31, the sum of selected
        options if OPT is -1, negative if error.
```

ZORDB

```
ZORDB,  Initialise Specified Random Database
ARGS:   DBNAME,DBPASS,HSPASS,RDPASS,WRPASS, UPD,PREFIX,PRELEN
DESC:   ZORDB is called to attach the database.  It also
        verifies the passwords to ensure database security
        and accessibility.  It must be called prior to any
        reference to the new database.  Each open database
        requires a significant amount of memory to be
        allocated in the table area regardless of whether it
        is currently being used or not.  ZORDB may only be
        called for a database that is in random format.
ENTRY:  DBNAME(N * 8) Name of the database. This is the same
                      name that would appear on an "OLD FILE"
                      command.
        DBPASS(N * 8) The password for the database.
        HSPASS(N * 8) The password required to access the
                      database via HOST. If no password is
                      required then this field should
                      contain a blank name.
        RDPASS(N * 8) The read security password for the
                      database. This password is used to
                      define the default read security level
                      for any stream which does not
                      specifically set the read security
                      level for this database. If the
                      argument contains 8 blanks then the
                      standard DBMS default is used for
                      any stream not specifying a read
                      security password.
        WDPASS(N * 8) The write security password for the
                      database. This password is used to
                      define the default write security
                      level for any stream which does not
                      specifically set the write security
                      level for this database. If the
                      argument contains 8 blanks then the
                      standard DBMS default is used for
                      any stream not specifying a write
                      security password.
        UPD (I * 4)   Update/nonupdate flag. If UPD is 0
                      then the database is attached for read
                      only access. If however the value of
                      this argument is 1 or 2 then the
                      database is attached for exclusive
                      usage in order to allow the job to
                      modify the database. If the value is 2
                      then every change to the database
                      causes the changed internal tables to
                      be rewritten to the database file. If
                      the value is 1 then only when HOST
                      determines it is necessary will the
                      tables be rewritten to the database
```

```
                    file.
        PREFIX(B * n) Prefix used for database filenames.
        PRELEN(I * 4) Number of characters in "PREFIX".
EXIT:   None.
RETURN: The database number used in creating descriptors if
        no errors are encountered ( 1 for 1st opened
        database, 2 for 2nd, etc.  Negative if error.
```

ZOSDB

```
ZOSDB,  Initialise Specified Sequential Database
ARGS:   DBNAME,DBPASS,HSPASS,RDPASS,WRPASS,UPD,
        PREFIX,PRELEN,SIFNAM,SOFNAM
DESC:   ZOSDB is called initially to attach a sequential
        format database.  It also verifies the passwords to
        ensure database security and accessibility.  Each
        open database requires a significant amount of
        memory to be allocated in the table area regardless
        of whether it is currently being used or not.  ZOSDB
        may only be called for a database which is in
        sequential format.
ENTRY:  DBNAME(N * 8) Name of the database. This is the same
                      name that would appear on an "OLD FILE"
                      command.
        DBPASS(N * 8) The password for the database.
        HSPASS(N * 8) The password required to access the
                      database via HOST. If no password is
                      required then this field should contain
                      a blank name.
        RDPASS(N * 8) The read security password for the
                      database. This password is used to
                      define the default read security level
                      for any stream which does not
                      specifically set the read security
                      level for this database. If the
                      argument contains 8 blanks then the
                      standard DBMS default is used for any
                      stream not specifying a read security
                      password.
       WRPASS (N * 8) The write security password for the
                      database. This password is used to
                      define the default write security level
                      for any stream which does not
                      specifically set the write security
                      level for this database. If the
                      argument contains 8 blanks then the
                      standard DBMS default is used for any
                      stream not specifying a write security
                      password.
       UPD (I * 4)    Update/nonupdate flag. If UPD is 0 then
                      the database is attached for read only
                      access. If however the value of this
                      argument is 1 or 2 then the database is
                      attached for exclusive usage in order
                      to allow the job to modify the
                      database. If the value is 2 then every
                      change to the database causes the
                      changed internal tables to be rewritten
                      to the database file. If the value is I
                      then only when HOST determines it is
                      necessary will the tables be rewritten
```

```
                to the database file.
        PREFIX(B * n)  Prefix used for database filenames.
        PRELEN(I * 4)  Number of characters in "PREFIX".
        SIFNAM (N * 8) The "FILENAME" of the file containing
                       the sequential format input database.
        SOFNAM (N * 8) The  "FILENAME" of the file which will
                       contain the sequential format output
                       database. If this is not an update run
                       then this argument should contain all
                       blanks.
EXIT:   None.
RETURN: The database number used in creating descriptors if
        no errors are encountered ( 1 for 1st opened
        database, 2 for 2nd, etc ). negative if error.
```

ZRCNT

```
ZRCNT,  Initialise COUNT Record Processing Block
ARGS:   RECTYP,TRECS,INDEX,START
DESC:   ZRCNT is the first of a series of routines that can
        be called in order to create "PROCESS RECORD" block
        with the COUNT option.  ZRCNT can be used to process
        ALL records by setting INDEX and START to 1 and
        TRECS to -1.  The arguments correspond to the
        arguments on the COUNT = clause of the PROCESS CASE
        command in RETRIEVAL.
ENTRY:  RECTYP(I * 4) The record type number to be
                      processed.
        TRECS (I * 4) Total number of records to be
                      processed.
        INDEX (I * 4) Every INDEX the record will be
                      processed.
        START (I * 4) Starting with the START the record in
                      the database.
EXIT:   None.
RETURN: Stack level of the newly created block, negative if
        error.
```

ZRCNTD

```
ZRCNTD, Set Current Database and Initialise COUNT
        Record-processing Block
ARGS:   RECTYP,TRECS,INDEX,START,DBNAME
DESC:   ZRCNTD is called to make DBNAME the current database
        and to start creating a "PROCESS RECORD" block with
        the COUNT option.  The purpose of this call is to
        initialise the retrieval stack with the required
        information.  The arguments correspond to the
        arguments on the COUNT = clause of the PROCESS CASE
        command in RETRIEVAL.
ENTRY:  RECTYP(I * 4) The record type number to be
                      processed.
        TRECS (I * 4) Total number of records to be
                      processed.
        INDEX (I * 4) Every INDEX the record will be
                      processed.
        START (I * 4) Starting with the START the record in
                      the database.
        DBNAME(N * 8) Database name.
EXIT:   None.
RETURN: Stack level of the newly created block, negative if
        error.
```

ZRCNTL

```
ZRCNTL, Initialise COUNT Record Processing Block
        Belonging to CIR
ARGS:   RECTYP,TRECS,INDEX,START,LEVEL
DESC:   ZRCNTL is called to start creating a "PROCESS
        RECORD" block (with the COUNT option) that belongs
        to the CIR at the specified level.  The purpose of
        this call is to initialise the retrieval stack with
        the required information.  Potentially, it can
        switch databases.  ZRCNTL can be used to process ALL
        records by setting INDEX and START to 1 and TRECS to
        -1.  The arguments correspond to the arguments on
        the COUNT clause of the PROCESS CASE command in
        RETRIEVAL.
ENTRY:  RECTYP(I * 4) The record type number to be
                      processed.
        TRECS (I * 4) Total number of records to be
                      processed.
        INDEX( I * 4) Every INDEX the record will be
                      processed.
        START (I * 4) Starting with the START the record in
                      the database.
        LEVEL (I * 4) Stack level of the CIR block to which
                      the record should belong.
EXIT:   None.
RETURN: Stack level of the newly created block, negative if
        error.
```

ZRCTDT

```
ZRCTDT, Transfer Date from CIR/Record
ARGS:   VDESC,DATEST,LENGTH,DATEMAP
DESC:   ZRCTDT retrieves a date value from the specified
        CIR/data record and converts it to a date string
        according to the user specified date map.
ENTRY:  VDESC (D * 8) Variable descriptor of variable to
                      retrieve.
        DATEST(B * n) Location to place date string.
        LENGTH(I * 4) Number of characters in strings DATEST
                      and DATEMP.
        DATEMP(B * n) String  containing the decoding format
                      for the date string contained in
                      DATEST. Legal values are W(day of
                      week) M(month), D(day), Y(year). For
                      example, 'MM/DD/YY' would produce
                      '12/31/86'.
EXIT:   DATEST(B * n) Will contain date string retrieved
                      from current CIR/data record.
```

ZRCTFP

```
ZRCTFP, Transfer Floating-point Value from Record
ARGS:   VDESC,DATA
DESC:   ZRCTFP retrieves a real value from the specified
        CIR/data record.
ENTRY:  VDESC (D * 8) Variable descriptor of variable to
                      retrieve.
EXIT:   DATA (R * 4)  Will contain real value retrieved from
                      current CIR/data record.
```

ZRCTIN

```
ZRCTIN, Transfer Integer Value from Record
ARGS:   VDESC,DATA
DESC:   ZRCTIN retrieves an integer value from the specified
        CIR/data record.
ENTRY:  VDESC (D * 8) Variable descriptor of variable to
                      retrieve.
EXIT:   DATA (I * 4)  Will contain integer value retrieved
                      from current CIR/data record.
```

ZRCTRC

```
ZRCTRC, Transfer Value from One Variable to Another
ARGS:   VDESCI,VDESC2
DESC:   ZRCTRC transfers data from one variable in one
        record to a variable in either another record or the
        same record.
ENTRY:  VDESC1(D * 8)  The descriptor of the source
                       variable.
        VDESC2 (D * 8) The descriptor of the destination
                       variable.
EXIT:   None.
```

ZRCTST

