Leif Svalgaard

MI Programming

408250 & 408251

http://iSeries400.org

Introduction

Why Program at the Machine-Level?

Leaving aside the precise definition of Machine-Level for a bit, a good reason is that it is just plain ol' *fun*. It certainly is different from cranking out RPG-programs. Then there are all the standard reasons that have to do with wringing more performance out of your system with the usual caveats about only optimizing what is worth optimizing. There are many APIs on the AS/400 that are just machine-level instructions in disguise, and at times bad ones at that. When you are using such an API, you are doing machine-level programming, but often without understanding precisely what is going on and often forced into a clumsy and tedious format required by the high-level language you are using. If you know how to program at the machine-level, you will find it easier to use APIs from your high-level language, and will use them with more confidence. Finally, there are things that you just can't do in any reasonable, other way without going to the machine-level.

What is the Machine-Level?

One definition is that it simply is the lowest level at which you can work as a programmer. This level constitutes an *abstract* machine, and you program to that abstraction. There are always levels below that you ordinarily don't care about, and in fact often don't want even to *know* about. The AS/400 is somewhat unusual because there are *two* machine-levels that play a role in practice. The upper one is called the MI-level and the lower one is the CISC/RISC platform**.** MI, or the *Machine Interface*, is in a very real sense what makes your machine an AS/400 rather than just a souped-up PowerPC. This book will show you how to program to the MI-level. We shall also examine the RISC platform in detail so you will understand something of what goes on "under the covers".

Above and Below the MI

The operating system that controls your machine has two parts. Over time some misleading nomenclature has been used. You have probably heard about "horizontal" and "vertical" micro-code. These names have fallen out of favor, mainly for legal reasons. Since "micro-code" is considered part of the hardware you can *own* micro-code. IBM doesn't want you to own the operating system, so requires you to *license* it instead, hence had to change the names. Today, names like OS/400 and SLIC (System Licensed Internal Code) are used. Basically, OS/400 is programmed to the MI-level and SLIC is programmed to the PowerPC RISC platform. This is often expressed by saying that OS/400 (and your applications) are *above* the MI and SLIC is *below* the MI, hence justifying talking about the machine *interface*.

Old MI and New MI

Just as the AS/400 hardware has evolved, MI has too. MI was designed to be extensible and new operations and functionality have been added over time as needed. We are at a point now where one can talk about the "old" or classic MI supporting the "old programming model", OPM, and the "new" MI supporting the ILE programming model with its emphasis on C-style programs. Today's RISC-based AS/400 only support the ILE programming model, but a special module in SLIC takes care of transforming OPM program objects into ILE modules bound into an ILE program. The module that does that has been called the "Magic" module. There is this notion that there is some magic involved in MI-programming. I don't like magic. There is a famous quote from the Science Fiction master Arthur C. Clarke that "any sufficiently advanced technology is indistinguishable from magic". One purpose of this book is to dispel some of the magic by seeking an actual understanding of what is happening.

Is MI Hard and Arcane?

It is a common misconception that machine-level programming is *hard*. There may be some truth to that at the RISC level, but that certainly is not so at the MI-level. MI is a very expressive language with powerful data structuring facilities that makes programming easy and straightforward. I can still remember my very first MI-program (some time back in 1989). I converted a 2000-line COBOL program into MI in less than a

week and it ran the first time dropping the time it took to generate a 5250 datastream from 0.337 seconds to 0.017 seconds for a speed-up factor of 20. If I can do it, so can you.

What About All Those MI-Instructions?

Analysis of several large production-type MI-programs containing thousands of instructions show that only 10 instructions comprise almost 80% of all instructions used:

Mnemonic	Freq	% of Total	Instruction Description
CPYBLA	19.89 %	19.89 %	Copy Bytes Left Adjusted
CPYNV	10.86 %	30.75 %	Copy Numeric Value
B	10.03 %	40.78 %	Branch
CMPNV	9.79 %	50.57 %	Compare Numeric value
ADDN	7.95 %	58.52 %	Add Numeric
CMPBLA	6.53 %	65.05 %	Compare Bytes Left Adjusted
CPYBLAP	4.27 %	69.32 %	Copy Bytes Left Adjusted with Pad
CALLX	3.80 %	73.12 %	Call External (program)
SUBN	3.62 %	76.74 %	Subtract Numeric
CALLI	2.67 %	79.41 %	Call Internal (subroutine)

Fundamental operations include copying characters (**CPYBLA** and **CPYBLAP**), copying numbers (**CPYNV**), branching (**B**), and comparisons (**CMPBLA** for characters and **CMPNV** for numbers). These alone make up more than 61% of all instructions coded. The moral of this exercise was to show that a lot could be accomplished with a little, so you should be able to quickly become productive. Ovid said "Add little to little and there will be a big pile", same thing here.

Small Programs or Large Programs

Some people advocate only writing very small programs in MI. Useful programs can be as short as a single instruction (with an implied return instruction). The argument is that maintaining MI programs is hard. This is actually not the case, rather, well-written MI-programs are easy to maintain (as are most *well-written* programs in any language). What *was* true, was that finding people with the skills needed was hard. That is another one of the reasons for this very book. When you have *worked* your way through the book, you will find that acquiring MI-skills was not all that hard.

I know of whole applications written solely in MI, comprising tens of thousands of lines of source code. Experience shows that these applications are not any harder to maintain than applications written in other languages. Because of the interoperability of programs on the AS/400 (one of the delivered promises of ILE) it probably would make good sense to write pieces of the application in languages best suited for that particular piece, with MI taking its place among the others on an equal footing doing what *it* does best.

Why is MI so Secret?

As we all know, IBM has not been very helpful in supplying information about MI-programming. The hoped for support, expressed in NEWS 3X/400 September 1989, that "You may decide that you would like IBM to let the S/38 and AS/400 "be all that they can be" by openly supporting MI" did not come to pass. There have been a handful of articles in the various AS/400 publications, and only recently has there been a mailing list (MI400@MIDRANGE.COM) to cater for the curious-minded AS/400 programmers. The IBM public documentation for the MI language is very sparse. The MI Functional Reference Manual, which describes each (of a subset, only) MI instruction doesn't even have one single MI example within it's many pages. The System API Reference manual has a tiny chapter, dedicated to the MI language syntax, but all of this information not really enough to be a working set of reference materials for programming in MI.

Getting Your Own MI-Compiler

MI-Compilers

If you want to program in MI, you of course will need a compiler. While you could purchase an MI-compiler for the S/38 from IBM (the "PRPQ"), IBM claimed than ***no*** MI-compiler for the AS/400 was available. Several people discovered early on that contrary to this claim, every AS/400 was, in fact, shipped with an MI-compiler built in. What is missing is a convenient way to invoke the compiler, i.e. there is no **CRTMIPGM** command. Some people would you sell a simple front-end program as "An MI-compiler" for several thousand dollars. Maybe to stop that practice (or for reasons unknown), IBM finally documented an API named QPRCRTPG (Create Program) to invoke the compiler. You can find this information (at least until IBM pulls it again) at this web-site:

"http://as400bks.rochester.ibm.com/cgi-bin/bookmgr/bookmgr.cmd/BOOKS/QB3AVC00/7.0".

The method described at the above URL is needlessly complex and even wrong if you follow the instructions to the letter (hint: one of the CL-programs blows up because the source presented to it exceeds the 2000-character limit imposed on a variable in CL) and as such we shall not duplicate that material here. Instead we shall write an RPG-program that will invoke the API. If you do not have an RPG compiler, you can download a ready-made savefile with the resulting compiler front-end from our website at this URL: http://iSeries400.org/micomp.exe. You will learn more from trying to build one yourself following the steps given in this book.

The Create Program (**QPRCRTPG**) API converts the symbolic representation (read: simple source text) of a machine interface (MI) program into an OPM program object. This symbolic representation is known as the intermediate representation of a program. The **QPRCRTPG** API creates a program object that resides in the *USER domain and runs in the *USER state. In later chapters we shall examine the domain/state concepts in detail.

Create Program (QPRCRTPG) API

The **QPRCRTPG** API is documented in the books for V3R2 and in the System API manual for later releases. Here is the an on-line reference http://publib.boulder.ibm.com/pubs/html/as400/online/v3r2eng.htm. Below I excerpt here the relevant information for your convenience (who knows for how long the above link is still good?). Much of the detailed description (shown in a smaller font) may not make much sense to your at this stage, so you should just consider it to be reference material, skim it, and continue with confidence.

What makes the API a little tricky to use is that you do not pass it the name of a member in a source file as you would ordinarily do, but instead, you pass the API a large character array containing the source. For this reason, a *front-end* is usually required to read the source records into an array. Here is first the parameter list for the API:

1	Program source statements	In	Char(*)
2	Length of program source statements	In	Binary(4)
3	Qualified program name	In	Char(20)
4	Program text	In	Char(50)
5	Qualified source file name	In	Char(20)
6	Source file member information	In	Char(10)
7	Source file last changed date and time	In	Char(13)
8	Qualified printer file name	In	Char(20)
9	Starting page number	In	Binary(4)
10	Public authority	In	Char(10)
11	Option template	In	Char(*)
12	Number of option template entries	In	Binary(4)
13	Error code (optional)	I/O	Char(*)

The MI-Compiler Front-End

We must first decide which language to use for our compiler front-end. Every AS/400 has a CL-compiler, but the limitation of 2000 characters for the size of the variable to hold the MI source string is severe. Most AS/400 shops have an RPG-compiler, but even RPG has limitations. Other languages are not used broadly enough to be viable for my purpose, so I decided to write the front-end in MI! Of course, I still have to compile the front-end, but since the front-end code is small (about 200 lines) I can embed the code in an RPG array and have a simple RPG-program call the **QPRCRTPG** API to generate the MI front-end program.

```
/*============================================================
 * This program creates MI compiler front-end CRTMIPGM in *CURLIB=
 * Source statements for the MI compiler are found in array MI.  =
 *============================================================
E                    MI      1 208 80
I            DS
I                                    B   1   40#SRCLN
I I            'CRTMIPGM  *CURLIB'       5  24 #PGMLB
I                                      25  74 #TEXT
I I            '*NONE'                 75  94 #SRCFL
I                                      95 104 #MBR
I                                     105 117 #CHGDT
I                                     105 105 #CENT
I                                     106 107 #YY
I                                     108 111 #MMDD
I                                     112 117 #HMS
I                                     118 137 #PRTFL
I                                    B 138 1410#STRPG
I                                      142 151 #AUT
I                                      152 327 #OP
I                                    B 328 3310#NOOPT
C                      CALL 'QPRCRTPG'
C                      PARM           MI
C                      PARM 16640     #SRCLN
C                      PARM           #PGMLB
C                      PARM 'MI Comp' #TEXT
C                      PARM           #SRCFL
C                      PARM           #MBR
C                      PARM           #CHGDT
C                      PARM ' '       #PRTFL
C                      PARM 0         #STRPG
C                      PARM '*USE'    #AUT
C                      PARM '*REPLACE'#OP
C                      PARM 1         #NOOPT
C                      MOVE *ON       *INLR
```

You can easily recognize the parameters to the **QPRCRTPG** API. All we have to do now is to populate the array MI with the appropriate code (as shown below), compile the RPG program and run it. The result is the **CRTMIPGM** program object. Now, you should try this right away.

Below are the contents of the MI array. The curious '*/' in the first line has a matching '/*' at the beginning of the RPG-program. Comments in MI are free-form text (can be spread over several lines) starting with '/*' and ending with '*/'. That makes the entire code part of the RPG-program an MI-comment. In fact, the program is at the same time a valid RPG-program *and* a valid MI-program. You can give it to either compiler and it will compile and run. Why would I do a thing like this? Maintenance. I can enhance the front-end and make it a real pre-compiler supported added functionality. I have, in fact, done that already with the **%INCLUDE** facility. I can then test the result as I would test any other MI-program.

Don't worry yet about how the code in the array works. We shall get to all that in due time. For now, just look at it as an example of a non-trivial MI-program. This is not a toy program