```
ZRCTST, Transfer String from CIR/Record
ARGS:   VDESC,DATA,LENGTH
DESC:   ZRCTST retrieves a character string value from the
        specified CIR/data record.
ENTRY:  VDESC (D * 8) Variable descriptor of variable to
                      retrieve.
        LENGTH(I * 4) Number of characters at location DATA.
EXIT:   DATA  (B * n) Will contain character string
                      retrieved from current CIR/data
                      record.
```

ZRCTTM

```
ZRCTTM, Transfer Time from CIR/Record
ARGS:   VDESC,TIMEST,LENGTH,TIMEMP
DESC:   ZRCTTM retrieves a time string from the specified
        CIR/data record.
ENTRY:  VDESC (D * 8)   Variable descriptor of variable to
                        retrieve.
        TIMEST (B * n)  Location to place time string.
        LENGTH (I * 4)  Number of characters in strings
                        TIMEST and TIMEMP.
        TIMEMP (B * n)  String containing the decoding
                        format for the time string contained
                        in TIMEST. Legal values are H(hour),
                        M(minute), S(second). For example,
                        'HH:MM:SS' would produce '11:59:59'.
EXIT:   TIMEST (B * n)  Will contain time string retrieved
                        from current CIR/data record.
```

ZRCXDT

```
ZRCXDT, Transfer Date from CIR/Record
ARGS:   VDESC,DATEST,ORDINAL,LENGTH,DATEMP
DESC:   ZRCXDT retrieves a date string from the specified
        CIR/data record.
ENTRY:  VDESC (D * 8)  Variable descriptor of variable to
                       retrieve. See routines ZDESC and
                       ZDESCD for creating a descriptor.
        ORDINAL(I * 4) Starting byte number in area DATA to
                       transfer the value to. For a simple
                       variable this value is 1.
        LENGTH(I * 4)  Number of characters in strings
                       specified above.
        DATEMP(B * n)  String containing format for decoding
                       the date specified in DATEST.  Legal
                       characters are Y(year), M(month), or
                       D(day).  All other characters are
                       transferred "as is" to DATEST. For
                       example, 'MM/DD/YY' would produce
                       '12/25/81'.
EXIT:   DATEST(B * n)  Will contain date string retrieved
                       from current CIR/data record.
```

ZRCXFP

```
ZRCXFP, Transfer Real from CIR/Record
ARGS:   VDESC,DATA,ORDINAL,LENGTH
DESC:   ZRCXFP retrieves a real value from the specified
        CIR/data record.
ENTRY:  VIDESC (D * 8)  Variable descriptor of variable to
                        retrieve.
        ORDINAL (I * 4) Starting byte number in area DATA to
                        transfer the value to. For a simple
                        variable this value is 1.
        LENGTH (I * 4)  Number of bytes at location DATA.
EXIT:   DATA (R * n)    Will contain real value retrieved
                        from current CIR/data record.
```

ZRCXIN

```
ZRCXIN, Transfer Integer from CIR/Record
ARGS:   VDESC,DATA,ORDINAL,LENGTH
DESC:   ZRCXIN retrieves an integer value from the specified
        CIR/data record.
ENTRY:  VDESC (D * 8)   Variable descriptor of variable to
                        retrieve.
        ORDINAL (I * 4) Starting byte number in area DATA to
                        transfer the value to. For a simple
                        variable this value is 1.
        LENGTH(I * 4)   Number of bytes at location DATA.
EXIT:   DATA (I * n)  Will contain integer value retrieved
                      from current CIR/data record.
```

ZRCXST

```
ZRCXST, Transfer a String from a CIR/Record
ARGS:   VDESC,DATA,ORDINAL,LENGTH
DESC:   ZRCXST retrieves a character string value from the
        specified CIR/data record.
ENTRY:  VDESC (D * 8)  Variable descriptor of variable to
                       retrieve.
        ORDINAL(I * 4) Starting byte number in area DATA to
                       transfer the value to. For a simple
                       variable this value is 1.
        LENGTH (I * 4) Number of bytes at location DATA.
EXIT:   DATA (B * n)   Will contain character string
                       retrieved from current CIR/data
                       record.
```

ZRCXTM

```
ZRCXTM, Transfer Time from CIR/Record
ARGS:   VDESC,TIMEST,ORDINAL,LENGTH,TIMEMP
DESC:   ZRCXTM retrieves a time string from the specified
        CIR/data record.
ENTRY:  VDESC (D * 8)  Variable descriptor of variable to
                       retrieve.
        ORDINAL(I * 4) Starting byte number in area DATA to
                       transfer the value to. For a simple
                       variable this value is 1.
        LENGTH (I * 4) Number of characters in strings
                       above.
        TIMEMP (B * n) String containing the decoding format
                       for the time string contained in
                       TIMEST. Legal values are H(hour),
                       M(minute), S(second). For example,
                       'HH:MM:SS' would produce '11:59:59'.
EXIT:   TIMEST(B * n)  Will contain time string retrieved
                       from current CIR/data record.
```

ZRDEL

```
ZRDEL,  Delete Current Record from Database
ARGS:   DUMMY
DESC:   ZRDEL is called after a record is retrieved to
        delete the current record.
ENTRY:  DUMMY (I * 4) Dummy argument needed to make a
                      syntactically correct FORTRAN
                      function. Should always be 0.
EXIT:   None.
```

ZREXIT

```
ZREXIT, Terminate Record Processing Level
ARGS:   OLDSEED
DESC:   ZREXIT is called to terminate a record level and pop
        back one level in the retrieval stack.  It only
        terminates the innermost record level.
ENTRY:  None.
EXIT:   OLDSEED(I * 4) It is set to the current value of the
                       seed, if the current level is a
                       PROCESS RECORD level with the SAMPLE
                       option.  Otherwise it value is
                       undefined.
```

ZRFIND

```
ZRFIND, Find Record with Given Key
ARGS:   OPT, LFLAG, DIRECT, BEGINR, BEGINK
DESC:   ZRFIND finds an existent record or creates a new
        record with a previously specified key.  It is
        called after ZRGDMY, ZRGDML or ZRRDMY has created a
        dummy block and after putting values into the record
        id variable in the record and ( if OPT is 1 ) in
        other record variables.  When ZRFIND is executed,
        the values of the record ids are used to create the
        record key.  If the record ids variable are
        undefined, then the dummy block is converted into a
        PROCESS RFC ALL block and the next or previous
        record is read.  Otherwise, the dummy block ZRGDMY
        is converted into a REC IS block and: a) if OPT is
        1, then the undefined values are updated to the
        values read from the found record or b) if OPT is 2
        the full current record is replaced by the read
        record.
ENTRY:  OPT (I * 4)    Controls the update/replace of the
                       CIR.
        LFLAG (I * 4)  Lock flag.
        DIRECT (I * 4) If PROCESS REC then use 1 to get the
                       next record and -1 to get the
                       previous record.
        BEGINR(I * 4)  Number of record ids to include in
                       range.
        BEGINK (I * 4) Number of record ids to include in
                       starting key.
EXIT:   None.
RETURN: -4001 if record not found;
        -3026 if record is found but is incompatible locked;
        + 0003 if record is found and has a compatible lock;
        + 0004 if record is found and available for REC IS;
        + 0000 if record is found and available for PROCESS
        REC; negative if error.
```

ZRFRST

```
ZRFRST, Start-up a Record Level Block and Get First
        Record
ARGS:   LFIAG
DESC:   ZRFRST is called after record-block initialisation
        and all key creations, and after all key-definition
        routines.  It starts-up the record block and gets
        the first record that meets all of the selection
        options previously specified.  No other record-level
        function can be called until this block is
        successfully executed.  After ZRFRST is executed, no
        further key definitions may be made.  The system
        checks that the current lock flag is compatible with
        LFLAG and if it is, LFILAG becomes the new lock.
ENTRY:  LFLAG (I * 4)  The lock flag.
EXIT:   None.
```

ZRGDMD

```
ZRGDMD, Set Current Database and initialise Dummy Block
ARGS:   RECTYP, DBNAME
DESC:   ZRGDMD is called to make DBNAME the current database
        and to start creating a dummy block.  The purpose of
        this call is to initialise the retrieval stack with
        the required information.
ENTRY:  RECTYP (I * 4) The record type to be processed.
        DBNAME (N * 8) The name of the database from which
                       the record should be selected.
EXIT:   None.
RETURN: Stack level of the newly created block, negative if
        error.
```

ZRGDML

```
ZRGDML, Start Dummy Record Block Belonging to CIR
ARGS:   RECTYP,LEVEL
DESC:   ZRGDML is called to start creating a dummy block
        that belongs to the CIR at the specified level.  The
        purpose of this call is to initialise the retrieval
        stack with the required information.  Potentially,
        it can switch databases.
ENTRY:  RECTYP(I * 4) Record type.
        LEVEL (I * 4) Level in stack of the CIR block to
                      which the record belongs.
EXIT:   None.
RETURN: Stack level of the newly created block, negative if
        error.
```

ZRGDMY

```
ZRGDMY, Start Dummy Record Block
ARGS:   RECTYP
DESC:   ZRGDMY is called to start creating a dummy block
        that belongs to the CIR at the specified level.  The
        purpose of this call is to initialise the retrieval
        stack with the required information
ENTRY:  RECTYP(I * 4)   Record type for record block.
EXIT:   None.
RETURN: Stack level of the newly created block, negative if
        error.
```

ZRIS