Here is then, finally, the RPG-array:

```
**                                                          */
DCL SPCPTR .MBR PARM;
DCL SPCPTR .FIL PARM;
DCL SPCPTR .DET PARM;
DCL OL *ENTRY (.MBR, .FIL, .DET) PARM EXT MIN(1);
DCL DD MBR CHAR(10) BAS(.MBR);
DCL DD FIL CHAR(10) BAS(.FIL);
DCL DD DET CHAR(10) BAS(.DET);
```

```
DCL SPC PCO  BASPCO;
    DCL SPCPTR .PCO DIR;

DCL SPC SEPT BAS(.PCO);
    DCL SPCPTR .SEPT(2000) DIR;

DCL SPCPTR .UFCB INIT(UFCB);
DCL DD UFCB CHAR(214) BDRY(16);
    DCL SPCPTR .ODP           DEF(UFCB) POS(  1);
    DCL SPCPTR .INBUF         DEF(UFCB) POS( 17);
    DCL SPCPTR .OUTBUF        DEF(UFCB) POS( 33);
    DCL SPCPTR .OPEN-FEEDBACK DEF(UFCB) POS( 49);
    DCL SPCPTR .IO-FEEDBACK   DEF(UFCB) POS( 65);
    DCL SPCPTR .NEXT-UFCB     DEF(UFCB) POS( 81);

    DCL DD *         CHAR(32) DEF(UFCB) POS( 97);
    DCL DD FILE      CHAR(10) DEF(UFCB) POS(129) INIT("QMISRC");
    DCL DD LIB-ID    BIN ( 2) DEF(UFCB) POS(139) INIT(-75);
    DCL DD LIBRARY   CHAR(10) DEF(UFCB) POS(141) INIT("*LIBL");
    DCL DD MBR-ID    BIN ( 2) DEF(UFCB) POS(151) INIT( 73);
    DCL DD MEMBER    CHAR(10) DEF(UFCB) POS(153);

    DCL DD ODP-DEVICE-NAME    CHAR(10) DEF(UFCB) POS(163);
    DCL DD ODP-DEVICE-INDEX   BIN ( 2) DEF(UFCB) POS(173);

    DCL DD FLAGS-PERM-80      CHAR( 1) DEF(UFCB) POS(175) INIT(X'80');
    DCL DD FLAGS-GET-20       CHAR( 1) DEF(UFCB) POS(176) INIT(X'20');
    DCL DD REL-VERSION        CHAR( 4) DEF(UFCB) POS(177) INIT("0100");
    DCL DD INVOC-MARK-COUNT   BIN ( 4) DEF(UFCB) POS(181);
    DCL DD MORE-FLAGS         CHAR( 1) DEF(UFCB) POS(185) INIT(X'00');
    DCL DD *                  CHAR(23) DEF(UFCB) POS(186);

    DCL DD RECORD-PARAM       BIN ( 2) DEF(UFCB) POS(209) INIT(1);
    DCL DD RECORD-LENGTH      BIN ( 2) DEF(UFCB) POS(211) INIT(92);

    DCL DD NO-MORE-PARAMS     BIN ( 2) DEF(UFCB) POS(213) INIT(32767);

DCL SPC ODP BAS(.ODP);
    DCL DD *          CHAR(16) DIR;
    DCL DD DEV-OFFSET BIN ( 4) DIR;

DCL SPCPTR .DMDEV;
DCL SPC     DMDEV BAS(.DMDEV);
    DCL DD MAX-DEVICE      BIN ( 2) DIR;
    DCL DD NBR-DEVICES     BIN ( 2) DIR;
    DCL DD DEVICE-NAME     CHAR(10) DIR;
    DCL DD WORKAREA-OFFSET BIN ( 4) DIR;
    DCL DD WORKAREA-LENGTH BIN ( 4) DIR;
    DCL DD LUD-PTR-INDEX   BIN ( 2) DIR;
    DCL DD DM-GET          BIN ( 2) DIR;

DCL SPCPTR .GETOPT INIT(GETOPT);
DCL DD      GETOPT CHAR(4);
    DCL DD  GET-OPTION-BYTE CHAR(1) DEF(GETOPT) POS(1) INIT(X'03');
    DCL DD  GET-SHARE-BYTE  CHAR(1) DEF(GETOPT) POS(2) INIT(X'00');
    DCL DD  GET-DATA-BYTE   CHAR(1) DEF(GETOPT) POS(3) INIT(X'00');
    DCL DD  GET-DEVICE-BYTE CHAR(1) DEF(GETOPT) POS(4) INIT(X'01');

DCL SPCPTR .NULL;
DCL OL GET  (.UFCB, .GETOPT, .NULL);
DCL OL OPEN (.UFCB);
DCL OL CLOSE(.UFCB);

DCL SPC INBUF BAS(.INBUF);
    DCL DD INBUF-DATE CHAR(12) DEF(INBUF) POS( 1);
    DCL DD INBUF-LINE CHAR(80) DEF(INBUF) POS(13);
        DCL DD INBUF-KEYWORD CHAR( 9) DEF(INBUF-LINE) POS( 1);
        DCL DD INBUF-NEWMBR  CHAR(10) DEF(INBUF-LINE) POS(10);

DCL SPCPTR .SOURCE;
DCL DD LINE(10000) CHAR(80) AUTO;
DCL DD LINE-NBR BIN(4);
DCL DD READ-NBR BIN(4);
DCL DD SAVE-NBR BIN(4);
DCL DD SKIP-NBR BIN(4);
DCL DD INCL-NBR BIN(2);

DCL SPCPTR .SIZE INIT(SIZE);
DCL DD      SIZE BIN(4);
```

```
DCL SPCPTR .PGM INIT(PGM);
DCL DD      PGM CHAR(20);
    DCL DD PGM-NAME  CHAR(10) DEF(PGM) POS( 1);
    DCL DD PGM-LIB   CHAR(10) DEF(PGM) POS(11) INIT("*CURLIB");

DCL SPCPTR .PGM-TEXT INIT(PGM-TEXT);
DCL DD      PGM-TEXT CHAR(50) INIT(" ");

DCL SPCPTR .PGM-SRCF INIT(PGM-SRCF);
DCL DD      PGM-SRCF CHAR(20) INIT("*NONE");

DCL SPCPTR .PGM-SRCM INIT(PGM-SRCM);
DCL DD      PGM-SRCM CHAR(10) INIT(" ");

DCL SPCPTR .PGM-SRCD INIT(PGM-SRCD);
DCL DD      PGM-SRCD CHAR(13) INIT(" ");

DCL SPCPTR .PRTF-NAME INIT(PRTF-NAME);
DCL DD      PRTF-NAME CHAR(20);
    DCL DD PRTF-FILE CHAR(10) DEF(PRTF-NAME) POS( 1) INIT("QSYSPRT ");
    DCL DD PRTF-LIB  CHAR(10) DEF(PRTF-NAME) POS(11) INIT("*LIBL   ");

DCL SPCPTR .PRT-STRPAG INIT(PRT-STRPAG);
DCL DD      PRT-STRPAG BIN(4) INIT(1);

DCL SPCPTR .PGM-PUBAUT INIT(PGM-PUBAUT);
DCL DD      PGM-PUBAUT CHAR(10) INIT("*ALL");

DCL SPCPTR .PGM-OPTS INIT(PGM-OPTS);
DCL DD      PGM-OPTS(16) CHAR(11) INIT("*REPLACE   ", "*NOADPAUT  ",
                         "*NOCLRPSSA ", "*NOCLRPASA ", "*SUBSCR    ",
                         "*LIST      ", "*ATR       ", "*XREF      ");

DCL SPCPTR .NBR-OPTS INIT(NBR-OPTS);
DCL DD      NBR-OPTS BIN(4);

DCL OL QPRCRTPG (.SOURCE, .SIZE, .PGM,   .PGM-TEXT,  .PGM-SRCF,
                 .PGM-SRCM,   .PGM-SRCD, .PRTF-NAME, .PRT-STRPAG,
                 .PGM-PUBAUT, .PGM-OPTS, .NBR-OPTS) ARG;

DCL SYSPTR .QPRCRTPG INIT("QPRCRTPG", CTX("QSYS"), TYPE(PGM));

DCL DD NBR-PARMS BIN(2);
DCL EXCM * EXCID(H'5001') BP(EOF) IMD;

DCL DD START CHAR(80);
    DCL DD *         CHAR(12) DEF(START) POS( 1) INIT("/* INCLUDE: ");
    DCL DD NEWMBR    CHAR(10) DEF(START) POS(13);
    DCL DD *         CHAR(58) DEF(START) POS(23) INIT(" */");

DCL DD STOP  CHAR(80);
    DCL DD * CHAR(80) DEF(STOP) POS(1) INIT("/* END INCLUDE */");

/*************************************************************/

ENTRY * (*ENTRY) EXT;
    CPYNV        LINE-NBR, 1;
    CPYNV        INCL-NBR, 0;
    CPYNV        SKIP-NBR, 0;

    CPYBWP      .NULL, *;
    CPYNV        NBR-OPTS,  6; /* YES: *LIST; NO: *ATR, *XREF */
    STPLLEN      NBR-PARMS;
    CMPNV(B)     NBR-PARMS, 3/NEQ(PREPARE-FILE);
    CMPBLA(B)    DET, <10|*DETAIL   >/EQ(YES-DETAIL);
    CMPBLA(B)    DET, <10|*NOLIST   >/EQ(NO-LIST);
    B            PREPARE-FILE;
YES-DETAIL:      CPYNV(B) NBR-OPTS, 8/NNAN(PREPARE-FILE);
NO-LIST:         CPYNV(B) NBR-OPTS, 5/NNAN(PREPARE-FILE);

PREPARE-FILE:
    CPYBLAP      FILE, "QMISRC", " ";
    CMPNV(B)     NBR-PARMS, 1 /EQ(SET-MEMBER);
    CPYBLA       FILE, FIL;
SET-MEMBER:
    CPYBLA       MEMBER, MBR;
    CPYBLA       PGM-NAME, MBR;
OPEN-FILE:
    CPYNV        READ-NBR, 0;
    CALLX       .SEPT(12), OPEN, *;
    ADDSPP      .DMDEV, .ODP, DEV-OFFSET;
```

```
NEXT-SOURCE-RECORD:
    CALLX       .SEPT(DM-GET), GET, *;
    ADDN(S)      READ-NBR, 1;
    SUBN(SB)     SKIP-NBR, 1/NNEG(NEXT-SOURCE-RECORD);
    CMPBLA(B)    INBUF-KEYWORD, "%INCLUDE "/EQ(INCLUDE-MEMBER);
    CPYBLA       LINE(LINE-NBR), INBUF-LINE;
    ADDN(S)      LINE-NBR, 1;
    B            NEXT-SOURCE-RECORD;

EOF:
    CALLX       .SEPT(11), CLOSE, *;
    CMPNV(B)     INCL-NBR, 0/HI(END-INCLUDE);
    CPYBLAP      LINE(LINE-NBR), <23|/*'/*'/*"/*"*/; PEND;;;>, " ";
    MULT         SIZE, LINE-NBR, 80;
    SETSPP      .SOURCE, LINE;
    CALLX       .QPRCRTPG, QPRCRTPG, *;
    RTX          *;

ERROR:
    RTX          *;

INCLUDE-MEMBER:
    ADDN(S)      INCL-NBR, 1;
    CPYBLA       NEWMBR, INBUF-NEWMBR;
    CALLX       .SEPT(11), CLOSE, *;
    CPYBLA       MEMBER, NEWMBR;
    CPYBLA       LINE(LINE-NBR), START;
    ADDN(S)      LINE-NBR, 1;
    CPYNV(B)     SAVE-NBR, READ-NBR/NNAN(OPEN-FILE);

END-INCLUDE:
    CPYBLA       LINE(LINE-NBR), STOP;
    ADDN(S)      LINE-NBR, 1;
    SUBN(S)      INCL-NBR, 1;
    CPYBLA       MEMBER, MBR;
    CPYNV(B)     SKIP-NBR, SAVE-NBR/NNAN(OPEN-FILE);

PEND;
```

The curious statement "`CPYBLAP LINE(LINE-NBR),<23|/*'/*'/*"/*"*/; PEND;;;>," ";`" catches strings and comments that are not closed. Without this statement, the translator is prone to crashing ("dumping") if end-of-file is encountered and a string or comment is still open.

The CRTMIPGM Command

It is most convenient to use **`CRTMIPGM`** from a *command*. Create the source member `CRTMIPGM` in the source file QCMDSRC:

```
CMD     PROMPT('Create MI program')
PARM    KWD(PGM) TYPE(*CHAR) LEN(10) MIN(1)  +
        PROMPT('Program Source Member')
PARM    KWD(FILE) TYPE(*CHAR) LEN(10)        +
        DFT(QMISRC)                          +
        PROMPT('  in Source File')
PARM    KWD(LIST) TYPE(*CHAR) LEN(10)        +
        DFT(*LIST)                           +
        PROMPT('  LIST option')
```

Then use option 14 with prompting to create the command:

```
                          Create Command (CRTCMD)
Type choices, press Enter.
Command  . . . . . . . . . . . . > CRTMIPGM     Name
  Library  . . . . . . . . . . . >   LSVALGAARD  Name, *CURLIB
Program to process command . . . > CRTMIPGM     Name, *REXX
  Library  . . . . . . . . . . . >   *LIBL       Name, *LIBL, *CURLIB
Source file  . . . . . . . . . . > QCMDSRC      Name
  Library  . . . . . . . . . . . >   LSVALGAARD  Name, *LIBL, *CURLIB
Source member  . . . . . . . . . > CRTMIPGM     Name, *CMD
Threadsafe . . . . . . . . . . .   *NO          *YES, *NO, *COND
```

You can the either run it from the command line:

```
===> CRTMIPGM PGM(yourpgm) FILE(QMISRC) LIST(*LIST)
```

or (if the defaults are to your liking):

```
===> CRTMIPGM yourpgm
```

You can also prompt the command:

```
                        Create MI program (CRTMIPGM)
Type choices, press Enter.
Program Source Member  . . . . . yourpgm      Character value
  in Source File . . . . . .   QMISRC       Character value
  LIST option  . . . . . .     *LIST        Character value
```

Note, that the source file is by default **QMISRC** on the library list. The following *options* are selected automatically in the front-end:

```
   *REPLACE    *NOADPAUT   *NOCLRPSSA *NOCLRPASA *SUBSCR     *LIST       *ATR        *XREF
```

The *LIST option on the command cuts the last two options from the above set, meaning, in effect *LIST only. *NOLIST cuts the last three, and *DETAIL cuts none.

At this point I suggest that you copy the RPG-source of **CRTMIPGM** to a member with a different name in **QMISRC** (create the 92-character source file first, if needed), e.g. **QMISRC/TEST**, then try to compile **TEST**:

```
===> CRTMIPGM test
```

A listing can be found in the spoolfile **QSYSPRT**. Before continuing, make sure that all this works and that you understand the process (even if the code is still voodoo or 'magic').

Hello World

It is customary in programming language books to start your first programming lesson by showing how to write a program that displays the text "Hello World" in that particular language. Some languages ask you to marvel over the fact that this program can be written as a single statement, in which case, of course, the exercise is vacuous, because you haven't learnt anything by writing **display "Hello World"**. Here is what an MI-version might look like:

```
CPYBLAP MSG-TEXT, "Hello World", " ";
CALLI SHOW-MESSAGE, *, .SHOW-MESSAGE;
RTX *;
%INCLUDE SHOWMSG
```

The first line copies (**CPY**) the bytes (**B**) left-aligned (**LA**) from the 11-bytes long literal "Hello World" to the variable **MSG-TEXT**, then pads (**P**) the result with blanks (" ") until the end of the receiver. Note, that instructions are generally executing from right-to-left, so that "CPYBLA A, B" copies B to A**.** This is the opposite of RPG's and COBOL's "MOVE A to B" that work left-to-right. This may take some time to get used to, but is similar to mathematical expressions, e.g. A = B, means assign B to A**.** Or A = sqrt(B), which means: take B, extract its square-root, then assign the result to A**.**

The second line calls (**CALL**) the internal (**I**) subroutine **SHOW-MESSAGE**. Which presumably is going to show the message (possibly on the display device). Finally, the program returns (**RT**) to its external caller (**X**). Commas separate operands and a semi-colon (;) terminates each instruction. Instructions are conventionally written one to a line, but they don't need to be. MI is completely free-form. Except for literal strings, everything must be in UPPER CASE. This is one of the restrictions we will relax in a later chapter when we introduce more pre-processing facilities in our MI front-end.