```
ZRIS,   Initialise RECORD IS Block
ARGS:   RECTYP,NEW,OLD
DESC:   ZRIS is the first of a series of routines that can
        be called in order to create "RECORD IS" block.  The
        purpose of this call is to initialise the retrieval
        stack with the required information.
ENTRY:  RECTYP(I * 4) The record type to be processed.
        NEW (I * 4)   NEW is I if a new record can be created
                      by this level, otherwise it is 0. The
                      database must have been opened for
                      UPDATE to allow a value of 1.
        OLD (I * 4)   OLD is I if an old record can be
                      accessed by this level, otherwise it is
                      0.
EXIT:   None.
RETURN: Stack level of the newly created block, negative if
        error.
```

ZRISD

```
ZRISD,  Set Current Database and Initialise RECORD IS
        Block
ARGS:   RECTYP,NEW,OLD,DBNAME
DESC:   ZRISD is called to make DBNAMF, the current database
        and to start creating a "RECORD IS" block.  The
        purpose of this call is to initialise the retrieval
        stack with the required information.
ENTRY:  RECTYP(I * 4) The record type to be processed.
        NEW (I * 4) NEW is 0 if a record cannot be created
                    by this level. The database must have
                    been opened for UPDATE to allow a value
                    of 1.
        OLD (I * 4) OLD is 0 if an old record cannot be
                    accessed by this level, otherwise it is
                    1.
        DBNAME(N * 8) The name of the database from which the
                    record should be selected.
EXIT:   None.
RETURN: Stack level of the newly created block, negative if
        error.
```

ZRISL

```
ZRISL,  Initialise RECORD IS Processing Block Belonging to
        CIR
ARGS:   RECTYP,NEW,OLD,LEVEL
DESC:   ZRISL is called to start creating a "RECORD IS"
        block that belongs to the CIR at the specified
        level.  The purpose of this call is to initialise
        the retrieval stack with the required information.
        Potentially, it can switch databases.
ENTRY:  RECTYP(I * 4) The record type to be processed.
        NEW (I * 4)   NEW is 0 if a new record is not legal
                      (NEW record IS). 1 if editing record is
                      legal (OLD record IS or record IS).
        OLD (I * 4)   OLD is 0 if an old record is not legal
                      (OLD record IS). 1 if a new record is
                      legal (NEW record IS or record IS).
        LEVEL (I * 4) Level in stack of the CIR block to
                      which the record belongs.
EXIT:   None.
RETURN: Stack level of the newly created block, negative if error.
```

ZRLAST

```
ZRLAST, Start-up Record Level Block and Get Last Record
ARGS:   LFLAG
DESC:   ZRLAST is called after record-block initialisation
        and all key creation, and after all key definition
        routines.  It starts-up the record block and gets
        the last record that meets all of the selection
        options previously specified.  No other record-level
        function can be called until this block is
        successfully executed.  After ZRLAST is executed, no
        further key definitions may be made.  The system
        checks that the current lock flag is compatible with
        LFLAG and if it is, LFLAG becomes the new lock.
ENTRY:  LFLAG (I * 4)     The lock flag.
EXIT:   None.
```

ZRLOCK

```
ZRLOCK, Get Lock Type of Innermost Level of Execution
        Stack
ARGS:   LFLAG
DESC:   ZRLOCK returns the lock type at the current
        innermost level of the execution stack.
ENTRY:  None.
EXIT:   LFLAG (I * 4)     Lock type.
RETURN: 0 if the level is not write-locked.
        1 if the level is write-locked.
```

ZRNAMD

```
ZRNAMD, Get the Record Name for Record Type
ARGS:   DBNAME,RECTYP,RECNAM
DESC:   ZRNAMD lookups a record type and returns the record
        name associated with it.
ENTRY:  DBNAME ( N * 8 ) The name of the database to which the
                         record belongs.
        RECTYP (I * 4)   The record type.
EXIT:   RNAME (N * 8)    The record name corresponding to
                         database and record type specified.
```

ZRNEXT

```
ZRNEXT, Start-up Record Level Block and Get Next Record
ARGS:   LFLAG
DESC:   ZRNEXT is called after record-block initialisation
        and all key creation, and after all key definition
        routines.  It starts-up the record block and gets
        the next record that meets all of the selection
        options previously specified.  No other record-level
        function can be called until this block is
        successfully executed.  After ZRNEXT is executed, no
        further key definitions may be made.  The system
        checks that the current lock flag is compatible with
        LFLAG and if it is, LFLAG becomes the new lock.
ENTRY:  LFLAG (I * 4)  The lock flag.
EXIT:   None.
```

ZRNUM

```
ZRNUM,  Get Record Number for Record Name of Current
        Database
ARGS:   RNAME
DESC:   ZRNUM lookups the record name specified in the
        current database and returns the corresponding
        record type number.
ENTRY:  RNAME (N * 8) The record name to lookup.
EXIT:   None.
RETURN: Record type number, negative if error.
```

ZRNUMD

```
ZRNUMD, Get Type Number for Record Name of Specified
        Database
ARGS:   DBNAME,RNAME
DESC:   ZRNUMD lookups a record name and returns the record
        type value associated with it for the specified
        database.
ENTRY:  DBNAME(N * 8) The database in which the record
                      resides.
        RNAME(N * 8) The record name to lookup.
EXIT:   None.
RETURN: The record type number associated with the specified
        record name, negative if error.
```

ZRPREV

```
ZRPREV, Start-up Record Level Block and Get Previous
        Record
ARGS:   LFLAG
DESC:   ZRPREV is called after record-block initialisation
        and all key creation, and after all key definition
        routines.  It starts-up the record block and gets
        the previous record that meets all of the selection
        options previously specified.  No other record-level
        function can be called until this block is
        successfully executed.  After ZRPREV is executed, no
        further key definitions may be made.  The system
        checks that the current lock flag is compatible with
        LFLAG and if it is, LFLAG becomes the new lock.
ENTRY:  LFLAG (I * 4)         The lock flag.
EXIT:   None.
```

ZRRDMY

```
ZRRDMY, Terminate Innermost Level and Reset Block to
        Dummy
ARGS:   DUMMY
DESC:   ZRRDMY terminates all processing at this level,
        writes the record if necessary, and resets the block
        back to dummy status.
ENTRY:  DUMMY(I * 4) Dummy argument needed to make a
                     syntactically correct FORTRAN function.
                     Should always be 0.
EXIT:   None.
```

ZRREST

```
ZRREST, Restore Record from Database
ARGS:   LFLAG
DESC:   ZRREST is called to replace a record in the
        retrieval stack with the corresponding record from
        the database.
ENTRY:  LFLAG(I * 4)      The lock flag.
EXIT:   None.
```

ZRSAM

```
ZRSAM,  Initialise Record Processing Loop
ARGS:   RECTYP,SAMPLE,SEED
DESC:   ZRSAM is the first of a series of routines that can
        be called in order to create "PROCESS RECORD" block
        with the SAMPLE option.
ENTRY:  RECTYP (I * 4) The record type value to be processed.
        SAMPLE (R * 4) Sample size (SAMPLE).
        SEED (I * 4)   Starting seed for random-number
                       generator.  Same seed always produces
                       the same random selection sequence.
                       Any odd value can be used for the
                       seed.  See routine ZREXIT for
                       obtaining the value of the seed after
                       the loop is finished.
EXIT:   None.
RETURN: Stack level of the newly created block, negative if
        error.
```

ZRSAMD

```
ZRSAMD, Set Current Database and Initialise SAMPLE
        Record-processing block
ARGS:   RECTYP,SAMPLE,SEED,DBNAME
DESC:   ZRSAMD is called to make DBNAME the current database
        and to start creating a "PROCESS CASE" block with
        the SAMPLE option.  The purpose of this call is to
        initialise the retrieval stack with the required
        information.
ENTRY:  RECTYP(I * 4)  Record type.
        SAMPLE (R * 4) Sample size ( 0 SAMPLE 1 )
        SEED (I * 4)   Starting seed.
        DBNAME(N * 8)  The name of the database form which
                       the record is selected.
EXIT:   None.
RETURN: Stack level of the newly created block, negative if
        error.
```

ZRSAML

```
ZRSAML, Start SAMPLE PROCESS RECORD Block
        Belonging to CIR
ARGS:   RECTYP,SAMPLE,SEED,LEVEL
DESC:   ZRSAML is called to start creating a "PROCESS
        RECORD" block (with the SAMPLE option) that belongs
        to the CIR at the specified level.  The purpose of
        this call is to initialise the retrieval stack with
        the required information.  Potentially, it can
        switch databases.
ENTRY:  RECTYP(I * 4) The record type value to be processed.
        SAMPLE(R * 4) Sample size ( 0 SAMPLE 1 ).
        SEED (I * 4)  Starting seed. The same seed always
                      generates the same sequence of random
                      numbers.
        LEVEL (I * 4) Level in stack of the CIR block to
                      which the record belongs.
EXIT:   None.
RETURN: Stack level of the newly created block, negative if error.
```

ZRWRIT

```
ZRWRIT, Replace Modified Record on Database and Lock
        of Block
ARGS:   LFLAG
DESC:   ZRNWIT is called to replace the database version of
        the record with the current version in the retrieval
        stack.  This process is performed automatically by
        the HOST routines when the next record is retrieved
        or the record level is terminated.  However the user
        may want to write the modified record to the
        database and change the lock for further processing.
ENTRY:  LFLAG (I * 4)           Lock flag.
EXIT:   None.
```

ZSDESC

```
ZSDESC, Get Descriptor for Sort-id Variable
ARGS:   DBNAME,RECTYP,SIDNUM,VDESC
DESC:   ZSDESC returns the variable descriptor corresponding
        to a specified sort-id of a given record type.
ENTRY:  DBNAME(N*8) The database name.
        RECTYP(I * 4) The record type number.
        SIDNUM(I * 4) The sort-id number.
EXIT:   VDESC (D * 8) The descriptor for the specified
                      sort-id.
```

ZSECLV

```
ZSECLV, Return the Current Security Values for a Database
ARGS:   DBNAME,RDLEV WRLEV
DESC:   ZSECLV returns the read and write access levels.
ENTRY:  DBNAME N * 8) The database name.
EXIT:   RDLEV (I * 4) The read level for the database.
        WRLEV (I * 4) The write level for the database.
```

ZSECUR

```
ZSECUR, Specify Security Passwords
ARGS:   DBNAME,RDPASS,WRPASS
DESC:   ZSECUR specifies the passwords for the read and
        write security levels for a specific database for
        the current stream.  It should be called for each
        database used by each stream, which needs security
        levels different from those specified when the
        database was initially attached.  Otherwise the
        default levels specified on the attach call are
        assigned.  If the passwords specified by ZSECUR
        select security levels less than the default values,
        then the default values are used.
ENTRY:  DBNAME(N * 8) The name of the database the following
                      security passwords apply to.
        RDPASS(N * 8) Read security password for the current
                      stream.
        WRPASS(N * 8) Write security password for the current
                      stream.
EXIT:   None.
```

ZSTART

```
ZSTART, Initialise HOST System
ARGS:   MAXUSR,TABTYP,TAB1,TAB2
DESC:   ZSTART initialises all tables needed by the other
        HOST routines.  It must be called prior to any other
        HOST routine.
ENTRY:  MAXUSR (I * 4) Maximum number of streams accessing
                       the HOST system within this job.  A
                       small amount of space is allocated
                       for each stream specified regardless
                       of the number of streams currently
                       active.
        TABTYP (I * 4) Type of table space to be allocated
                       for use by the HOST system.  See
                       Machine Specifics documentation
                       for more information on use of this
                       and the next two associated
                       arguments.
        = 1            The HOST system gets space from the
                       operating system and returns it when
                       ZEND is called.
        TAB1  (I * 4)  Amount of space (in "double-word"s)
                       to be requested.  At the end of the
                       job the amount actually used is
                       returned so that a better estimate
                       can be applied next time.
        TAB2 (R * 8)   0. Unused argument for this type but
                       must appear on call in order to obey
                       FORTRAN syntax rules.
        = 2            The calling routine provides an area
                       to be used by HOST.  This area must
                       be "doubleword" aligned and must be
                       available for use at all times (i.e.
                       if program is segmented then this
                       area must be in the root segment).
       TAB1 (I * 4)    Amount of contiguous space available
                       for use by the HOST system (in units
                       of "dwrds").
       TAB2 (R * 8)    Array to be used by the HOST system.
                       In this case the array TAB2 must
                       not be modified in any way until
                       after ZEND is called. If any item of
                       this array is modified by the user
                       then the database will probably have
                       erroneous information written to it
                       and HOST actions will become unpre-
                       dictable.
EXIT:   None.
```

ZSTTKY

```
ZSTTKY, Enter String Value into Key
ARGS:   DATA,LENGTH
DESC:   ZSTTKY is called after one of the key initialisation
        routines in order to insert a character value into
        the next location of the key currently being defined.
ENTRY:  DATA (B * n)  Character string to insert into key.
        LENGTH(I * 4) Number of characters in string.
EXIT:   None.
```

ZSTTRC

```
ZSTTRC, Move String into CIR/Record
ARGS:   DATA,LENGTH,VDESC
DESC:   ZSTTRC transfers a character string into a CIR or
        data record.
ENTRY:  DATA (B * n)  Character string to transfer to record.
        LENGTH(I * 4) Number of characters in string.
        VDESC (D * 8) Variable descriptor of variable to receive
                      value.
EXIT:   None.
```

ZSTXKY

```
ZSTXKY, Enter String Value into Key
ARGS:   DATA,ORDINAL,LENGTH
DESC:   ZSTXKY is called after one of the key initialisation
        routines in order to insert a character value into
        the next location of the key currently being defined.
ENTRY:  DATA (B * n)   Character string to insert into key.
        ORDINAL(I * 4) Starting byte number in area DATA to
                       transfer the value from. For a simple
                       variable this value is 1.
        LENGTH(I * 4)  Number of characters in string.
EXIT:   None.
```

ZSTXRC

```
ZSTXRC, Move String into CIR/Record
ARGS:   DATA,ORDINAL,LENGTH,VDESC
DESC:   ZSTXRC transfers a character string into a CIR or
        data record.
ENTRY:  DATA (B * n)   Character string to transfer to record.
        ORDINAL(I * 4) Starting byte number in area DATA to
                       transfer the value from. For a simple
                       variable this value is 1.
        LENGTH(I * 4)  Number of bytes in value.
        VDESC (D * 8)  Variable descriptor of variable to
                       receive value.
EXIT:   None.
```

ZTHRU

```
ZTHRU,  Start Creation of THRU Key
ARGS:   DUMMY
DESC:   ZTHRU starts the creation of a key.  It creates a
        "THRU" key.  That is, a key which is used to select
        all CIR/record's whose key matches or comes before
        the key that is currently being defined.  It is
        normally called after one of the case/record level
        initialisation routines in order to initialise the
        key selection options for case or record loops.
        Following a call to ZTHRU other routines (described
        below) are called to enter values into the key one
        at a time.  For a case key only one call is made to
        enter the value of the case id.  For record keys,
        the case id is assumed to be the same as the last
        CIR retrieved or restored.  Therefore, only the
        record sort ids have to be inserted into the key.
        The sort-ids must be entered in the order of their
        appearance in the key being created.
ENTRY:  DUMMY(I * 4) Dummy argument to make routine a
                     syntactically correct FORTRAN function.
                     Should always be 0.
EXIT:   None.
```

ZTIME

```
ZTIME,  Return Current Date and Time as Integers.
ARGS:   IDAYS,ISECS
DESC:   ZTIME returns the current date and time as standard
        julian date and time values.  It returns the same
        values as the DBMS functions TODAY(0) and NOW(0).
ENTRY:  None.
EXIT:   IDAYS (I * 4) Number of days since Oct 15, 1582.
        ISECS (I * 4)  Number of seconds since midnight.
```

ZTMTKY

```
ZTMTKY, Enter Time String into Key
ARGS:   TIMEST,LENGTH,TIMEMP
DESC:   ZTMTKY is called after one of the key initialisation
        routines in order to insert the value of a time
        string into the next location of the key currently
        being defined.
ENTRY:  TIMEST (B * n) Time string to insert into key.
        LENGTH (I * 4) Number of characters in strings TIMEST
                       and TIMEMP.
        TIMEMP (B * n) String containing the decoding format
                       for the time string contained in
                       TIMEST.  Legal values are 1(ignore),
                       H(hour), M(minute), S(second).  For
                       example: 'HHMMSS'.
EXIT:   None.
```

ZTMTRC

```
ZTMTRC, Move Time into CIR/Record
ARGS:   TIMEST,LENGTH,TIMEMP,VDESC
DESC:   ZTMTRC transfers the value of a time string into a
        CIR or data record.
ENTRY:  TIMEST (B * n) String containing the time value to transfer
                       to record.
        LENGTH (I * 4) Number of characters in strings TIMEST
                       and TIMEMP.
        TIMEMP (B * n) String containing the decoding format for
                       the time string contained in TIMEST.
                       Legal values are I(ignore), H(hour),
                       M(minute), S(second).
        VDESC (D * 8)  Variable descriptor of variables to receive
                       value.
EXIT:   None.
```

ZTMXIN

```
ZTMXIN, Convert Time String into Integer
ARGS:   TIMEST,ORDINAL,TIMEMP,LENGTH
DESC:   ZTMXIN converts a time string into a time integer
        value.
ENTRY:  TIMEST (B * n) Time string to convert.
        ORDINAL(I * 4) First character in string TIMEST to use.
        TIMEMP (B * n) String containing the decoding format for
                       the time string contained in TIMEST.
                       Legal values are I(ignore), H(hour), M(minute),
                       S(second).  For example: 'HHMMSS'.
        LENGTH (I * 4) Number of characters in strings above.
EXIT:   None.
RETURN: The integer equivalent of the time string (seconds
        since midnight), negative if error.
```

ZTMXKY

```
ZTMXKY, Enter Time String into Key
ARGS:   TIMFST,ORDINAL,LENGTH,TIMEMP
DESC:   ZTMXKY is called after one of the key initialisation
        routines in order to insert the value of a time
        string into the next location of the key currently
        being defined.
ENTRY:  TIMEST (B * n) Time string to insert into key.
        ORDINAL(I * 4) Starting byte number in area DATA to
                       transfer the value from. For a simple vari-
                       able this value is 1.
        LENGTH (I * 4) Number of characters in strings above.
        TIMEMP (B * n) String containing the decoding format for
                       the time string contained in TIMEST.
                       Legal values are I(ignore), H(hour),
                       M(minute), S(second). For example:
                       'HHMMSS'.
EXIT:   None.
```

ZTMXRC

```
ZTMXRC, Move Time into CIR/Record
ARGS:   TIMEST,ORDINAL,LENGTH,TIMEMP,VDESC
DESC:   ZTMXRC transfers the value of a time string into a
        CIR or data record.
ENTRY:  TIMEST (B * n) String containing the time value to transfer
                       to record.
        ORDINAL(I * 4) Starting byte number in area DATA to
                       transfer the value from. For a simple vari-
                       able this value is 1.
        LENGTH (I * 4) Number of characters in strings above.
        TIMEMP (B * n) String containing the decoding format for
                       the time string contained in TIMEST.
                       Legal values are 1(ignore), H(hour),
                       M(minute), S(second).
        VDESC (D * 8)  Variable descriptor of variables to receive
                       value.
EXIT:   None.
```

ZUNTIL