We haven't said anything about the fourth line (%INCLUDE SHOWMSG). We are hiding something messy here. The display the text, we are sending a message to a job using the **QMHSNDM** API. This API is *not* at the MI-level, but is a rather high-level system function. There is nothing that says that we have to stay at the machine-level all the time. In fact, if there is a higher-level function that does the job using a reasonable amount of resources, by all means use it. The problem with **QMHSNDM** is that it takes *ten* parameters specifying all kinds of complicated details that we really don't want to know about. There are two main approaches to this problem, the first is to make a separate, external program that you simply call with the

text to show, the second is to have an internal subroutine that you just call, after having loaded a common variable (**MSG-TEXT**) with the text to display. The declaration, variables, and code for the subroutine are held in a separate source member (QMISRC/SHOWMSG) and are simply included into your MI-program. This is one of the main reasons why it was handy to provide an include facility in our pre-processing MI front-end.

One thing of note here, is that as stated before, MI is completely free form, notice how we included the SHOWMSG source file at the *bottom* of the MIHELLO source file. In languages such as RPG, all variable definitions must be before the first calculation line so that, in essence, they are defined before they are first referenced. In MI there is no such restriction on the placement of variable definitions.

Below is the code to store in the QMISRC/SHOWMSG source member; we'll discuss how it works in a later chapter, but I guess that you can see that it basically defines and initializes the parameters to **QMHSNDM** and then calls the API with those values:

```
/* SHOW A MESSAGE */

DCL SPCPTR .MSG-ID   INIT(MSG-ID);
DCL DD      MSG-ID   CHAR (7) INIT(" ");

DCL SPCPTR .MSG-FILE INIT(MSG-FILE);
DCL DD      MSG-FILE CHAR(20) INIT(" ");

DCL SPCPTR .MSG-TEXT INIT(MSG-TEXT);
DCL DD      MSG-TEXT CHAR(70);

DCL SPCPTR .MSG-SIZE INIT(MSG-SIZE);
DCL DD      MSG-SIZE BIN( 4)  INIT(70);

DCL SPCPTR .MSG-TYPE INIT(MSG-TYPE);
DCL DD      MSG-TYPE CHAR(10) INIT("*INFO");

DCL SPCPTR .MSG-QS   INIT(MSG-QS);
DCL DD      MSG-QS   CHAR(20) INIT("*REQUESTER");

DCL SPCPTR .MSG-QSN  INIT(MSG-QSN);
DCL DD      MSG-QSN  BIN( 4)  INIT(1);

DCL SPCPTR .REPLY-Q  INIT(REPLY-Q);
DCL DD      REPLY-Q  CHAR(20) INIT(" ");

DCL SPCPTR .MSG-KEY  INIT(MSG-KEY);
DCL DD      MSG-KEY  CHAR( 4);

DCL SPCPTR .ERR-CODE INIT(ERR-CODE);
DCL DD      ERR-CODE BIN( 4) INIT(0);

DCL OL QMHSNDM (.MSG-ID,   .MSG-FILE, .MSG-TEXT, .MSG-SIZE,
                .MSG-TYPE, .MSG-QS,   .MSG-QSN,  .REPLY-Q,
                .MSG-KEY,  .ERR-CODE)  ARG;

DCL SYSPTR .SEPT(6440) BAS(SEPT-POINTER);
DCL SPC PROCESS-COMMUNICATION-OBJECT BASPCO;
    DCL SPCPTR SEPT-POINTER DIR;

DCL INSPTR .SHOW-MESSAGE;
ENTRY       SHOW-MESSAGE INT;
        CALLX     .SEPT(4268), QMHSNDM, *; /* SEND MSG TO MSGQ */
        B         .SHOW-MESSAGE;
```

Enter, compile, and run the MIHELLO program. You should see a result similar to this:

```
                          Display Messages
  Queue . . . . . :   LSVALGAARD              Program . . . . :   *DSPMSG
    Library . . . :     QUSRSYS                 Library . . . :
  Severity  . . . :   00                      Delivery  . . . :   *BREAK
  Type reply (if required), press Enter.
    From  . . . :   LSVALGAARD     07/24/00   16:55:31
    Hello World
```

MI Functional Reference Manual

The MI Functional Reference Manual is available online (http://www.as400.ibm.com/tstudio/tech_ref/mi/). You can also order it from IBM. It is a very large document (1440 pages), but it is a must-have. The price is about $100.00 depending on what version you want. There is very little difference between versions, so your best deal is to take the latest one you can get. Here is how to order it:

Go to http://www1.ibmlink.ibm.com/
Select IBMLink for the United States.
Select the PubsCenter link.
Choose United States as Country
Select Search for Publications
On the Search screen, enter "Machine Interface"
Click the Go button.
You should get a list of the publications you are looking for.
.

Data Types and Your First *Real* MI-Program

Data Types

When you think about machine-level programming for a conventional computer, you may think in terms of assembly language with registers, addresses, and explicit type manipulation (e.g. different instructions for adding binary halfwords and floating point numbers). The AS/400 is radically different at the MI-level, although not at the RISC-level. At the MI-level, there are no accessible registers (some other - older - computer systems didn't have registers either), no addresses (here the AS/400 is unique, although some computers worked with 'descriptors'), and many instructions are polymorphic (know what to do based on the type of their operands). What are being manipulated in MI-programs are *primitive data-items* and *objects*. Primitive data-items can be referred to directly as operands in instructions. Objects are represented by *pointers*. Primitive data-items can also be referenced by pointers to them. We are not trying to be complete this early in the book and some of the finer details will be introduced further on.

Character Data Type

In MI, all operands must be *declared* with a declare statement **DCL**. Primitive data-items are declared using the Data Definition modifier DD: **DCL DD**. The most common data type is the character string. Here is how to declare the 30-character string **MY-TEXT**:

```
DCL DD MY-TEXT CHAR(30);
```

We can also give it an initial value:

```
DCL DD MY-TEXT CHAR(30) INIT("Hello");
```

A question arises here: the data-item is 30 characters long, will the `INIT` clause only initialize the first five characters (the length of "Hello") or will something else happen? This is very easy to test. Let's modify our **HELLO** program as follows:

```
DCL DD MY-TEST1 CHAR(30);
DCL DD MY-TEST2 CHAR(30) INIT("Hello");

    CPYBLA      MSG-TEXT, MY-TEST1;
    CALLI       SHOW-MESSAGE, *, .SHOW-MESSAGE;

    CPYBLA      MSG-TEXT, MY-TEST2;
    CALLI       SHOW-MESSAGE, *, .SHOW-MESSAGE;

    RTX         *;

%INCLUDE SHOWMSG
```

Note, that although MI is completely free-form, there is a benefit to sensible indentation and use of white space. Compile and run it:

```
                              Display Messages
                                                        System:   AS400
Queue . . . . . :   LSVALGAARD              Program . . . . :   *DSPMSG
  Library . . . :     QUSRSYS                 Library . . . :
Severity  . . . :   00                      Delivery  . . . :   *BREAK

Type reply (if required), press Enter.
  From  . . . :   LSVALGAARD     07/25/00   15:18:22

  From  . . . :   LSVALGAARD     07/25/00   15:18:25
  Hello                         .
```

As you can see, the `INIT` clause initializes the remainder of the data-item to spaces (blanks). We can learn more from this simple test. **MSG-TEXT** is longer than 30 characters (in fact, it is 70 characters long), so the **CPYBLA** instruction (*without* the **P** for padding) copies as many characters as given by the length of the source (second) operand to the destination and leaves the rest of the destination unchanged. When you are in doubt as to what a particular instruction or clause does, don't use it in a program you are writing hoping things will work out. Test it first by modifying the **HELLO** program just as we did above.

When you use quotes in literal strings, you can use either single quotes (‘) or double quotes (“), although they must match as a pair, so “Hello’ is not valid, but “a single ‘ quote” and “a double “” quote” are fine, the latter having the same value as ‘a double “” quote’.

Numeric Data Types

MI supports the following numeric data types (each shown as an example):

```
DCL DD MY-HALFWORD    BIN(2)   INIT(10000);       /* 2 byte halfword (16 bits)      */
DCL DD MY-FULLWORD    BIN(4)   INIT(1000000);     /* 4 byte fullword (32 bits)      */
DCL DD MY-SHORT-FLOAT FLT(4)   INIT(F‘3.1416’);   /* 4 byte floating point          */
DCL DD MY-LONG-FLOAT  FLT(8)   INIT(E‘-1.5E+05’); /* 8 byte floating point          */
DCL DD MY-ZONED       ZND(8,2) INIT(Z‘-123.45’);  /* 8 digit zoned with 2 decimals  */
DCL DD MY-PACKED      PKD(9,2) INIT(P‘+123.45’);  /* 9 digit packed with 2 decimals */
```

As MI is “old” technology, there is no `BIN(8)` data type denoting a doubleword integer, although the 64-bit RISC processors naturally support these in hardware. Binary numbers can be unsigned. You declare that by using the `UNSGND` clause, as in:

```
DCL DD MY-UNSIGNED-HALFWORD BIN(2) UNSGND INIT(50000); /* 2 byte unsigned (16 bits) */
```

Note, that data-item names can be long. Note also, the peculiar way you have to specify initial values of floating-point, zoned, and packed data-items. This is related to the way you specify constants in instructions. The compiler needs to know what type the constant has. For packed and zoned data-items the first number, e.g. in `PKD(15,5)` is the *total* number of digits including the decimals and the second number after the comma, the 5, is how many of those digits are to be used for the digits after the decimal point . So in the example, the largest value our PKD(15,5) data-item can represent is ‘9999999999.99999’ This should be of no surprise to RPG and CL-programmers.

To try out some of this, modify our (ever-useful) **`HELLO`** program to read:

```
DCL DD MY-HALFWORD BIN(2) INIT(1000);
DCL DD MY-FULLWORD BIN(4) INIT(1000000);
DCL DD MY-ZONED    ZND(10,2); /* nnnnnnnn.dd */

    ADDN         MY-ZONED, MY-HALFWORD, MY-FULLWORD;
    CPYBLAP      MSG-TEXT, MY-ZONED, " ";
    CALLI        SHOW-MESSAGE, *, .SHOW-MESSAGE;

    RTX          *;

%INCLUDE SHOWMSG
```

I used the **`ADDN`** (Add Numeric) instruction to compute **`MY-ZONED`** = **`MY-HALFWORD`** + **`MY-FULLWORD`**. The fact that the three operands are of different types is no problem at all because the **`ADDN`** instruction is polymorphic (knows how to handle different types). To see the result, we now copy the result to **`MSG-TEXT`** and send the message as before. Because a zoned number has one digit per character, copying the result as a character string with **`CPYBLAP`** will work just fine. Here is what we get:

```
From  . . . :   LSVALGAARD     07/25/00   20:44:00
0100100000
```

Which after positioning the decimal point in the correct place, is 100100.00, just as we would expect.

Data Names

Although reserved words (like `DCL`, `INIT`, and instruction mnemonics) have to be UPPER case, you have more freedom with data names. A name can be up to 48 characters long and consist of any sequence of most of the printable characters, *except* the ones in the following set:

blank / , ; ( ) : < + ‘ “ %

In addition, the *first* character of a name *cannot* be one of

- 0 1 2 3 4 5 6 7 8 9

Here are some valid names:

```
ThisIsAVeryLongNameUsingMixedCaseWords
A-COBOL-STYLE-NAME
#RGPNM
CUSTOMER_NAME_12
.PROGRAM-OBJECT
```

Names that begin with a period (`.`) are not inserted into the symbol table and cannot be referenced by the AS/400 debuggers. I personally use such names to signify *pointers* because of their readability (I never want to follow pointer chains with the debugger anyway). Names should neither be too long nor too short and cryptic. A good name is one you can read aloud over the telephone. Nothing new here.

Comments

Comments can occur anywhere except in literal values and inside other comments, so cannot be nested. As we have seen, comments are bracketed by the special delimiters "/*" and "*/", just like in CL-programs. A comment can span several lines. Beware of "run-away" comments where the ending delimiter "*/" is missing. Comments are treated as blanks, so they are significant in delimiting names and tokens.

Hexadecimal Constants

Character strings, binary integers, and floating-point values can be specified in hexadecimal notation, albeit in a different way for each:

```
DCL DD MY-TEXT        CHAR(7)  INIT(X'C1C2C3C4C500');      /* ABCDEnu77 */
DCL DD MY-BINARY      BIN(2)   INIT(H'07D0');              /* 2000      */
DCL DD MY-SHORT-FLOAT FLT(4)   INIT(XF'BFC00000');         /* -1.5      */
DCL DD MY-LONG-FLOAT  FLT(8)   INIT(XE'BFF8000000000000'); /* -1.5      */
```

Binary numbers are right-justified with zero-fill on the left, so `H'01'` is the same as `H'0001'`, which is, of course, just the `BIN`(2) representation of the number 1. As before, whenever in doubt about what something will do (or won't do) or you just want to explore something, modify the **HELLO** program:

```
DCL DD MY-HALFWORD1   BIN(2)  INIT(H'01'); /* is this H'0100' or H'0001' ? */
DCL DD MY-HALFWORD2   BIN(2)  INIT(H'0001');
DCL DD MY-TEXT       CHAR(2);

    CPYBLA      MY-TEXT, MY-HALFWORD1;     /* make copy to character type */
    CVTHC       MSG-TEXT(1:4), MY-TEXT;    /* convert to hexadecimal form */
    CALLI       SHOW-MESSAGE, *, .SHOW-MESSAGE;

    CPYBLA      MY-TEXT, MY-HALFWORD2;
    CVTHC       MSG-TEXT(1:4), MY-TEXT;
    CALLI       SHOW-MESSAGE, *, .SHOW-MESSAGE;
    . . .
```

The three periods simply mean that the remaining boilerplate is understood, viz.:

```
    RTX         *;

%INCLUDE SHOWMSG
```

Convert Hexadecimal to Character

Notice the new instruction: **CVTHC** `MSG-TEXT(1:4), MY-TEXT`. If we had just copied the binary value to **MSG-TEXT** with `CPYBLA MSG-TEXT, MY-HALFWORD1` the message would contain non-printable characters and we wouldn't be able to tell the result. What we need is an instruction that converts the binary value into a string of hexadecimal digits that we can immediately understand. This is precisely what the **CVTHC**– instruction does. Its name (Convert Hexadecimal to Character) is somewhat misleading, as it really converts an arbitrary character string into its hexadecimal representation (i.e. goes the other way, "ABC" -> X'C1C2C3'). The instruction does require a character data-item and not a numeric data-item; that is why we had to first copy the numeric item to a character item. If you run the program, you'll see that the two values are the same. Yet another example of how you can easily try things out (promise: this is the last time I'll harp on that).