```
ZUNTIL, Start Creation of UNTIL Key
ARGS:   DUMMY
DESC:   ZUNTIL starts the creation of a key.  It creates an
        "UNTIL" key.  That is, a key which is used to select
        all CIR/record's whose key comes before the key that
        is currently being defined.  It is normally called
        after one of the case/record level initialisation
        routines in order to initialise the key selection
        options for case or record loops.  Following a call
        to ZUNTIL other routines (described below) are
        called to enter values into the key one at a time.
        For a case key only one call is made to enter the
        value of the case id.  For record keys, the case id
        is assumed to be the same as the last CIR retrieved
        or restored.  Therefore, only the record sort ids
        have to be inserted into the key.  The sort-ids must
        be entered in the order of their appearance in the
        key being created.
ENTRY:  DUMMY(I * 4) Dummy argument to make routine a
                     syntactically correct FORTRAN function.
                     Should always be 0.
EXIT:   None.
```

ZUPLEV

```
ZUPLEV, Get Specified Database Update Level
ARGS:   DBNAME
DESC:   ZUPLEV returns the current update level of the
        database prior to this update.
ENTRY:  DBNAME(N*8) The name of the database.
EXIT:   None.
```

ZUSER

```
ZUSER,  Set to Different Stream
ARGS:   USERNO
DESC:   ZUSER saves the current stream's retrieval stack and
        reactivates the specified stream's stack.
ENTRY:  USERNO(I * 4) The number of the stream to process next.
                      Range is from I to maxnum specified on
                      HOST initialisation call.
EXIT:   None.
```

ZVARLB

```
ZVARLB, Get Label of Variable
ARGS:   VDESC,STRING,LENGTH
DESC:   ZVARLB searches the database and return the variable
        label for a specified variable.
ENTRY:  VDESC (D * 8)  The descriptor of the variable.
        LENGTH (I * 4) The number of characters of the label to
                       transfer to STRING.
EXIT:   STRING (B * n) The area which will receive the
        variable label.
RETURN: Number of characters actually transferred, negative
        if error.
```

ZVERS

```
ZVERS,  Return Version, Revision Numbers and Levels of
        HOST
ARGS:   VERNUM, VERLEV, REVNUM, REVLEV
DESC:   ZVERS returns the version and revision number and
        levels of the HOST package that the user is
        currently using.  It also returns an indication of
        whether it is regular or concurrent HOST.
ENTRY:  None.
EXIT:   VERNUM (I * 4) Version number.
        VERLEV (I * 4) Version level.
        REVNUM (I * 4) Revision number.
        REVLEV (I * 4) Revision level.
RETURN: 0 if regular HOST,1 if concurrent  HOST.
```

ZVNAME

```
ZVNAME, Get Name for Variable Descriptor
ARGS:   VDESC,VNAME
DESC:   ZVNAME looks up a common or record variable
        descriptor and returns the variable's name.
ENTRY:  VDESC (D * 8) The variable descriptor.
EXIT:   VNAME (N * 8) contains the variable name
                      corresponding to the variable
                      descriptor specified.
```

ZVTYPE

```
ZVTYPE, Get Type of Variable
ARGS:   VDESC,LENGTH
DESC:   ZVTYPE returns the type of a specified variable.
ENTRY:  VDESC (D * 8) The descriptor of the variable to get
                      the type for.
EXIT:   LENGTH(I * 4) The size of the variable in bytes.
RETURN: The code of the type of the specified variable,
        negative if error.
        CODE     TYPE
           1     String variable
           2     Categorical integer variable
           3     Date integer variable
           4     Time Integer variable
           5     Integer variable
           6     Real variable
```

ZWITH

```
ZWITH,  Start Creation of WITH Key
ARGS:   DUMMY
DESC:   ZWITH starts the creation of a key.  It creates a
        "WITH" key.  ZWITH is used with CASE/RECORD IS
        levels to specify a key for a single CIR/record.  It
        can also be used with the PROCESS RECORD level to
        specify the first n sort-ids of a key, which causes
        all records with the same first n sort-ids to be
        selected in order.  It is normally called after one
        of the case/record level initialisation routines in
        order to initialise the key selection options for
        case or record loops.  Following a call to ZWITH
        other routines (described below) are called to enter
        values into the key one at a time.  For a case key
        only one call is made to enter the value of the case
        id.  For record keys, the case id is assumed to be
        the same as the last CIR retrieved or restored.
        Therefore, only the record sort ids have to be
        inserted into the key.  The sort-ids must be entered
        in the order of their appearance in the key being
        created.
ENTRY:  DUMMY(I * 4) Dummy argument to make routine a
                     syntactically correct FORTRAN function.
                     Should always be 0.
EXIT:   None.
```

Program Layout

Column four in the typical HOST program layout on below identify options on the basic steps through which a typical HOST program moves. The comments in the examples refer to these steps.

A Typical HOST Program Layout

```
Step 1     Initialise the system:          Call ZSTART   (1.A)

           and login into MASTER if        Call ZLOGIN   (1.B)
           necessary:

Step 2     Open the database:              Call ZORDB or
                                           ZOSDB

Step 3     Start a case block on the       Call
           stack:                          ZCCNT(D) or
                                           ZCIS(D) or
                                           ZCSAM(D)

Step 4     If no key processing is done,
           skip to step 5.

           If a single value is used for   Call ZWITH    (4.A)
           the key( "WITH" ) then:
              Call key definition routines              (4.B)
              in the order the sort-ids
              appear in schema definition.
              Skip to step 5.

          If a range of keys is to be      Call ZFROM or (4.C)
          specified and a low limit is re-      ZAFTER
          quired:
              Call key definition routines              (4.D)
              in the order that the sort-ids
              appear in schema definition.

          If no high limit on the key
          range is required, skip to step
          5.

          If a range of keys is to be      Call ZTHRU or (4.E)
          specified and a high limit is    ZUNTIL
          required:
              Call key definition routines in           (4.F)
              the order the sort-ids appear
              in schema definition.

Step 5   End the key processing and        Call ZCNEXT
         retrieve one case in range.       or ZCPREV or
```

		ZCFRST or ZCLAST	
	If there are cases to delete	Call ZCDEL	
	If there are cases to be processed then continue with Step 6, otherwise skip to Step 13 to end the case-block processing.		
Step 6	For the case level processings, call routines that transfer data to and from CIR variables.		
Step 7	Start a record processing block:	Call ZRCNT(D) or ZRIS(D) or ZRSAM(D)	
Step 8	If no key processing is done, skip to step 9.		
	If a single value is used for the key "WITH" then:	Call ZWITH	(8.A)
	Call key definition routines in the order the sort-ids appear in schema definition.		(8.B)
	Skip to step 9.		
	If a range of keys is to be specified and a low range is required then:	Call ZFROM or ZAFTER	(8.C)
	Call key definition routines in the order that the sort-ids appear in schema definition. If no high limit on the key range is required, skip to step 9.		(8.D)
	If a range of keys is to be specified and a high limit is required:	Call ZTHRU or ZUNTIL	(8.E)
	Call key definition routines in the order that the sort-ids appear in schema definition.		(8.F)
Step 9	End the key processing and retrieve one record in range:	Call ZRNEXT or ZRPREV or ZRFRST or ZRIAST	
	If there are records to be processed, then continue with Step 10. Otherwise, skip to Step 11.		

```
Step 10 Record level processings.                        (10.A)
        Call routines that transfer
        data to and from the record
        variables.

        If the record is to be deleted     Call ZRDEL    (10.B)
        then:
        Continue with Step 9 to
        process a new record.

Step 11 End record-block processing:       Call ZREXIT
Step 12 Perform additional case            Call routines (12.A)
        processing.                        that transfer
                                           data to and from
                                           CIR variables.

        If the case is to be deleted       Call ZCDEL    (12.B)
        then:
        Continue with Step 5 to
        process a new case.

Step 13 End of case-block processing:      Call ZCEXIT

Step 14 Close the database down:           Call ZENDDB

Step 15 Shut down the whole HOST           Call ZEND
        system:
```

Another Typical HOST Program Layout

```
.

(1) Call ZSTART then ZLOGIN

(2) Call ZORDB or ZOSDB

    (3) Call ZCGDMY

    (4) Call routines to transfer values into case-ids variables
    (5) Call ZCFIND
    (6) Ordinary case processing calls
    (7) If case is brand new and update mode: Call ZCWRIT

        (8) Call ZRGDMY

        (9) Call routines to transfer values Into record-ids
            variables
        (10) Call ZRFIND
        (11) Ordinary record processing calls
        (12) K record is to be written call ZRWRIT If record
             is to be deleted call ZRDEL
        (13) To process a new record call ZRREST and
             continue with (9)
        (14) Call ZREXIT when record processing is over

    (15) Ordinary case processing calls
```

```
    (16) If CIR variables were modified and/or records were
         added/deleted call ZCWRIT
         if case is to be deleted call ZCDEL
    (17) To process a new case call ZCREST and continue with (4)

    (18) Call ZCEXUT when case processing Is over

(19) Call ZENDDB

(20) Call ZEND
```

A Note on Error Checking

This chapter contains several examples of PQL retrieval programs and their FORTRAN counterparts using HOST subroutine calls. Every HOST function call always returns a value. In the following examples, this value is stored in variable 'IERR'. **This value should be checked after each function call** in case an error has been detected by the routine. To continue the program after an error has been generated may damage the databases that the program accesses.

The examples below do not do this error checking. This is for readability only. It is not suggested programming practice.

Print the Value of a Variable In a Record

DBMS Retrieval Version

```
OLD FILE  MOTHERS
PASSWORD  LOVE
SECURITY  RS1, WS1
C
C PRINTS THE STATUS OF PATIENT 1 0001.
C THE DATA IS CONTAINED IN RECORD TYPE 47
C WITH SORT IDS 3 AND 5
C
RETRIEVAL
. OLD CASE  IS 10001
.   OLD RECORD IS 17 (3,5)
.     WRITE  'PATIENT 10001 STATUS IS' STATUS1
.   END RECORD IS
. END CASE IS
END RETRIEVAL
```

HOST Retrieval Version

```
C        IN THE FOLLOWING ROUTINE EACH FUNCTION RETURNS AN
C        ERRORVALUE AND THAT VALUE IS STORED IN VARIABLE 'IERR', IN
C        COMMON BLOCK 'HERROR'. THE FUNCTION NAME IS STORED
C        IN VARIABLE 'ZZNAME', IN THE SAME COMMON BLOCK.