Another thing that the CVTHC-instruction insists on, is that the receiver must be exactly twice as long as the source, so we need a character item 4 bytes long to hold the hexadecimal representation of a 2-byte value. However, this isn't that strange if you think about it. If you look at the example where the three bytes for "ABC" expands to 6 bytes for X'C1C2C3' you can see the reason why.. You can accomplish that by a substring notation: `MSG-TEXT(1:4)`, meaning the substring starting in position 1 and extending 4 characters to the right.

Compile Errors

Now is the time to deal with the problem of errors. You *will* make errors (I make many each and every day), so here goes. My first version of showing the hexadecimal value of a binary number looked like this:

```
DCL DD MY-HALFWORD  BIN(2) INIT(H'01');
DCL DD MY-TEXT     CHAR(2);

    CVTHC       MSG-TEXT(1:4), MY-HALFWORD;
    CALLI       SHOW-MESSAGE, *, .SHOW-MESSAGE;
    . . .
```

When compiling it, I got this message:

```
Processing command failure, refer to job log for details.
```

The job log contained:

```
     Intermediate representation of program (IRP) contains 1 errors. Probable
       compiler error.
```

No, it's not a misprint or typo. The MI-translator (to use the proper term) was designed to process output from the HLL compilers, so if there were errors in the MI-statements, there probably *was* an error in the HLL compiler.

The MI-translator listing is spooled to **QSYSPTR**:

```
 5769SS1 V4R4M0 990521                                  Generated Output
  SEQ   INST Offset    Generated Code     *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 .
 00001                                     DCL DD MY-HALFWORD BIN(2) INIT(H'0001')
 00002                                     DCL DD MY-TEXT CHAR(2)
 00003  0001 000004  1086 600B 2001 2004   CVTHC MSG-TEXT(1:4), MY-HALFWORD
                     0001
 00004  0002 00000E  0293 001C 0000 001B   CALLI SHOW-MESSAGE, *, .SHOW-MESSAGE
 00005  0003 000016  22A1 0000             RTX *
                                           /* INCLUDE: SHOWMSG    */
 . . . (lines omitted)

   MSGID    ODT   ODT Name                          Semantics and ODT Syntax Diagnostics
    MSGID   MI Instruction Stream Semantic Diagnostics
* CPF6412  Attributes of instruction X'0001' operand 2 not valid.
```

Clearly, a numeric data-item was not valid. It is not every time that the MI-compiler (back to a more traditional name) is that nice, sometimes there is no clue to *where* the error is, and sometimes there is even no clue to *what* the error is. It is therefore a good idea to code a little, try it, code a little more, try it, etc. Errors are then most likely in what you just added. Don't type in pages and pages before compiling.

Conditions and Branching

Testing a condition and branching to a different point in the program depending on that condition is a fundamental operation. MI is very rich in this respect. Let us start with a simple loop executing 5 times; each time around the loop we simply display the loop counter:

```
DCL DD LOOP-COUNTER ZND(6,0);                 /* zoned number for ease of display     */

    CPYNV       LOOP-COUNTER, 0;              /* Copy Numeric Value 0 to LOOP-COUNTER */
LOOP:                                         /* a label, LOOP                        */
    ADDN        LOOP-COUNTER, LOOP-COUNTER, 1; /* LOOP-COUNTER = LOOP-COUNTER + 1     */
    CPYBLAP     MSG-TEXT, LOOP-COUNTER, " ";  /* show LOOP-COUNTER                    */
    CALLI       SHOW-MESSAGE, *, .SHOW-MESSAGE;
    CMPNV(B)    LOOP-COUNTER, 5/LO(LOOP);     /* if LOOP-COUNTER < 5, go to LOOP      */
    . . .
```

CPYNV (Copy Numeric Value) is another polymorphic instruction that copies the second numeric operand of any type to the first numeric operand of any type, with automatic type conversion (here from binary 0 to zoned 0000000). Then we add 1 to LOOP-COUNTER and show the result. Nothing new here. Except that one could gripe about the inconsistency of the instruction mnemonics, CPYN**V** has a "**V**" but ADDN has not. This hits me at least once a week.

The crucial instruction is the polymorphic Compare Numeric Value: **CMPNV(B)** , that compares the zoned LOOP-COUNTER with an immediate binary value 5 and sets *one* of four conditions:

- **LO** (if the first operand is lower than the second),
- **HI** (if it is higher),
- **EQ** (if they are equal),
- **UNOR** ("unordered", if any of the operands is an invalid floating-point number, a so-called NaN – Not a Number).

A branch extender or modifier (B) to the operation code then directs the machine to execute one of more branches depending on the condition. In our case, we want to re-execute the loop if the condition is LO (we have not reached the end). The syntax for this is: CMPNV**(B)** LOOP-COUNTER, 5**/LO(LOOP)**. You can branch on several conditions in the same instruction:

```
CMPNV(B) LOOP-COUNTER, 5/LO(LOOP),HI(ERROR),EQ(DONE)
```

You reverse a condition by prefixing it with the letter **N** (for Not), e.g. **/NLO(GREATER-EQUAL)**.

Labels and Branch Points

Labels are names followed by a colon (**:**). A point in the program marked with a label (labeled as is it) is called a *branch point*. You can *only* branch (goto, jump,…) to a labeled branch point. The standard error on other systems where you branch into the middle of some data and your program goes down (and maybe takes the system with it) cannot occur on an AS/400. Well, let's say it is much *harder* to make that happen.

Short Form of Instructions

Several instructions allow a shortened syntax if two operands are the same, e.g.:

```
    ADDN        LOOP-COUNTER, LOOP-COUNTER, 1; /* LOOP-COUNTER = LOOP-COUNTER + 1      */
```

can be shortened with the modifier **(S)** to:

```
    ADDN(S)     LOOP-COUNTER, 1;               /* LOOP-COUNTER = LOOP-COUNTER + 1      */
```

You can combine the short modifier with the branch modifier as in the following example that uses the Subtract Numeric (SUBN) instruction to execute the loop counting the LOOP-COUNTER *down* from 5 to 0:

```
    CPYNV       LOOP-COUNTER, 5;
LOOP:
    CPYBLAP     MSG-TEXT, LOOP-COUNTER, " ";
    CALLI       SHOW-MESSAGE, *, .SHOW-MESSAGE;
    SUBN(SB)    LOOP-COUNTER, 1/POS(LOOP);    /* if positive ( > 0 ) loop again */
```

Here, the decrement, the test, and the branch are all handled by the same instruction. The preferred condition names for this instruction are:

- **POS** (result>0),
- **NEG** (result<0),
- **ZER** (result=0),

but **HI**, **LO**, and **EQ** work just the same. The order of the instruction-modifiers does not matter, so for example SUBN(SB) is the same as SUBN(BS), although it is best to stick to the same order throughout.

Structured Data

In RPG and COBOL we are familiar with the concept of a *record*, that contains various *fields*. We say that the record contains structured data. Consider the following COBOL record:

```
01  MY-RECORD.
    02  MY-BINARY          PIC S9(9)   BINARY.
    02  FILLER             PIC X(1)    VALUE "=".
    02  MY-PACKED          PIC S9V9(8) PACKED.
    02  FILLER             PIC X(6)    VALUE SPACES.
```

It has a direct counterpart in MI as follows:

```
DCL DD MY-RECORD CHAR(16);
    DCL DD MY-BINARY     BIN(4) DEF(MY-RECORD) POS( 1);
    DCL DD *            CHAR(1) DEF(MY-RECORD) POS( 5) INIT("=");
    DCL DD MY-PACKED   PKD(9,8) DEF(MY-RECORD) POS( 6);
    DCL DD *            CHAR(6) DEF(MY-RECORD) POS(11) INIT(" ");
```

Fields are defined (**DEF**) on the record starting in the character position stated (**POS**). The special name "*****" means "no name" or "FILLER" in COBOL. We can now assign values to the fields and move the record as a unit:

```
    CPYNV       MY-PACKED, P'3.14159265';
    CPYNV       MY-BINARY, MY-PACKED;
    CVTHC       MSG-TEXT(1:32), MY-RECORD;
    CALLI       SHOW-MESSAGE, *, .SHOW-MESSAGE;
```

with this result: **00000003**7E**314159265F**404040404040

Note, that the moving 3.14159265 to the binary integer yields (as we expect) the number 3.

Here is a somewhat more complicated structure (that we'll use in a later section):

```
DCL DD  MACHINE-ATTR  CHAR(256) BDRY(16);
    DCL DD  BYTES-PROVIDED    BIN(4) DEF(MACHINE-ATTR) POS(1) INIT(256);
    DCL DD  BYTES-AVAILABLE   BIN(4) DEF(MACHINE-ATTR) POS(5);
    DCL DD  THE-ATTRIBUTES CHAR(248) DEF(MACHINE-ATTR) POS(9);

    DCL DD  THE-TIMESTAMP     CHAR(8)  DEF(THE-ATTRIBUTES) POS( 1);
        DCL DD THE-TIME-HI BIN(4) UNSGND DEF(THE-TIMESTAMP)  POS(1);
        DCL DD THE-TIME-LO BIN(4) UNSGND DEF(THE-TIMESTAMP)  POS(5);

    DCL DD  SERIAL-NBR         CHAR(8) DEF(THE-ATTRIBUTES) POS( 1);

    DCL DD  NETWORK-ATTRS    CHAR(190) DEF(THE-ATTRIBUTES) POS( 1);
        DCL DD SYSTEM-NAME       CHAR(8) DEF(NETWORK-ATTRS)  POS( 1);
        DCL DD *                  BIN(2) DEF(NETWORK-ATTRS)  POS( 9);
        DCL DD NEW-SYSTEM-NAME   CHAR(8) DEF(NETWORK-ATTRS)  POS(11);
        DCL DD *                  BIN(2) DEF(NETWORK-ATTRS)  POS(19);
        DCL DD LOCAL-NETWORK-ID  CHAR(8) DEF(NETWORK-ATTRS)  POS(21);
        DCL DD *                  BIN(2) DEF(NETWORK-ATTRS)  POS(29);
```

Note, how THE-TIME-HI is defined on THE-TIMESTAMP, which in turn is defined on THE-ATTRIBUTES, which is turn is defined on MACHINE-ATTR. Note also, how the substructures THE-TIMESTAMP, SERIAL-NBR and NETWORK-ATTRS are all redefinitions of THE-ATTRIBUTES.

Again, the best way to describe the above data structure is in COBOL:

```
01  MACHINE-ATTR.
    02  BYTES-PROVIDED             PIC S9(9)    BINARY  VALUE 256.
    02  BYTES-AVAILABLE            PIC S9(9)    BINARY.
    02  THE-ATTRIBUTES             PIC X(248).

    02  THE-TIMESTAMP              REDEFINES  THE-ATTRIBUTES.
        03  THE-TIME-HI            PIC 9(9)     BINARY.
        03  THE-TIME-LO            PIC 9(9)     BINARY.

    02  SERIAL-NBR                 REDEFINES  THE-ATTRIBUTES.
                                   PIC X(8).

    02  NETWORK-ATTRS              REDEFINES  THE-ATTRIBUTES.
        03  SYSTEM-NAME            PIC X(8).
        03  FILLER                 PIC S9(4)    BINARY.
        03  NEW-SYSTEM-NAME        PIC X(8).
        03  FILLER                 PIC S9(4)    BINARY.
        03  LOCAL-NETWORK-ID       PIC X(8).
        03  FILLER                 PIC S9(4)    BINARY.
```

Our First "Real" MI-Program

Using the structure we have just defined, we shall now proceed to write our first "real" MI-program, defined as one that does something useful, instead of just serving as a learning tool as our venerable **HELLO** program did. There is a very useful MI-instruction called Materialize Machine Attributes (**MATMATR**). As you can guess, this instruction gives us all kinds of information about the machine you are running on. The description of this single instruction in the MI Functional Reference Manual (see later, how to get one) takes up 34 pages of dense text! The instruction is coded with two operands, basically like this:

```
MATMATR      .MACHINE-ATTR, MATMATR-CONTROL;
```

The second operand, MATMATR-CONTROL, is simply a 2-character control value whose contents determine what the instruction "materializes". You will see that very many MI-instructions use the concept of *materializing* information from internal sources and making it available to you in a suitable data structure, that is often called a *template*. The first operand is not the template, but a *pointer* to the template. We'll explore the concept of a pointer in great detail later. For now, all we need is how to make a pointer point to our data structure. The kind of pointer that simply points to some data is called a *space pointer*. You declare a space pointer by using the **SPCPTR** modifier and you can in the declare-statement directly initialize the pointer to point to the desired data structure, like this:

```
DCL SPCPTR .MACHINE-ATTR INIT(MACHINE-ATTR);
DCL DD  MACHINE-ATTR  CHAR(256) BDRY(16);
```

For reasons we'll come to later, this template has to *aligned* on a 16-byte boundary, indicated by the **BDRY(16)** clause. Note our naming convention of having a pointer to a data-item have the same name as the item, except prefixed by a period. Other people have conventions that use a prefix of "?", "@", or as in C: "*".

We got the layout of the template from the MI Functional Reference (maybe we should invent a new acronym: MIFR). The MIFR also lists the value of the control parameter. The values we are interested in are:

X'0004'	Get the machine Serial Number
X'0100'	Get the Internal Clock value
X'0130'	Get Network Attributes

Declare the control operand as:

```
DCL DD MATMATR-CONTROL CHAR(2);
```

The program code is now straightforward:

```
CPYBLA      MATMATR-CONTROL, X'0004'; /* Get Serial Number */
MATMATR    .MACHINE-ATTR, MATMATR-CONTROL;
CPYBLAP     MSG-TEXT, SERIAL-NBR, " ";      /* Text        */
CALLI       SHOW-MESSAGE, *, .SHOW-MESSAGE;

CPYBLA      MATMATR-CONTROL, X'0100'; /* Get Timestamp     */
MATMATR    .MACHINE-ATTR, MATMATR-CONTROL;
CVTHC       MSG-TEXT(1:16), THE-TIMESTAMP;  /* Binary      */
CALLI       SHOW-MESSAGE, *, .SHOW-MESSAGE;

CPYBLA      MATMATR-CONTROL, X'0130'; /* Get Network Attrs */
MATMATR    .MACHINE-ATTR, MATMATR-CONTROL;
CPYBLAP     MSG-TEXT(1:30), NETWORK-ATTRS, " "; /* mixed   */
CALLI       SHOW-MESSAGE, *, .SHOW-MESSAGE;
. . .
```

The result should be similar to this:

```
From  . . . :   LSVALGAARD     07/26/00   21:46:57
 1234567                         <= serial number
810566576CDA8000                 <= current internal time valu
AS400           APPN             <= system name, etc
```

For completeness, we show the entire program (**MIMCHATR**) on a single page below. Copy and paste it to a file, transfer it to the AS/400, compile and run it.

```
/* Materialize Machine Attributes */
DCL DD MATMATR-CONTROL CHAR(2);
DCL SPCPTR .MACHINE-ATTR INIT(MACHINE-ATTR);

DCL DD  MACHINE-ATTR  CHAR(256) BDRY(16);
DCL DD  MACHINE-ATTR  CHAR(256) BDRY(16);
    DCL DD  BYTES-PROVIDED    BIN(4) DEF(MACHINE-ATTR) POS(1) INIT(256);
    DCL DD  BYTES-AVAILABLE   BIN(4) DEF(MACHINE-ATTR) POS(5);
    DCL DD  THE-ATTRIBUTES CHAR(248) DEF(MACHINE-ATTR) POS(9);

    DCL DD  THE-TIMESTAMP     CHAR(86)  DEF(THE-ATTRIBUTES) POS( 1);
        DCL DD THE-TIME-HI BIN(4) UNSGND DEF(THE-TIMESTAMP)  POS(1);
        DCL DD THE-TIME-LO BIN(4) UNSGND DEF(THE-TIMESTAMP)  POS(5);

    DCL DD  SERIAL-NBR         CHAR(8) DEF(THE-ATTRIBUTES) POS( 1);

    DCL DD  NETWORK-ATTRS    CHAR(190) DEF(THE-ATTRIBUTES) POS( 1);
        DCL DD SYSTEM-NAME       CHAR(8) DEF(NETWORK-ATTRS)  POS( 1);
        DCL DD *                  BIN(2) DEF(NETWORK-ATTRS)  POS( 9);
        DCL DD NEW-SYSTEM-NAME   CHAR(8) DEF(NETWORK-ATTRS)  POS(11);
        DCL DD *                  BIN(2) DEF(NETWORK-ATTRS)  POS(19);
        DCL DD LOCAL-NETWORK-ID  CHAR(8) DEF(NETWORK-ATTRS)  POS(21);
        DCL DD *                  BIN(2) DEF(NETWORK-ATTRS)  POS(29);

    CPYBLA      MATMATR-CONTROL, X'0004'; /* Get Serial Number */
    MATMATR    .MACHINE-ATTR, MATMATR-CONTROL;
    CPYBLAP     MSG-TEXT, SERIAL-NBR, " ";      /* Text        */
    CALLI       SHOW-MESSAGE, *, .SHOW-MESSAGE;

    CPYBLA      MATMATR-CONTROL, X'0100'; /* Get Timestamp     */
    MATMATR    .MACHINE-ATTR, MATMATR-CONTROL;
    CVTHC       MSG-TEXT(1:16), THE-TIMESTAMP;  /* Binary      */
    CALLI       SHOW-MESSAGE, *, .SHOW-MESSAGE;

    CPYBLA      MATMATR-CONTROL, X'0130'; /* Get Network Attrs */
    MATMATR    .MACHINE-ATTR, MATMATR-CONTROL;
    CPYBLAP     MSG-TEXT(1:30), NETWORK-ATTRS, " "; /* mixed   */
    CALLI       SHOW-MESSAGE, *, .SHOW-MESSAGE;


    RTX         *;

%INCLUDE SHOWMSG
```

Pointers, Pointers, and Pointers

Space Data Object

In chapter 2 we developed a program (**MIMCHATR**) to materialize machine attributes into a structured character data-item called a template. The various data-items were declared to be in specific positions relative to the beginning of the area. Creating and (especially maintaining) such a sequence of absolute positions is tedious and prone to errors. There is a much more elegant way of specifying a data structure by using a *space data object*, which is defined as a *based* (i.e. beginning at a location in memory identified by a pointer) 32,767-character string. What in effect we are doing is telling the MI-compiler that no matter what the data 'looks' like in memory, at the location the pointer this structure is based upon points to, we want to map this layout over it so that we can access it using the structures data-items. A space data object is an internal program declaration of a storage mapping. No real space in memory is allocated for the structure, but the structure is laid over an actual character string determined by the pointer, allowing us to reference sections of the data by the data-item names we used in defining the data space object. Here is the declaration of the materialization area using a space data object (**DCL SPC**):

```
DCL DD MATERIALIZE-AREA CHAR(256) BDRY(16);     /* This is the actual data-item */

DCL SPCPTR .MACHINE-ATTR  INIT(MATERIALIZE-AREA); /* Points to actual data-item */
DCL SPC     MACHINE-ATTR  BAS(.MACHINE-ATTR);     /* based upon the pointer     */
    DCL DD  BYTES-PROVIDED  BIN(4) DIR;
    DCL DD  BYTES-AVAILABLE BIN(4) DIR;

        DCL DD  THE-TIMESTAMP    CHAR(8) DIR POS(9);
            DCL DD THE-TIME-HI       BIN(4) UNSGND DIR;
            DCL DD THE-TIME-LO       BIN(4) UNSGND DIR;

        DCL DD  SERIAL-NBR       CHAR(8) DIR POS(9);

        DCL DD  NETWORK-ATTRS  CHAR(190) DIR POS(9);
            DCL DD SYSTEM-NAME       CHAR(8) DIR;
            DCL DD *                  BIN(2) DIR;
            DCL DD NEW-SYSTEM-NAME   CHAR(8) DIR;
            DCL DD *                  BIN(2) DIR;
            DCL DD LOCAL-NETWORK-ID  CHAR(8) DIR;
            DCL DD *                  BIN(2) DIR;
```

If you compare this with the previous definition, you will appreciate the simplicity gained. There is no need to tediously count character positions as the "direct" (**DIR**) clause instructs the compiler to automatically increment its internal position counter from its previous value, starting at 1 and increasing this internal position counter with the size of each item. You can still, if needed, specify a **POS** clause to explicitly change the position counter. We did this thrice to specify that THE-TIMESTAMP, SERIAL-NBR, and NETWORK-ATTR all start at the same position (and thus overlay each other). Because a space data object only maps into the real area and could map into different areas by changing the pointer, we cannot initialize data-items in the map, so the following would be an incorrect definition;

```
DCL SPCPTR .MACHINE-ATTR  INIT(MATERIALIZE-AREA); /* Points to actual data-item */
DCL SPC     MACHINE-ATTR  BAS(.MACHINE-ATTR);     /* based upon the pointer     */
    DCL DD  BYTES-PROVIDED  BIN(4) DIR;
    DCL DD  BYTES-AVAILABLE BIN(4) DIR;

        DCL DD  THE-TIMESTAMP    CHAR(8) DIR POS(9);
            DCL DD THE-TIME-HI       BIN(4) UNSGND DIR INIT( X'0101'); /*<- incorrect */
            DCL DD THE-TIME-LO       BIN(4) UNSGND DIR;
…
```

But we can do it in the actual data-item being pointed to, in the instruction section of the program:

```
    CPYNV       BYTES-PROVIDED, 256;
```

The rest of the code stays the same.

Setting a Space Pointer

While the `INIT` clause can establish addressability to a data-item in a space pointer declaration, like the one we used above: `DCL SPCPTR .MACHINE-ATTR INIT(MATERIALIZE-AREA)`, the connection, in this example that the space pointer `MACHINE-ATTR` is to point to the area of memory that the compiler has allocated for the `MATERIALIZE-AREA` data-item, is established once when the program starts. If we need to point to a different area, we need a way to set addressability in a space pointer dynamically under program control. The Set Space Pointer (**SETSPP**) instruction does just that. So, instead of initializing the pointer in the declaration, we could have written:

```
DCL DD MATERIALIZE-AREA CHAR(256) BDRY(16);   /* This is the actual data-item.   */
DCL SPCPTR .MACHINE-ATTR;                     /* Does not point to anything yet  */

    SETSPP     .MACHINE-ATTR, MATERIALIZE-AREA; /* Only now can be use the data  */
    CPYNV       BYTES-PROVIDED, 256;            /*    based on the pointer       */
```

Explore Pointers: Materialize Invocation Stack

This problem comes up regularly: how do I find out which program called the current program? As programs call each other an *invocation stack* is built up. You can see the contents of the stack using the **DSPJOB** command, selection 11:

```
       Program
Rqs    or
Lvl    Procedure   Library     Statement        Instruction
       QCMD        QSYS                            043B
       QUICMENU    QSYS                            00C1
  1    QUIMNDRV    QSYS                            0502
  2    QUIMGFLW    QSYS                            04B5
  3    QUICMD      QSYS                            0419
       MIINVSTK    LSVALGAARD                      000F
```

This problem is well suited to explore various pointer types and their use and is non-trivial. A quick look though the MIFR table of contents finds an MI instruction named **MATINVS** (MATerialize INVocation Stack) that looks to be a prime candidate to help us achieve a solution. The **MATINVS** MI-instruction takes as its first operand a space pointer to the materialization area and as its second operand a new kind of pointer that we haven't covered yet, called a *system pointer*, to the job for which to retrieve the invocation stack. At this point we are only interested in our own job. Using a *null* operand, "*", selects our own job. So this is the instruction we shall use:

```
    MATINVS   .THE-STACK, *; /* OWN JOB */
```

The stack has no set depth, so we don't know how many programs will be on it. There are two methods we could use static memory or dynamically allocated memory.

Static Storage Method

One way (which we shall use initially) is to allocate a fixed area of a reasonably large size, say fifty programs. As before, the template begins with two binary numbers, the first giving the size of the materialization area, the second will be set to how many bytes were actually returned ("materialized") by the instruction. The number of entries on the stack is returned as a binary number in the third position. The list of entries is returned in the materialization area. Because each entry of the list contains pointers and because pointers (for historical reasons) must be aligned on 16-byte boundaries, the entire area must also be aligned on a 16-byte boundary. Finally, the list within the area must also be aligned on a 16-byte boundary. The net effect of all these alignment requirements is a 4-byte "hole" or slack-area. In descriptions of the MI-instructions in the MIFR, such holes are often designated as "reserved". When creating data definitions in MI programs, such unused space is conventionally named "*". You could also simply omit any reference to it, if data-item following the unused space is defined by its explicit position using the `POS(n)` clause. Here is then the full declaration:

```
DCL SPCPTR .THE-STACK;
DCL DD      THE-STACK CHAR(6416) BDRY(16);
    DCL DD  STK-BYTES-PRV      BIN(4) DEF(THE-STACK) POS( 1);
    DCL DD  STK-BYTES-AVL      BIN(4) DEF(THE-STACK) POS( 5);
    DCL DD  STK-NBR-OF-ENTRIES BIN(4) DEF(THE-STACK) POS( 9);
    DCL DD  *                  BIN(4) DEF(THE-STACK) POS(13);
```

```
    DCL DD  STK-ENTRY(50)   CHAR(128) DEF(THE-STACK) POS(17);
```

Note how we declare an *array* of 50 entries. According to the MIFR, each entry must be 128 characters long. The exact layout of an invocation stack entry is given in all its gory details in the MIFR. What is of immediate interest to us is only the fact the a *system pointer* to the program occupying this position of the stack is stored starting in position 33 within each entry:

```
DCL DD THE-ENTRY CHAR(128) BDRY(16);
    DCL SYSPTR .THE-ENTRY-PGM DEF(THE-ENTRY) POS(33);
```

To process the *n*'th entry, we must either copy it to an entry holding area (like the one we just declared): `CPYBLA THE-ENTRY, STK-ENTRY (n)`, or define a space data object based on a space pointer set dynamically to the *n*'th entry: `SETSPP .THE-ENTRY, STK-ENTRY(n)` using this declaration instead:

```
DCL SPCPTR .THE-ENTRY;
DCL DD      THE-ENTRY CHAR(128) BAS(.THE-ENTRY);
    DCL SYSPTR .THE-ENTRY-PGM DEF(THE-ENTRY) POS(33);
```

To materialize the stack, we code:

```
GET-STACK:
    CPYNV      STK-BYTES-PRV, 6416;   /* area size provided */
    SETSPP    .THE-STACK, THE-STACK;  /* point to the area  */
    MATINVS      .THE-STACK,  *;                    /*  your  own  job              */
```

We process the stack entries with a simple loop:

```
DCL DD PGM-NBR BIN(4);

    CPYNV      PGM-NBR, 0;
NEXT-PROGRAM:
    ADDN(S)    PGM-NBR, 1;
    CMPNV(B)   PGM-NBR, STK-NBR-OF-ENTRIES/HI(DONE);
    CPYBWP     THE-ENTRY, STK-ENTRY(PGM-NBR);
    . . . /* process this entry */
    B          NEXT-PROGRAM;
DONE:
```

Since each entry returned contains at least one pointer, we need a special copy-instruction **CPYBWP** (Copy Bytes With Pointers). Pointers carry special *tag bits* that must be set for the pointer to be valid. The usual copy instructions do not preserve the tag bits, hence the special instruction. We shall deal extensively with the issue of tag bits later on.