      IMPLICIT INTEGER*4 (Z)
      .
      .
      .
      CHARACTER*8           DBNAME
      CHARACTER*8           DBPASS
      CHARACTER*S           HSPASS
      CHARACTER*8           RDPASS
```

```
      CHARACTER*8           WRPASS
      CHARACTER*8           VNSTAT
C
      CHARACTER*5           PREFIX
      CHARACTER*6           MDSN
      CHARACTER*10          SDSN
C
      REAL*8                VDSTAT
C
      INTEGER*4             DUMMY
      INTEGER*4             TSPACE
C
C FOR ERROR PROCESSING
C
      REAL*8                ZZNAME
      INTEGER*4             IERR
      INTEGER*4             IDUMMY
      COMMON                /HERROR/ ZZNAME,IERR,IDUMMY
C
C
C
      DATA            DBNAME /'MOTHERS '/
      DATA            DBPASS /'LOVE '/
      DATA            HSPASS /'HOSTOKAY'/
      DATA            RDPASS /'RS1 '/
      DATA            WRPASS /'WS1 '/
      DATA            VNSTAT /'STATUS'/
C
      DATA            PREFIX/'[SIR]'/
      DATA            MDSN /'MASTER'/
      DATA            SDSN /'MY_PROGRAM'/
C
C START HOST SYSTEM:                              STEP 1.A
C
      IF(ZSTART( 1,1,5000,0).LT.0) STOP 300
C
C LOG INTO MASTER:                                STEP 1.B
C
      IF(ZLOGIN(MDSN,LEN(MDSN),SDSN,LEN(SDSN)).LT.0) GOTO 200
C
C ATTACH DATABASE NEEDED FOR RUN:                 STEP 2
C
      IF (ZORDB(DBNAME,DBPASS,HSPASS,RDPASS,WRPASS,0,
     *PREFIX,LEN(PREFIX) ).LT.0) GOTO 200
C
C START A "CASE IS" LEVEL:                        STEP 3
C
      IF(ZCIS(0, 1 ) LT.0) GOTO 200
C
C CREATE A "WITH" KEY:                            STEP 4.A
C
      IF(ZWITH(0).LT.0) GOTO 200
C
C DEFINE THE KEY:                                 STEP 4.B
C
      IF(ZINTKY(10001 ).LT.0) GOTO 200
C
```

```
C GET THE CASE(FOR SURE, IT IS THERE!):          STEP 5
C
      IF(ZCNEXT(0).LT.0) GOTO 200
C
C START A "RECORD IS" LEVEL:                     STEP 7
C
      IF(ZRIS(17,0,1 ).LT.0) GOTO 200
C
C CREATE A "WITH" KEY:                           STEP 8.A
C
      IF(ZWITH(0).LT.0) GOTO 200
C
C DEFINE THE KEY:                                STEP 8.B
C
      IF(ZINTKY(3).LT.0) GOTO 200
      IF(ZINTKY(5).LT.0) GOTO 200
C
C GET THE RECORD(FOR SURE, IT IS THERE!):        STEP 9
C
      IF(ZRNEXT(0).LT.0) GOTO 200
C
C BUILD A DESCRIPTOR FOR VARIABLE
C
      IF(ZDESCO(VDSTAT,DBNAME,17,VNSTAT,0).LT.0) GOTO 200
C
C RETRIEVE THE VALUE (FOR SURE, ISDEFINED!):     STEP 10.A
C
      IF(ZRCTIN(VDSTAT,I).LT.0) GOTO 200
      PRINT 100,1
100   FORMAT('PATIENT 10001 STATUS IS',I5)
C
C END OF RECORD IS LEVEL:                        STEP 11
C
      IF(ZREXIT(0).LT.0) GOTO 200
C
C END OF CASE IS LEVEL:                          STEP 13
C
      IF(ZCEXIT(0).LT.0) GOTO 200
C
C CLOSE THE DATABASE:                            STEP 14
C
      IF(ZENDDB(DBNAME).LT.0) GOTO 200
C
C SHUT DOWN HOST:                                STEP 15
C
150   IF(ZEND(TSPACE).LT.0) STOP 400
      GOTO 1000
C
C ERROR PROCESSING SECTION
C
200   PRINT 201, ZZNAME,IERR
201   FORMAT(1X,A8,' FAILED WITH ERROR CODE',I4) GOTO 150
1000  STOP
END
```

Retrieval Update with RECORD IS Nested within a PROCESS CASE ALL

DBMS Retrieval Version

```
OLD FILE  MOTHERS
PASSWORD  LOVE
SECURITY  RS1,WS1
C         PROCESS ALL CASES IN THE DATABASE
C         IF VARIABLE 'SICK' IN RECORD TYPE 16 IS
C         GREATER THAN 0 SET 'SICK' EQUAL TO 1
RETRIEVAL UPDATE
.  PROCESS CASES ALL
.    OLD RECORD IS 16
.      IFTHEN (SICK GT 0)
.        COMPUTE SICK = 1
.      ENDIF
.    END RECORD IS
.  END PROCESS CASES
END RETRIEVAL
```

HOST Retrieval Version

```
C            IN THE FOLLOWING ROUTINE EACH FUNCTION RETURNS AN
C            ERROR VALUE THAT IS PROCESSED BY 'ZCALL'.
C
      IMPLICIT INTEGER*4 (Z)
      CHARACTER*8       DBNAME
      CHARACTER*8       DBPASS
      CHARACTER*8       HSPASS
      CHARACTER*8       RDPASS
      CHARACTER*8       WRPASS
      CHARACTER*8       VNSTAT
  C
      CHARACTER*5       PREFIX
  C
      REAL*8            VDSTAT
  C
      INTEGER*4         DUMMY
      INTEGER*4         TSPACE
  C
  C
  C
      DATA        DBNAME /'MOTHERS '/
      DATA        DBPASS /'LOVE '/
      DATA        HSPASS /'HOSTOKAY'/
      DATA        RDPASS /'RS1 '/
      DATA        WRPASS /'WS1 '/
      DATA        VNSTAT /'STATUS'/
  C
      DATA        PREFIX /'[SIR]'/
```

```
C
C START HOST SYSTEM:                                STEP 1.A
C
100   IERR= ZCALL(ZSTART(1,1,5000,0),2,-2,100,0,0)
C
C ATTACH REQUIRED DBMS FILES:       STEP 2
C
200   IERR= ZCALL(ZORDB(DBNAME,DBPASS,HSPASS,RDPASS,
     *WRPASS,L,PREFIX,LEN(PREFIX)),2,-2,200,0,0)
C
C GET VARIABLE DESCRIPTOR OF VARIABLE 'SICK'
C            FOR USE LATER
C
300   IERR= ZCALL(ZDESCD(VDSTAT,DBNAME,16,VNSTAT,0),2,-2,300,0,0)
C
C DO PROCESS CASES ALL LEVEL:                       STEP 3
C
400   IERR= ZCALL(ZCCNT(-I,1,1),2,-2,400,0,0)
C
C GET THE CASE:                                     STEP 5
C
500   IF(ZCALL(ZCNEXT(0),2,-2,500,-4002,-4001).LT.0)GOT01100
C
C DO RECORD IS LEVEL:                               STEP 7
C
600   IERR=ZCALL(ZRIS(16,0,1),2,-2,600,0,0)
C
C GET THE RECORD:                                   STEP 9
C
700   IF(ZCALL(ZRNEXT(L),2,-2,700,-4002,-4001).LT.0)GOT01000
C
C RETRIEVE VALUE AND UPDATE IT IF NECESSARY:
C                                                   STEP 10
C
800   IF(ZCALL(ZRCTIN(VDSTAT,ISICK).2,-2,800,-5008,(-5005).LT.0) GOTO
1000
      IF (ISICK.LE.0) GOTO 1000
      I = 1
900   IERR= ZCALL(ZINTRC(I,VDSTAT),2,-2,900,-5008,-5005)
C
C END OF RECORD IS LEVEL:                           STEP 11
C
1000  IERR= ZCALL(ZREXIT(0),2,-2,1000,0,0)
C
C            CONTINUE WITH STEP 5
C
      GOTO 500
C
C END OF PROCESS CASE LOOP:                         STEP 13
C
1100  IERR= ZCALL(ZCEXIT(0),2,-2,1100,0,0)
C
C CLOSE THE DATABASE:                               STEP 14
C
1200  IERR= ZCALL(ZENDDB(DBNAME),2,-2,1200,0,0)
C
C CLOSE HOST SYSTEM:                                STEP 15
```

```
C
1300  IERR= ZCALL(ZEND(TSPACE),2,-2,1300,0,0)
```

RECORD IS for a Caseless Database

DBMS Retrieval Version

```
OLD FILE  MOTHERS
PASSWORD  LOVE
SECURITY  RS1,WS1
C PROCESS ALL RECORD TYPE 16 IN THE DATABASE IF VARIABLE
C 'SICK' IS GREATER THAN 0, SET 'SICK' EQUAL TO 1
RETRIEVAL UPDATE
. PROCESS RECORD 16
.   IFTHEN (SICK GT 0)
.     COMPUTE SICK = 1
.   ENDIF
. END RECORD IS
END RETRIEVAL
```

HOST Retrieval Version

```
C IN THE FOLLOWING ROUTINE EACH FUNCTION RETURNS AN
C ERROR VALUE THAT IS PROCESSED BY 'ZCALL'.