Alternatively, we can use a space pointer to point dynamically to the current entry, and then access the entry through a space data object based on the pointer:

```
    CPYNV      PGM-NBR, 0;
NEXT-PROGRAM:
    ADDN(S)    PGM-NBR, 1;
    CMPNV(B)   PGM-NBR, STK-NBR-OF-ENTRIES/HI(DONE);
    SETSP     .THE-ENTRY, STK-ENTRY(PGM-NBR);
    . . . /* process this entry */
    B          NEXT-PROGRAM;
DONE:
```

In both cases, the code for processing the entry is the same. We want to show the name of the program object and the name of the library (called a *context* at the MI-level) where the program object is stored. The materialized entry returned from `MATINVS` does not contain this information directly. Instead, we just get a system pointer to the program object, remember how we defined the data-item `.THE-ENTRY-PGM` as a system pointer (`SYSPTR`) at position 33 of `THE-ENTRY`?. A system pointer uniquely identifies an object in the system, but the identifier is in an internal binary format. The Materialize Pointer (**MATPTR**) instruction can return the *symbolic* identification (i.e. the type, name, and context) of the object identified by the pointer. As with all materialize-instructions we provide a space pointer to the area where we want the information to be returned:

```
DCL SPCPTR .PTR-TEMPLATE INIT(PTR-TEMPLATE);
DCL DD      PTR-TEMPLATE CHAR(76);
    DCL DD  PTR-PROVIDED        BIN(4) DEF(PTR-TEMPLATE) POS( 1);
    DCL DD  PTR-RETURNED        BIN(4) DEF(PTR-TEMPLATE) POS( 5);
    DCL DD  PTR-TYPE           CHAR(1) DEF(PTR-TEMPLATE) POS( 9);
```

```
    DCL DD  PTR-DATA          CHAR(68) DEF(PTR-TEMPLATE) POS(10);
        DCL DD PTR-CTX-TYPE   CHAR( 2) DEF(PTR-DATA)     POS ( 1);
        DCL DD PTR-CTX-NAME   CHAR(30) DEF(PTR-DATA)     POS ( 3);
        DCL DD PTR-OBJ-TYPE   CHAR( 2) DEF(PTR-DATA)     POS (33);
        DCL DD PTR-OBJ-NAME   CHAR(30) DEF(PTR-DATA)     POS (35);
        DCL DD PTR-...        CHAR( 4) DEF(PTR-DATA)     POS (65);
```

First, we specify how much of the data that could be materialized we are really interested in, then we materialize the object information for THE-ENTRY-PGM identified by the system pointer returned in the stack entry:

```
    CPYNV       PTR-PROVIDED, 76;
    MATPTR     .PTR-TEMPLATE, .THE-ENTRY-PGM;
```

Dealing with Exceptions

If we run the program that we have put together so far (try it) we get a "hardware protection" error (MCH6801) at security levels above 30 (and you don't run below, do you?). This is because some of the programs in the stack are located in the *system domain*, and therefore are not accessible from a user-state program:

```
        QCMD         QSYS          user domain
        QUICMENU     QSYS          system domain
  1     QUIMNDRV     QSYS          system domain
  2     QUIMGFLW     QSYS          system domain
  3     QUICMD       QSYS          system domain
        MIINVST0     LSVALGAARD    user domain
```

We would like to monitor for this exception and deal gracefully with the situation, e.g. simply skip programs we are not allowed to see. You can declare an exception monitor (**EXCM**) to catch exceptions and direct the flow of the program to your exception handler. Here is how:

```
DCL EXCM * EXCID(H'4401') BP(NEXT-PROGRAM) CV("MCH") IMD;
```

This monitor is unnamed (the "*") and takes effect immediately (the **IMD**) upon the exception being signaled transferring control to the branch point (the BP) NEXT-PROGRAM. The exception identifier has two parts, a hexadecimal value (the EXCID) and a so-called compare value (the CV). A tricky detail is that for MCH (Machine Check) exceptions, the value reported (e.g. MCH6801) is *not* what you should use as the exception identifier in the exception monitor. Instead, the first two digits and the last two digits must be converted from decimal to hexadecimal *separately*. Since the decimal value 68 = 6*10+8 has the hexadecimal value 44 = 4*16+4, we must monitor for MCH4401 to detect MCH6801. I have given up on trying to see the logic behind this and have accepted that that's the way it works. For CPF and other exceptions, no such conversion is needed as they are already in hex format; only MCH exceptions have this quirk. This difference is constant pain.

We can now complete the processing to show the value of the pointer and the names of the context and program using our standard show-message routine:

```
    CPYNV       PGM-NBR, 0;
NEXT-PROGRAM:
    ADDN(S)     PGM-NBR, 1;
    CMPNV(B)    PGM-NBR, STK-NBR-OF-ENTRIES/HI(DONE);
    CPYBWP      THE-ENTRY, STK-ENTRY(PGM-NBR);

    CPYNV       PTR-PROVIDED, 76;
    MATPTR     .PTR-TEMPLATE, .THE-ENTRY-PGM;

    CPYBREP     MSG-TEXT, " ";
    CVTHC       MSG-TEXT(1:32), THE-ENTRY(33:16);

    CPYBLAP     MSG-TEXT(34:11), PTR-CTX-NAME, " ";
    CPYBLAP     MSG-TEXT(46:11), PTR-OBJ-NAME, " ";
    CALLI       SHOW-MESSAGE, *, .SHOW-MESSAGE;
    B           NEXT-PROGRAM;

DONE:
    RTX         *;

%INCLUDE SHOWMSG
```

Here is a typical result (note that only the user domain programs are showing):

```
From  . . . :   LSVALGAARD     08/03/00   20:20:43
0000000000000000002433117B98000200 QSYS        QCMD
00000000000000000067C6EE75F000200 LSVALGAARD  MIINVSTK
```

Automatic Storage Allocation

We had allocated an array of 50 entries for the materialized invocation stack. The materialized data was part of the *static storage* that is allocated when the program is first called. If we do not wish to be limited to a predefined number of entries, it is possible to allocate the stack in *automatic storage* and to modify its size dynamically as needed. The first step is to declare the stack with the **AUTO** clause with the smallest possible size (8 characters) holding only the number of bytes provided and the number of bytes available for materialization:

```
DCL SPCPTR .THE-STACK;
DCL DD      THE-STACK CHAR(8) AUTO  BDRY(16);
    DCL DD  STK-BYTES-PRV      BIN(4) DEF(THE-STACK) POS( 1);
    DCL DD  STK-BYTES-AVL      BIN(4) DEF(THE-STACK) POS( 5);
```

We then set the number of bytes provided to cover the minimum size, set a space pointer to point to the materialization area, and materialize the stack:

```
    CPYNV      STK-BYTES-PRV, 8;      /* Minimum                    */
    SETSPP    .THE-STACK, THE-STACK;  /* Point to the receiver area */
    MATINVS   .THE-STACK, *;          /* Materialize the minimum    */
```

As result we get the number of bytes needed, `STK-BYTES-AVL`, to receive all entries in the stack. The Modify Automatic Storage instruction, **MODASA**, can then be used to change the size of the area pointed to:

```
    MODASA    .THE-STACK, STK-BYTES-AVL; /* Change size of area */
```

Because this operation may allocate a different area of storage, we need to repeat the instruction that sets a space pointer to point to the area, and then finally materialize the invocation stack again::

```
    SETSPP    .THE-STACK, THE-STACK;  /* Point to changed area  */
    MATINVS   .THE-STACK, *;          /* Materialize full stack */
```

To access the entries, we can define an array as before. Since we cannot define a variable size array, we define the array with size 1, and then switch off subscript checking (using the Override Program Attribute instruction, `OVRPGATR`):

```
    DCL DD  STK-NBR-OF-ENTRIES BIN(4) DEF(THE-STACK) POS( 9);
    DCL DD  STK-ENTRY(1)    CHAR(128) DEF(THE-STACK) POS(17);

    OVRPGATR   1, 2; /* Do not constrain array references */
```

To summarize:

```
DCL SPCPTR .THE-STACK;
DCL DD      THE-STACK CHAR(8) AUTO  BDRY(16);
    DCL DD  STK-BYTES-PRV      BIN(4) DEF(THE-STACK) POS( 1);
    DCL DD  STK-BYTES-AVL      BIN(4) DEF(THE-STACK) POS( 5);
    DCL DD  STK-NBR-OF-ENTRIES BIN(4) DEF(THE-STACK) POS( 9);
    DCL DD  STK-ENTRY(1)    CHAR(128) DEF(THE-STACK) POS(17);

GET-STACK:
    CPYNV      STK-BYTES-PRV, 8;      /* Minimum                    */
    SETSPP    .THE-STACK, THE-STACK;  /* Point to the receiver area */
    MATINVS   .THE-STACK, *;          /* Materialize the minimum    */

    MODASA    .THE-STACK, STK-BYTES-AVL; /* Change size of area */

    SETSPP    .THE-STACK, THE-STACK;
    MATINVS   .THE-STACK, *;

    OVRPGATR   1, 2; /* Do not constrain array references */

    CPYNV      PGM-NBR, 0;
NEXT-PROGRAM:
    …
```

Instruction Pointers

We have several times used the following instruction to call the SHOW-MESSAGE subroutine without really understanding its operands:

```
    CALLI       SHOW-MESSAGE, *, .SHOW-MESSAGE;
```

The third operand **.SHOW-MESSAGE** is an *instruction pointer*. The **CALLI** instruction stores the return point in this pointer variable then goes to the SHOW-MESSAGE **entry** point which is **int**ernal to our program:

```
DCL INSPTR .SHOW-MESSAGE;          /* declare instruction pointer  */
ENTRY       SHOW-MESSAGE INT;
        . . .                      /* other processing  . . .      */
        B          .SHOW-MESSAGE;  /* return to where it came from */
```

This may look very strange in the source code, and if you're not thinking in MI-mode when reading the source, it may look like the final branch statement of SHOW-MESSAGE will actually cause the SHOW-MESSAGE subroutine to be executed again causing an endless loop. However what really happens is that when SHOW-MESSAGE is finished, the subroutine returns using a branch-instruction (B) passing program control back to the return point instruction that is *pointed* to by .SHOW-MESSAGE pointer.

Arithmetic and Timestamps

Timestamps

The AS/400 maintains an internal clock whose value is often used as a fine-grained timestamp. The clock value is 64 bits long and increments by 4096 every microsecond (or at least looks like it does as the exact way it counts may depend on the hardware implementation). The starting time (12:03:06.3148pm on August 23rd, 1928) of the counter was chosen such that the most significant bit would change ("roll over") at the end of day on December 31st, 1999. In this chapter we shall develop a program to convert the timestamp into a more conventional text format: YYYYMMDDhhmmss. This allows us to explore several arithmetical instructions on various data types. We learned in Chapter 2 how to materialize the machine clock. The materialization area had this structure:

```
DCL DD MACHINE-CLOCK CHAR(2) INIT(X'0100');
DCL SPCPTR .MACHINE-ATTR INIT(MACHINE-ATTR);
DCL DD      MACHINE-ATTR CHAR(24) BDRY(16);
    DCL DD  MAT-MAX-SIZE   BIN(4) DEF(MACHINE-ATTR)  POS( 1) INIT(16);
    DCL DD  MAT-ACT-SIZE   BIN(4) DEF(MACHINE-ATTR)  POS( 5);
    DCL DD  MAT-TIMESTAMP CHAR(8) DEF(MACHINE-ATTR)  POS( 9);
        DCL DD MAT-TIME-HI BIN(4) UNSGND DEF(MAT-TIMESTAMP) POS(1);
        DCL DD MAT-TIME-LO BIN(4) UNSGND DEF(MAT-TIMESTAMP) POS(5);
```

Convert Timestamp to Date and Time

The 8-character timestamp consists of two *unsigned* binary halves, a high-order part and a low-order part. We need to do this because MI does not support 8-byte binary values `BIN(8)`. Instead, we shall use packed decimal arithmetic, so the first task is to convert the 8-byte binary value into a packed decimal number. Let us define the following variables:

```
DCL DD TIMESTAMP     PKD(21,0);    /* Can hold 64 bits unsigned */
DCL DD TIMESTAMP-HI  PKD(11,0);    /* Can hold 32 bits unsigned */
DCL DD TIMESTAMP-LO  PKD(11,0);    /* Can hold 32 bits unsigned */
DCL DD TWO**32       PKD(11,0) INIT(P'4294967296');    /* 2**32    */
```

Since we *can* copy the unsigned 32-bit binary halves of the timestamp to their corresponding packed values, `TIMESTAMP-HI` and `TIMESTAMP-LO`, we can easily compute the packed decimal full value of the timestamp: `TIMESTAMP` = `TIMESTAMP-HI` * 2^{32} + `TIMESTAMP-LO`:

```
    CPYNV       TIMESTAMP-LO, MAT-TIME-LO;  /* copy unsigned binaries */
    CPYNV       TIMESTAMP-HI, MAT-TIME-HI;

    MULT        TIMESTAMP, TIMESTAMP-HI, TWO**32;
    ADDN(S)     TIMESTAMP, TIMESTAMP-LO;
```

Since the clock increments by 4096 every microsecond there are 4,096,000,000 "clicks" in one second. To compute the number of seconds that have elapsed since the starting time, we must divide by 4,096,000,000:

```
    DIV(SR)     TIMESTAMP, 4096000000;        /* Now seconds, rounded */
```

Because we had decided to work to an accuracy of one second only, we *rounded* the result using the **`R`** operation extender on the Divide instruction (**`DIV`**). Although the AS/400 internal clock starting time of 12:03:06.3148pm on August 23rd, 1928 gave a easy number to work with for the start of the year 2000, it also gives us a ugly number to use for date arithmetic, so we will make some initial adjustments to the timestamp returned by the `MATMATR` instruction internally in our program to make the mathematics a little easier. We can compensate for the awkward starting time within the day by adding the number of seconds from the previous midnight up to 12:03:06pm:

```
    ADDN(S)     TIMESTAMP, 43386;                      /* 12:03:06pm */
```