      IMPLICIT INTEGER*4 (Z)
      .
      .
      .
      CHARACTER*8          DBNAME
      CHARACTER*8          DBPASS
      CHARACTER*8          HSPASS
      CHARACTER*8          RDPASS
      CHARACTER*8          WRPASS
      CHARACTER*8          VNSTAT
C
      CHARACTER*5          PREFIX
C
      REAL*8               VDSTAT
C
      INTEGER*4            DUMMY
      INTEGER*4            TSPACE
C
C
C
      DATA        DBNAME /'MOTHERS '/
      DATA        DBPASS /'LOVE'/
      DATA        HSPASS /'HOSTOKAY'/
      DATA        RDPASS /'RS1 '/
      DATA        WRPASS /'WS1 '/
      DATA        VNSTAT /'STATUS'/
C
```

```
      DATA        PREFIX /'[SIR]'/
C
C START HOST SYSTEM:                              STEP 1.A
C
100   IERR= ZCALL(ZSTART(1,1,5000,0),2,-2,100,0,0)
C
C ATTACH REQUIRED DBMS FILES:                     STEP 2
C
200   IERR= ZCALL(ZORDB(DBNAME,DBPASS,HSPASS,RDPASS,
     *WRPASS,L,PREFIX,LEN(PREFIX)),2,-2,200,0,0)
C
C GET VARIABLE DESCRIPTOR OF VARIABLE'SICK'
C              FOR USE LATER
C
300   IERR= ZCALL(ZDESCD(VDSTAT,DBNAME,16,VNSTAT,0),2,-2,300,0,0)
C
C              DO PROCESS RECORD LEVEL:           STEP 7
C
600   IERR= ZCALL(ZRCNT(16,-1,1,1),2,-2,600,0,0)
C
C GET THE RECORD:                                 STEP 9
C
700   IF(ZCALL(ZRNEXT(1),2,-2,700,-4002,-4001).LT.0)GOT0 1000
C
C RETRIEVE VALUE AND UPDATE IT IF NECESSARY :     STEP10
C
800   IF(ZCALL(ZRCTIN(VDSTAT,ISICK),2,-2,800,-5008,5005).LT.0) GOTO 700
      IF (ISICK.LE.0) GOTO 700
      I = 1
900   IERR= ZCALL(ZINTRC(I,VDSTAT),2,-2,900,-5008,-5005)
C
C CONTINUE WITH STEP 9
C
      GOTO 700
C
C END OF RECORD IS LEVEL:                         STEP 11
C
1000  IERR= ZCALL(ZREXIT(0),2,-2,1000,0,0)
C
C              CLOSE THE DATABASE:                STEP 14
C
1200  IERR= ZCALL(ZENDDB(DBNAME),2,-2,1200,0,0)
C
C              CLOSE HOST SYSTEM:                 STEP 15
C
1300  IERR= ZCALL(ZEND(TSPACE),2,-2,1300,0,0)
```

Multiple Nested Network Retrieval

DBMS Retrieval Version

```
OLD FILE  MOTHERS
PASSWORD  LOVE
SECURITY  RS1,WS1
```

```
C RECORD TYPE 1 RECORDS ARE PATIENTS IN THE STUDY.
C RECORD TYPE 2 RECORDS ARE CONTROLS FOR PATIENTS.
C
C EACH PATIENT HAS A CONTROL WHOSE CASE ID IS'IDPOINTR'
C AND RECORD TYPE 2 SORT ID IS'RECPOINT'.
C
C PRINT THE NUMBER OF CONTROLS WHOSE VALUE OF VARIABLE
C 'CNTLSTAT' IS LESS THAN THE PATIENT'S VARIABLE'PATSTAT'.
RETRIEVAL
. PROCESS CASES ALL
.   COMPUTE CNT = 0
.   PROCESS RECORD 1
.     MOVE VARS IDPOINTR RECPOINT PATSTAT
.     OLD CASE IS IDPOINTR
.       OLD RECORD IS 2 (RECPOINT)
.         IFTHEN (CNTLSTAT LT PATSTAT)
.           COMPUTE CNT = CNT + 1
.         ENDIF
.       END RECORD IS
.     END CASEIS
.   END PROCESS RECORD
. END PROCESS CASE
. WRITE CNT 'CONTROLS ARE BETTER THAN CURRENT PATIENTS.'
END RETRIEVAL
```

HOST Retrieval Version - Function C

```
C IN THE FOLLOWING ROUTINE EACH FUNCTION RETURNS A CALL
C ERROR VALUE AND THAT VALUE IS STORED IN VARIABLE'IERR'.
C THIS VARIABLE SHOULD BE CHECKED SOMEHOW AFTER EACH
C FUNCTION CALL, HOWEVER, IN ORDER TO IMPROVE THE
C READABILITY OF THE EXAMPLE THE TEST HAS BEEN OMITTED.
      IMPLICIT INTEGER*4 (Z)
      .
      .
      .
      CHARACTER*8      DBNAME
      CHARACTER*8      DBPASS
      CHARACTER*8      HSPASS
      CHARACTER*8      RDPASS
      CHARACTER*8      WRPASS
      CHARACTER*8      VNIDPT
      CHARACTER*8      VNRECP
      CHARACTER*8      VNPATS
      CHARACTER*8      VNCNTL
C
      CHARACTER*5      PREFIX
C
      REAL*8           VDIDPT
      REAL*8           VDRECP
      REAL*8           VDPATS
      REAL*8           VDCNTL
C
      INTEGER*4        DUMMY
      INTEGER*4        IERR
```

```
      INTEGER*4        TSUSED
C
C
C
      DATA  DBNAME /'MOTHERS'/
      DATA  DBPASS /'LOVE'/
      DATA  HSPASS /'HOSTOKAY'/
      DATA  PREFIX /'[SIR]'/
      DATA  RDPASS /'RS1 '/
      DATA  WRPASS /'WS1 '/
      DATA  VNIDPT /'IDPOINTR'/
      DATA  VNRECP /'RECPOINT'/
      DATA  VNPATS /'PATSTAT'/
      DATA  VNCNTL /'CNTLSTAT'/


C
C START HOST SYSTEM:                                STEP 1
C
      IERR = ZSTART(1,1,5000,0)
C
C ATTACH REQUIRED DBMS FILES:       STEP 2
C
      IERR = ZORDB(DBNAME,DBPASS,HSPASS,RDPASS,WRPASS,
     *0,PREFIX,LEN(PREFIX)
C
C GET VARIABLE DESCRIPTORS FOR REQUIRED VARIABLES CONCE
C
      IERR = ZDESCD(VDIDPT,DBNAME,1,VNIDPT,0)
      IERR = ZDESCD(VDRECP,DBNAME,1,VNRECP,0)
      IERR = ZDESCD(VDPATS,DBNAME,1,VNPATS,0)
      IERR = ZDESCD(VDCNTL,DBNAME,2,VNCNTL,0)
C
C DO PROCESS CASES ALL LEVELR
C                                                   STEP 3
      IERR = ZCCNT(-1,1,1)
C                                                   STEP 5
1000  IERR = ZCNEXT(0)
      CNT = 0
C
C IF NO CASES LEFT, SKIP TO STEP 13
C
      IF (IERR.LT.0) GOTO 6000
C
C DO PROCESS RECORD 1 LEVEL
C
C                                                   STEP 7
      IERR = ZRCNT(1,-1,1,1)
C                                                   STEP 9
2000  IERR = ZRNEXT(0)
C
C IF NO RECORDS LEFT, SKIP TO STEP 11
C
      IF (IERR.LT.0) GOTO 5000
C
C DO MOVE VAR STATEMENT
C
```

```
C                                                       STEP 10.A
      IERR = ZRCTIN(VDIDPT,IDPNTR)
      IERR = ZRCTIN(VDRECP,RCPNTR)
      IERR = ZRCTFP(VDPATS,PATSTT)
C
C DO CASE IS STATEMENT
C                                                       STEP 3
      IERR = ZCIS(0,I)
C                                                       STEP 4.A
      IERR = ZWITH(0)
C                                                       STEP 4.B
      IERR = ZINTKY(IDPNTR)
C                                                       STEP 5
      IERR = ZCNEXT(0)
C
C IF NO CASES LEFT, SKIP TO STEP 13
C
      IF (IERR.LT.0) GOTO 4000
C
C DO RECORD IS STATEMENT
C                                                       STEP 7
      IERR = ZRIS(2,0,1
C                                                       STEP 8.A
      IERR = ZWITH(0)
C                                                       STEP 8.B
      IERR = ZINTKY(RCPNTR)
C                                                       STEP 9
      IERR = ZRNEXT(0)
C
C IF NO RECORDS LEFT, SKIP TO STEP 11
C
      IF ( IERR.LT.0) GOTO 3000
C
C INCREMENT CNT AFTER TEST
C
C                                                       STEP 10.A
      IERR = ZRCTFP(VDCNTL,CNTSTT)
      IF (CNTSTT.LT.PATSTT) CNT = CNT + 1
C
C DO END RECORD IS
C
C                                                       STEP 11
3000  IERR = ZREXIT(0)
C
C   DO END CASE IS
C
C                                                       STEP 13
4000  IERR = ZCEXIT(0)
C
C   LOOP OVER INNER CASE BLOCK
C
      GOTO 2000

C
C DO END PROCESS REC
C                                                       STEP 11
C
```

```
5000  IERR = ZREXIT(0)
C
C CONTINUE WITH STEP 5 TO PROCESS A NEW CASE
C
      GOTO 1000
C
C DO END PROCESS CASE
C                                                  STEP 13
C
6000  IERR = ZCEXIT(0)
C
C END OF RETRIEVAL PRINT RESULT
C
      PRINT 100, CNT
100   FORMAT(I6,'CONTROLS ARE BETTER THAN CURRENT PATIENTS.')
C                                                  STEP 14
C
      IERR = ZENODB(DBNAME)                        STEP 15
C
      IERR = ZEND(TSPACE)
```

Reserved Entry Point Names and Common Blocks

The following names are reserved for use by `HOST` and should not be used in any application program:

```
  ABORT  ADDRES ALLOC  ASSIGN BST    BSTN   BYRSIZ CACHFL CACHMS CACHSH
  CACHST CACHXX CANEXT CHKLOK CHNGDB CL32   CLISF  CLOS   CM0808 CM0832
  CM3232 CMPC1  CMPI4  CMPRS  COMPRS CONSTR CONSTX CPJOIN CPYFIL CPYIND
  CPYREC CPYWDS CRACIQ CRACKD CRACKE CRACKF CRACKP CRACKS CREATK CRKGRP
  CVTGER CVTPUT DALLOC DBWRI  DCLEXT DCRYPR DEBUGT DEBUGU DSSIGN DUMPDB

  ENDKEY ENDRUN ESCAPE EXCRPV EXITTK EXPCI  EXPHO  EXPI4  EXPRS  EXTND
  FBREAK FILKEY FRANF  FREECH GERATR GERATX GERHLP GERMEM GERMOR GERWDS
  GETKEY GETRCX GETREC GTABNM GTAWDS HELP   IBREAK ICHECK ICLOSE IISRTW
  INTJUL INVERT INWR   IOPENX IRWSF  ISFIIL ISFPDL ISFSEO ISIRCM ISIRGT
  ISIRPT ISKPRC ISKPWD ISOPEN ISPCHR ISPEWD ISPF   ISPFAM ISPFI  ISPFIL

  ISPFIO ISPFWD ISPFWO ISPMAP ISPMEM ISPNAM ISPNUM ISPREL ISPSKB ISPSKC
  ISPSKP ISRTC  ISRTE  ISRTO  ISRTR  ISST   IST    ISTN   ISTRBE ISTRCM
  ISTRPT ISTRSB JNTJUL JRNOOD JRNOPN JULINT KEYCOM LDDUMR LDESORT
LOCATE
  LODMAP LODNAM LRECL  MATCH1 MATCH2 MATCHC MATCHF MATCHR MOVEAA MOVEAB
  MOVEAL MOVEAR MOVEB  MOVEB1 MOVEBA MOVEBC MOVEBI MOVEBL MOVEBN MOVECA

  MOVECB MOVECT MOVED  MOVEDA MOVEDN MOVEDO MOVEHD MOVEIB MOVELA MOVELC
  MOVERA MOVEUA MOVEZB MOVEZD MOVREV MRKDLT MUT    MUTN   MV0101 MV0808
  MV0832 MV3232 NCRYPT NL32   NLO8   ONVRT  OPAUSE OPENDB OPISF  OPSYMA
  OPSYSA OPSYSD OPSYSE OPSYSM OPSYSP OPSYSR OPSYST OPSYSX PCNSTR PCNSTX
  PIBBRK PIBCUR PIBDTX PIBSTR PIBTAB PIBTIM PIBTMX PISDAT POSREC PRINTB

  PRINTC PRINTK PRINTR PRNTCA PRNTTA PROMPT PSBBAK PSBCHI PSBCHR PSBDAT
  PSBDTM PSBDTX PSBFAM PSBFMT PSBI   PSBID  PSBMEM PSBPAR PSBQOT PSBSTI
  PSBSTQ PSBSTR PSBSW  PSBTAB PSBTIM PSBTMX PSBTRM PSBWRC PUTBLK PUTKEY
  PUTRCX PUTREC PUTWDS RDISF  READ1  READB  READC  READK  READN  REMARK
  REMRKN REMRKV REOPEN REWIND RMTIO  RMTKY  RSTCLO RSTLOD RSTOPN RSTRD

  RSTRDT RSTREC RSTSER RSTTRN RWRISF SBYTPR SCPFLT SCPINT SCRTCH SDLISF
  SETISF SETMEM SHUTDN SIRACT SKPISF SLEEP  SPIOPN SPIOPR SPIOPS SPIRES
  SPISAV SPSCLS SPSEOL SPSINO SPSINT SPSISF SPSISN SPSISO SPSLDI SPSRES
  SPSSAV SPSZD  SRTNXT SRTOPN SRTPOS SRTWRT STARTS STIMER SVJPCS SWRISF
  SYSACT TFAST  TFC01  TFCADR TFCC   TFCC2  TFCCI  TFCCKR TFCCKS TFCCLN

  TFCCLO TFCCPS TFCCRE TFCDLR TFCDUM TFCFIR TFCGFR TFCGI4 TFCGLR TFCGNR
  TFCGPR TFCGR8 TFCGRC TFCGS  TFCGSP TFCGST TFCGTR TFCLAS TFCO   TFCOP
  TFCPI4 TFCPMS TFCPOS TFCPR  TFCPR8 TFCPS  TFCPST TFCPUR TFCR   TFCRET
  TFCRPR TFCSBT TFCSKB TFCSKR TFCSTC TFCSTX TFCSTY TFCWMT TFF20  TFFAJR
  TFFCLO TFFCPL TFFCPM TFFCPW TFFCRE TFFCS1 TFFCSC TFFCSP TFFCSV TFFFRB

  TFFFRC TFFGBK TFFGET TFFINT TFFJOU TFFJST TFFJWR TFFLSH TFFOP  TFFOPN
  TFFPUR TFFPUT TFFRBK TFFRDB TFFREN TFFRET TFFTBN TFFTFN TFFTIN TFFWBK
  TFFWDB TFFWEN TFIADD TFIAFR TFIAIV TFIARS TFIAS  TFICKY TFICRE TFIDEL
  TFIEIC TFIFRM TFIIIT TFIIND TFILOD TFIMKY TFIOP  TFIPOS TFIPUR TFIR
  TFIRET TFITHR TFIUNT TFIWTH TFMR   TOGGLE TS1STR TS2STR TS3STR TSACOL
```

```
TSAMVN TSAMVS TSATAB TSAVLM TSAVLN TSAVLS TSEAR  TSEND  TSLCHR TSLCOL
TSLFH  TSLIVN TSLIVS TSLMVN TSLMVS TSLNUM TSLSTR TSLSWI TSLVLM TSLVLN
TSLVLS TSSCHR TSSFH  TSSIVM TSSIVN TSSIVS TSSNUM TSSSTR TSSSWI TSTFNC
TTIMER UAREXE UARSET UARTST UC0808 UC0832 UC3232 UNWIND UPDATE VCNALC
VCNDLC VCNGUN VCNRED VCNRST VCNSET VCNWRT VEDESC VHNAME VHST   VHSTNM

VHSTNO WEOR   WRITC  XBYTGT XFER   XFERFV XFERTV XMAP   XPIABS YAFTER
YATTR  YBEGIN YBLTRC YCACHE YCCNT  YCCNTD YCDEL  YCEXIT YCFIND YCFRST
YCGDMY YCIS   YCISD  YCLAST YCLEAR YCLOCK YCMPTB YCNEXT YCPREV YCRDMY
YCREST YCSAM  YCSAMD YCWRIT YDESC  YDESCB YDESCD YDESCM YDETAL YDTMST
YDTXIN YDTXKY YDTXRC YEND   YENDDB YERMSG YEXIT  YFPXKY YFPXRC YFROM

YINXDT YINXKY YINXRC YINXTM YLABEL YLABLN YLABLS YLOGIN YMSLAB YMSTRC
YNCASE YNEW   YNOR   YNORD  YNRECS YNSIDS YNVARS YOPEN  YOPT   YORBD
YORDBI YOSDB  YRCFFP YRCFST YRCNT  YRCNTD YRCNTL YRCTRC YRCXDT YRCXFP
YRCXIN YRCXST YRCXTM YRDEL  YREXIT YRFIND YRFRST YRGDMD YRGDML YRGDMY
YRIS   YRISD  YRISL  YRLAST YRLOCK YRMXRC YRNAMD YRNEXT YRNUM  YRNUMD

YRPREV YRRDMY YRREST YRSAM  YRSAMD YRSAML YRWRIT YSDESC YSECLV YSECUR
YSTART YSTMDT YSTMTM YSTXKY YSTXRC YTHRU  YTIME  YTMMST YTMXIN YTMXKY
YUNTIL YUPLEV YUSER  YVARLB YVERS  YVNAME YVRYPE YWTH   ZAFTER ZATTR
```

Common Blocks

The following names are reserved as common blocks:

```
ACCESS CACHMS CMSOMD COMMON COMO   COMTB  DETIND ENVIRO FCBS   GLOBAL
HERROR HLPHDR HSPACE HSTCOM INDEX  ISFCOM MASKS  MONTHS OLDIND PATBLK
PCINDX PIBSAV PSBSAV RESBLK RMTBLK STRING TABLE  TFLCOM
```