We now have the number of seconds elapsed since the beginning of the starting day. To get the number of days, `NBR-DAYS`, that have elapsed, we divide by the number of seconds in a day, 86400, and retain the remainder as the number of seconds, `NBR-SECONDS`, elapsed within the last day. The Divide with Remainder instruction (**`DIVREM`** *`quotient`*, *`dividend`*, *`divisor`*, *`remainder`*) does this handily:

```
    DIVREM     NBR-DAYS, TIMESTAMP, 86400, NBR-SECONDS;
```

Converting to a Date

Taking into account the intricacies of the calendar is eased by the fact that year 2000 is a leap year, so we don't need to use the 100-year rule. Again, to make the calculations easier, we can compensate for the awkward starting day by reducing the number of days by 131, thus making the starting day the beginning of the year 1929:

```
    SUBN(S)    NBR-DAYS, 131;  /* Was: Aug 23,1928, Now: Jan 01,1929 */
```

Four consecutive years, called a *period*, contains 1461 days. The number of periods that have elapsed and the number of days within the last period are simply the quotient and the remainder of the above NBR-DAYS divided by 1461:

```
    DIVREM     NBR-PERIODS, NBR-DAYS, 1461, NBR-DAYS;
```

Now, the number of years elapsed is the number of periods times four; add to that the beginning year, 1929, and we are almost there:

```
    MULT       NBR-YEARS, NBR-PERIODS, 4;
    ADDN(S)    NBR-YEARS, 1929;
```

The number of days within the last period divided by the number of days in a year gives the number of complete years that must still be added:

```
    DIVREM     ADD-YEARS, NBR-DAYS, 365, NBR-DAYS;
    ADDN       YEAR, NBR-YEARS, ADD-YEARS;
```

We accumulate the result as *zoned* numbers in:

```
DCL DD YYYYMMDDHHMMSS CHAR(14);
    DCL DD YEAR  ZND(4,0) DEF(YYYYMMDDHHMMSS) POS( 1);
    DCL DD MONTH ZND(2,0) DEF(YYYYMMDDHHMMSS) POS( 5);
    DCL DD DAY   ZND(2,0) DEF(YYYYMMDDHHMMSS) POS( 7);
    DCL DD HOUR  ZND(2,0) DEF(YYYYMMDDHHMMSS) POS( 9);
    DCL DD MIN   ZND(2,0) DEF(YYYYMMDDHHMMSS) POS(11);
    DCL DD SEC   ZND(2,0) DEF(YYYYMMDDHHMMSS) POS(13);
```

The remainder was the number of days within the last incomplete year. We save that in NBR-OF-DAYS for later on. Because of the choice of starting year, 1929, the last year of any period (for which ADD-YEARS is 3) is a leap year, e.g. 1932. If ADD-YEARS is less than 3, we know that the year is not a leap year. The task now is to find the month based of NBR-DAYS. We define a table with 12 entries where entry number *m* is the number of days from the beginning of the year until month *m*. To find the month, we loop through the table from the end until the entry value is not less than NBR-DAYS:

```
          /* Day Base for: JanFebMarAprMayJunJulAugSepOctNovDec*/
DCL DD DAYS CHAR(36) INIT("000031059090120151181212243273304334");
DCL DD DAYS-ACCUM(12) ZND(3,0) DEF(DAYS) POS(1);

    CPYNV      M, 13;
    CMPNV(B)   ADD-YEARS, 3/LO(FIND-MONTH);
LEAP-YEAR:
    CMPNV(B)   NBR-DAYS, 59/LO(FIND-MONTH),EQ(FEB-29TH);
    SUBN(S)    NBR-DAYS, 1;
FIND-MONTH:                                      <--------------------.
    SUBN(S)    M, 1;                                                  |
    SUBN(B)    DAY-MONTH, NBR-DAYS, DAYS-ACCUM(M)/NEG(FIND-MONTH); ----'
    ADDN       DAY, DAY-MONTH, 1;
    CPYNV(B)   MONTH, M/POS(COMPUTE-TIME);
FEB-29TH:
    CPYNV      MONTH, 2;
    CPYNV      DAY, 29;

COMPUTE-TIME:
```

Study the above code carefully. Note especially how February 29th is handled. Note also the use of *two* branch-conditions in:

```
    CMPNV(B)   NBR-DAYS, 59/LO(FIND-MONTH),EQ(FEB-29TH);
```

Because the month value, `M`, is always positive (that is: greater than zero), we can save a branch-instruction by using `POS` as a branch-condition in

```
    CPYNV(B)   MONTH, M/POS(COMPUTE-TIME);
```

to avoid falling into the `FEB-29TH` processing. The final step is to compute the hours, minutes, and seconds:

```
COMPUTE-TIME:
    DIVREM     HOUR, NBR-SECONDS, 3600, NBR-SECONDS;
    DIVREM     MIN,  NBR-SECONDS,   60, SEC;
```

Complete Program to Show Current Date and Time

Putting everything together we get the following complete program (**`MIRTVDT`**):

```
DCL DD MACHINE-CLOCK CHAR(2) INIT(X'0100');
DCL SPCPTR .MACHINE-ATTR INIT(MACHINE-ATTR);
DCL DD      MACHINE-ATTR CHAR(24) BDRY(16);
    DCL DD  MAT-MAX-SIZE   BIN(4) DEF(MACHINE-ATTR)  POS( 1) INIT(16);
    DCL DD  MAT-ACT-SIZE   BIN(4) DEF(MACHINE-ATTR)  POS( 5);
    DCL DD  MAT-TIMESTAMP CHAR(8) DEF(MACHINE-ATTR)  POS( 9);
        DCL DD MAT-TIME-HI BIN(4) UNSGND DEF(MAT-TIMESTAMP) POS(1);
        DCL DD MAT-TIME-LO BIN(4) UNSGND DEF(MAT-TIMESTAMP) POS(5);

DCL DD TIMESTAMP     PKD(21,0); /* CAN HOLD 64-BIT UNSIGNED */
DCL DD TIMESTAMP-HI  PKD(11,0);
DCL DD TIMESTAMP-LO  PKD(11,0);
DCL DD TWO**32       PKD(11,0) INIT(P'4294967296');
DCL DD NBR-SECONDS   PKD(15,0);
DCL DD NBR-DAYS      BIN(4);
DCL DD NBR-YEARS     BIN(4);
DCL DD ADD-YEARS     BIN(4);
DCL DD NBR-PERIODS   BIN(4);
DCL DD DAY-MONTH     BIN(4);
DCL DD D             BIN(4);
DCL DD S             BIN(4);
DCL DD M             BIN(4);
         /* DAY BASE FOR: JanFebMarAprMayJunJulAugSepOctNovDec*/
DCL DD DAYS CHAR(36) INIT("000031059090120151181212243273304334");
DCL DD DAYS-ACCUM (12)ZND(3,0) DEF(DAYS) POS(1);

DCL DD YYYYMMDDHHMMSS CHAR(14);
    DCL DD YEAR  ZND(4,0) DEF(YYYYMMDDHHMMSS) POS( 1);
    DCL DD MONTH ZND(2,0) DEF(YYYYMMDDHHMMSS) POS( 5);
    DCL DD DAY   ZND(2,0) DEF(YYYYMMDDHHMMSS) POS( 7);
    DCL DD HOUR  ZND(2,0) DEF(YYYYMMDDHHMMSS) POS( 9);
    DCL DD MIN   ZND(2,0) DEF(YYYYMMDDHHMMSS) POS(11);
    DCL DD SEC   ZND(2,0) DEF(YYYYMMDDHHMMSS) POS(13);

/*****************************************************************/

    MATMATR   .MACHINE-ATTR, MACHINE-CLOCK;
    CPYNV      TIMESTAMP-LO, MAT-TIME-LO;
    CPYNV      TIMESTAMP-HI, MAT-TIME-HI;
    MULT       TIMESTAMP, TIMESTAMP-HI, TWO**32;
    ADDN(S)    TIMESTAMP, TIMESTAMP-LO;
    DIV(SR)    TIMESTAMP, 4096000000;                 /* NOW SECONDS */
    ADDN(S)    TIMESTAMP, 43386;                      /* 12:03:06 PM */
    DIVREM     NBR-DAYS, TIMESTAMP, 86400, NBR-SECONDS;
    SUBN(S)    NBR-DAYS, 131;  /* WAS: AUG 23,1928, NOW: JAN 01,1929 */
    DIVREM     NBR-PERIODS, NBR-DAYS, 1461, NBR-DAYS; /*     4 YEARS */
    MULT       NBR-YEARS, NBR-PERIODS, 4;
    ADDN(S)    NBR-YEARS, 1929;
    DIVREM     ADD-YEARS, NBR-DAYS, 365, NBR-DAYS;
    ADDN       YEAR, NBR-YEARS, ADD-YEARS;
    CPYNV      M, 13;
    CMPNV(B)   ADD-YEARS, 3/LO(FIND-MONTH);
LEAP-YEAR:
    CMPNV(B)   NBR-DAYS, 59/LO(FIND-MONTH),EQ(FEB-29TH);
    SUBN(S)    NBR-DAYS, 1;
FIND-MONTH:
    SUBN(S)    M, 1;
    SUBN(B)    DAY-MONTH, NBR-DAYS, DAYS-ACCUM(M)/NEG(FIND-MONTH);
    ADDN       DAY, DAY-MONTH, 1;
    CPYNV(B)   MONTH, M/POS(COMPUTE-TIME);
FEB-29TH:
    CPYNV      MONTH, 2;
```

```
    CPYNV      DAY, 29;

COMPUTE-TIME:
    DIVREM     HOUR, NBR-SECONDS, 3600, NBR-SECONDS;
    DIVREM     MIN,  NBR-SECONDS,   60, SEC;

SHOW-DATE-TIME:
    CPYBLAP    MSG-TEXT, YYYYMMDDHHMMSS, " ";
    CALLI      SHOW-MESSAGE, *, .SHOW-MESSAGE;
    RTX        *;

%INCLUDE SHOWMSG
```

If we wanted to include the usual date/time separators, we could have defined them as part of the result structure:

```
DCL DD YYYYMMDDHHMMSS CHAR(19);
    DCL DD YEAR  ZND(4,0) DEF(YYYYMMDDHHMMSS) POS( 1);
    DCL DD *      CHAR(1) DEF(YYYYMMDDHHMMSS) POS( 5) INIT("/");
    DCL DD MONTH ZND(2,0) DEF(YYYYMMDDHHMMSS) POS( 6);
    DCL DD *      CHAR(1) DEF(YYYYMMDDHHMMSS) POS( 8) INIT("/");
    DCL DD DAY   ZND(2,0) DEF(YYYYMMDDHHMMSS) POS( 9);
    DCL DD *      CHAR(1) DEF(YYYYMMDDHHMMSS) POS(11) INIT(" ");
    DCL DD HOUR  ZND(2,0) DEF(YYYYMMDDHHMMSS) POS(12);
    DCL DD *      CHAR(1) DEF(YYYYMMDDHHMMSS) POS(14) INIT(":");
    DCL DD MIN   ZND(2,0) DEF(YYYYMMDDHHMMSS) POS(15);
    DCL DD *      CHAR(1) DEF(YYYYMMDDHHMMSS) POS(17) INIT(":");
    DCL DD SEC   ZND(2,0) DEF(YYYYMMDDHHMMSS) POS(18);
```

Calling Other Programs

External Program Objects

The Original Program Model with its run-time binding of programs was once touted as an important advantage of the AS/400 over other platforms. Later the pendulum swung completely to the other extreme where linking modules together was the fashion and late binding was being shunned as being old-fashioned. Then with Java, the pendulum swung back again. Actually, both approaches have their advantages depending on what your requirements are. A program is a separate, directly executable object having (as all objects at the MI-level) both an external *symbolic* representation (CONTEXT, PROGRAM) and an internal representation as a resolved *system pointer*. A program is called using the **CALLX** (Call eXternal) instruction with the first operand being (ordinarily) a system pointer to the program object. A second operand specifies an argument list. A third operand identifies a list of alternate return points, but is usually left as a null operand ("*****") which will cause the next MI instruction after the CALLX to be executed after the program called by the CALLX returns control.

Obtaining a System Pointer

There are several ways of obtaining a system pointer:

- Initializing the pointer using the INIT clause on a Declare statement
- Resolving the pointer using the Resolve System Pointer instruction (**RSLVSP**)
- Copying an existing pointer using the Copy Bytes With Pointers instruction (**CPYBWP**)
- Retrieving a pointer from the System Entry Point Table (SEPT - discussed in a later chapter)
- Counterfeiting the pointer (discussed in a later chapter)

Initializing a System Pointer

Here is the main syntax for the initial value of a program system pointer:

```
DCL SYSPTR pointername INIT("programname", CTX("libraryname"), TYPE(PGM))
```

The initialization takes place the first time the pointer is used in a session.

Resolving a System Pointer

You can also explicitly insert addressability to an object into a system pointer using the Resolve System Pointer instruction (**RSLVSP**). The instruction takes a 34-byte structure (the *template*) as its second operand and returns a resolved system pointer in its first operand (provided that the object can be found):

```
DCL DD RESOLVE CHAR(34);
    DCL DD RESOLVE-TYPE CHAR( 2) DEF(RESOLVE) POS( 1);
    DCL DD RESOLVE-NAME CHAR(30) DEF(RESOLVE) POS( 3);
    DCL DD RESOLVE-AUTH CHAR( 2) DEF(RESOLVE) POS(33);

DCL SYSPTR .MYPGM;

    CPYBLA      RESOLVE-TYPE, X'0201';
    CPYBLAP     RESOLVE-NAME, "MYPGM", " ";
    RSLVSP     .MYPGM, RESOLVE, *, *;
```

The object type ("02" for a program) and subtype (ordinarily "01") specify what kind of object we are trying to address. A list of a number of AS/400 objects showing their types and subtypes can be found in Appendix 2. Note that the object name is *30* characters long padded as necessary on the right with blanks, hence the CPYBLA**P**-instruction. The last element of the structure (and the last operand) used to contain the authority that should be associated with the pointer. The ability to set authority in a pointer is no longer used for user-domain objects because it constitutes a potential security breach (there is no way to retract or remove the authority once it has been granted). The third operand is a system pointer identifying a context to be searched, or null ("*") specifying that the library list should be used:

```
    RSLVSP      system pointer, template, context, authority
```

Copying a System Pointer

In Chapter 3 we saw an example where a stack entry was copied to a local holding area. Because the entry contained a pointer that we needed to use later on, a special copy-instruction "Copy Bytes With Pointers" (**CPYBWP**) was used that maintain the tag bits of the pointer:

```
NEXT-PROGRAM:
    ADDN(S)    PGM-NBR, 1;
    CMPNV(B)   PGM-NBR, STK-NBR-OF-ENTRIES/HI(DONE);
    CPYBWP     THE-ENTRY, STK-ENTRY(PGM-NBR);
```

System Entry Point Table (SEPT)

Every AS/400 has a space object called the System Entry Point Table, or SEPT for short. The SEPT is a table containing 'pre-resolved' system pointers to a large number of OS/400 programs, and the location within the table for a particular system pointer (that is for general use) has remained the same across every OS/400 release. The SEPT allows a quick look-up of the system pointer for a particular system program that can then be used as the second parameter in the CALLX instruction. The SEPT will be covered briefly later in this chapter, and in much more detail in a later chapter.

Counterfeiting a Pointer

Although not part of the normal programming paradigm for the AS/400 it is possible to construct pointers "out of thin air" for RISC-based AS/400s. This is possible because RISC-based AS/400s rely on *detecting* this blatant security breach instead of *preventing* it. The detection code can be removed and you end up with a valid pointer (valid, but counterfeit – in the sense of not being issued (and checked) by the system). We shall also discuss this in detail, in later chapters.

Encapsulated Objects

With a system pointer to an object you can perform operations on the object that are appropriate for the object: you can execute a program, search an index, retrieve controlled information about the object, etc. You are generally *not supposed* to be able to access the object in an uncontrolled manner, such as inspecting or changing data internal to the object. Objects are said to be encapsulated and thus inviolate. To access or change data inside an object you need a *space pointer* to that data. There is an MI-instruction to create a space pointer from another pointer, "Set Space Pointer From Pointer", SETSPPFP, so you might think that the following code fragment would do the trick:

```
DCL SYSPTR .SYSPTR; /* a system pointer */
DCL SPCPTR .SPCPTR; /* a space  pointer */

    SETSPPFP .SPCPTR, .SYSPTR; /* set space pointer */
```

In fact, it does create a valid space pointer, but *not* to the system object as we want. Instead, a space pointer to the "primary associated space" is created. For many encapsulated objects – especially program objects, the associated space does not contain much of interest, so the freshly minted space pointer is often rather useless. This is, of course, carefully designed to be so.

The Argument List

This is also called the *operand list*, hence the "**OL**" in the syntax for declaring it. First, here is how you declare the argument list in the *calling* program (the caller):

DCL **OL** *name* (*parg1*, *parg2*, …, *pargn*) **ARG**

Second, here is how you declare the corresponding list in the *called* program (the callee):

DCL **OL** *name* (*parg1*, *parg2*, …, *pargn*) **PARM EXT MIN(*m*)**

The number, *m*, of arguments passed (or parameters received) can be smaller (**MIN(*m*)**) than the maximum number, *n*, implied by the number of items (*pargn*) in the list. Up to 255 items can be specified. The "Store

Parameter List Length"-instruction (**STPLLEN**) can be used to obtain the actual number of parameters passed.

The arguments (parameters) are almost always *space pointers* to data-items, although they don't *have* to be. Many HLLs require arguments to be space pointers, so if you wish to be sure that your program could be called by an HLL-program it is wise to stick to that convention. The arguments must be declared *before* the operand list.

Program Call Example

To illustrate the technique we show two sample programs. **MICPGM1** calls **MICPGM2** with two arguments. **MICPGM2** can handle up to three parameters and simply replaces the values of any parameters received by a suitable text, which is displayed by **MICPGM1** when control returns to it.

Here is the main program:

```
DCL SPCPTR .ARG1 INIT(ARG1); /* pointer to 1st argument   */
DCL DD      ARG1 CHAR(10);   /*            1st argument   */

DCL SPCPTR .ARG2 INIT(ARG2); /* pointer to 2nd argument   */
DCL DD      ARG2 CHAR(10);   /*            2nd argument   */

DCL OL      MICPGM2 (.ARG1, .ARG2) ARG; /* argument list */
DCL SYSPTR .MICPGM2;         /* system pointer to callee */

DCL DD RESOLVE CHAR(34);
    DCL DD RESOLVE-TYPE CHAR( 2) DEF(RESOLVE) POS( 1);
    DCL DD RESOLVE-NAME CHAR(30) DEF(RESOLVE) POS( 3);
    DCL DD RESOLVE-AUTH CHAR( 2) DEF(RESOLVE) POS(33);

ENTRY * EXT;
RESOLVE-TO-PGM:
    CPYBLA      RESOLVE-TYPE, X'0201';          /* program type */
    CPYBLAP     RESOLVE-NAME, "MICPGM2" , " "; /* 30-char name */
    RSLVSP     .MICPGM2, RESOLVE, *, *;         /* * is liblist */

LOAD-ARGUMENTS-AND-CALL:
    CPYBLAP     ARG1, "ARG1", " ";    /* value of 1st arg  */
    CPYBLAP     ARG2, "ARG2", " ";    /* value of 2nd arg  */

    CALLX      .MICPGM2, MICPGM2, *;  /* call the program */

BACK-FROM-CALL:
    CAT         MSG-TEXT, ARG1, ARG2; /* concatenate ARGs */
    CALLI       SHOW-MESSAGE, *, .SHOW-MESSAGE;

RETURN:
    RTX        *;

%INCLUDE SHOWMSG
```

Note the **ENTRY** statement. It defines an entry point to the program. By default that would be the first executable instruction, but you can also define it explicitly as we just did. In our example, it has no special name ("*****") and is an *external* entry point (**EXT**). Note also the use of the Concatenate-instruction (**CAT**) that concatenates ARG1 and ARG2 into MSG-TEXT, padding with blanks on the right.

The called program, **MICPGM2**, accepts up to three parameters. Each parameter is a space pointer declared to be a parameter (**PARM**):

```
DCL SPCPTR .PARM1 PARM;
DCL DD      PARM1 CHAR(10) BAS(.PARM1);
```

The PARM1 data item is *based* on a parameter space pointer (.PARM1) as indicated by the BAS clause.

```
DCL SPCPTR .PARM2 PARM;
DCL DD      PARM2 CHAR(10) BAS(.PARM2);

DCL SPCPTR .PARM3 PARM;
DCL DD      PARM3 CHAR(10) BAS(.PARM3);
```

The parameters are external (**EXT**) and at least one must be present:

```
DCL OL MICPGM2 (.PARM1, .PARM2, .PARM3) PARM EXT MIN(1);
DCL DD NBR-PARMS BIN(2);
```

The entry point specifies the operand list defining possible parameters:

```
ENTRY * (MICPGM2) EXT;
    STPLLEN     NBR-PARMS;   /* get number of actual parameters */
    CPYBLAP     PARM1, "PARM1", " "; /* just mark the parameter */
```

We could unconditionally return "PARM1" in the first parameter because we are assured by the system that there will be at least one. For the remaining parameters, we need a guard based on the number of parameters to only allow returning a value if the parameter is present:

```
    CMPNV(B)    NBR-PARMS, 2/LO(=+2); /* if NBR-PARMS not < 2 */
    CPYBLAP     PARM2, "PARM2", " ";: /*     parm2 = "PARM2"  */

    CMPNV(B)    NBR-PARMS, 3/LO(=+2); /* if NBR-PARMS not < 3 */
    CPYBLAP     PARM3, "PARM3", " ";: /*     parm3 = "PARM3"  */
```

At the end, we simply return to the caller:

```
RETURN:
    RTX        *;
```

Relative Branch Conditions

Did you notice the curious branch targets (**=+2**) in the above guards? Let me rewrite the guards in a more traditional way:

```
    CMPNV(B)    NBR-PARMS, 2/LO(AFTER-2);
    CPYBLAP     PARM2, "PARM2", " ";
AFTER-2:
    CMPNV(B)    NBR-PARMS, 3/LO(AFTER-3);
    CPYBLAP     PARM3, "PARM3", " ";
AFTER-3:
```

Labels should be informative as to what the code is doing and not simply place names. I had to invent two labels with little informational contents (`AFTER-2` and `AFTER-3`) just to have targets for the branch conditions. Instead of using an actual label name as a branch condition, MI allows *relative* branch targets. Both positive and negative values may be used. The relative condition '`=+2`' means skip forward two (2) instructions (*including* the instruction with this condition), so we could just as well have written:

```
    CMPNV(B)    NBR-PARMS, 2/LO(=+2);
    CPYBLAP     PARM2, "PARM2", " ";
AFTER-2:
    CMPNV(B)    NBR-PARMS, 3/LO(=+2);
    CPYBLAP     PARM3, "PARM3", " ";
AFTER-3:
```

As always, you can only jump to a branch point (the '**:**' after the label). Because MI is free-form, we can write the label on the same line as the instruction being skipped:

```
    CMPNV(B)    NBR-PARMS, 2/LO(=+2);
    CPYBLAP     PARM2, "PARM2", " ";  AFTER-2:

    CMPNV(B)    NBR-PARMS, 3/LO(=+2);
    CPYBLAP     PARM3, "PARM3", " ";  AFTER-3:
```

Finally, since we are actually not using the label names, we can simply omit them (but keep the colons):

```
    CMPNV(B)    NBR-PARMS, 2/LO(=+2);
    CPYBLAP     PARM2, "PARM2", " ";:

    CMPNV(B)    NBR-PARMS, 3/LO(=+2);
    CPYBLAP     PARM3, "PARM3", " ";:
```

Now the code is no longer cluttered with the arbitrary, invented labels. Some people indent the guarded instructions for added clarity (?):

```
    CMPNV(B)    NBR-PARMS, 2/LO(=+2);
      CPYBLAP   PARM2, "PARM2", " ";:
```

```
    CMPNV(B)    NBR-PARMS, 3/LO(=+2);
      CPYBLAP   PARM3, "PARM3", " ";:
```

An Optimization

Since resolving a pointer takes time, one often uses a first-time flag to avoid unnecessary, subsequent executions of the resolve instruction. The RESOLVE-TYPE can serve as a convenient first-time flag having an initial value of all zeroes:

```
DCL DD RESOLVE CHAR(34);
    DCL DD RESOLVE-TYPE CHAR( 2) DEF(RESOLVE) POS( 1) INIT(X'0000');
    DCL DD RESOLVE-NAME CHAR(30) DEF(RESOLVE) POS( 3);
    DCL DD RESOLVE-AUTH CHAR( 2) DEF(RESOLVE) POS(33);

    CMPBLA(B)   RESOLVE-TYPE, X'0000'/NEQ(=+4); /* if TYPE = zeroes  */
    CPYBLA      RESOLVE-TYPE, X'0201';          /*    TYPE = 0201    */
    CPYBLAP     RESOLVE-NAME, "MYPGM", " ";     /*    NAME = …       */
    RSLVSP     .MYPGM, RESOLVE, *, *;:          /*    Resolve .MYPGM */
```

The System Entry Point Table

To speed up access to operating system functions and also to prevent programs with the same name being present in a context (library) on the library list to be called instead, as you remember, the system maintains a list of *pre-resolved* system pointers to operating systems functions and APIs called the System Entry Point Table (the SEPT). It contains more than 6,000 entries and the number grows with each release of OS/400. In a later chapter we shall look at the SEPT in detail. For now, we'll just touch briefly upon how to use the SEPT. We have used the Send Message API, **QMHSNDM**, several times (in the SHOWMSG include file). It is SEPT entry number 4268. We call the API like this:

```
        CALLX     .SEPT(4268), QMHSNDM, *; /* SEND MSG TO MSGQ */
```

The SEPT is accessed through a pointer found in the *Process Communication Object* (about which much more later). Here is how we declare the array of system pointers, which is the SEPT:

```
DCL SYSPTR .SEPT(6440) BAS(SEPT-POINTER);
DCL SPC PROCESS-COMMUNICATION-OBJECT BASPCO;
    DCL SPCPTR SEPT-POINTER DIR;
```

The secret to all this is the curious **BASPCO** clause, meaning "based on the PCO (Process Communication Object)". A list of public entries into the SEPT and their numbers will be given in appendix 4.

Getting/Setting File Member Information

File Member Information

In this chapter we shall solve a problem that seems to crop up again and again. How to change the date, type, and text information for a physical file member. Often a file is maintained on a PC rather than on the AS/400 and it is desirable to maintain the date information (when created, modified, etc) so that it is the same on both platforms. There is no API to do this, so we have to roll our own. This chapter focuses on how we find out where the various pieces of information are stored and in what format we will find this information. An alternative way of changing some of this information is to open the file member and look at the Open Data Path. We shall explore that approach in a later chapter.

File Member Cursors

Records for physical files are stored in *members*. A physical file can have one or more members. An MI-system object, called tthe *cursor*, identifies each member. A cursor can point directly to a data space (for arrival sequence files) or to a data space through a dataspace index (for keyed sequence files). The object type/subtype for a member cursor is x'0D50'. The cursor name consists firstly of the 10-character filename, then the 10-character member name and is finally padded out with blanks for a total of 30 characters. Given that representaion, we can then resolve a system pointer to the cursor using code similar to the following:

```
    CPYBLA       RESOLVE-TYPE, X'0D50';
    CPYBLAP      RESOLVE-NAME( 1:30), PARM-FILE, " ";
    CPYBLA       RESOLVE-NAME(11:10), PARM-MEMBER;
    RSLVSP      .CURSOR, RESOLVE, .CONTEXT, *;
```

We can also dump the object using SST (note that the object is in the system domain, **8000**):

```
    DISPLAY/ALTER/DUMP                                                          09/05/00  17:33:09   PAGE   1
CURSOR                  SUBTYPE:  50      NAME:  QMISRC    MIHELLO            ADDRESS:  0419D793F7 000000
SEGMENT HEADER   (YYSGHDR)
    TYPE  0001  SIZE  0008  NEWFLAGS  00  FLAGS  89  DOMAIN  8000  OBJECT  0419D793F7 000000   SPACE  0419D793F7 0007F0
EPA HEADER   (YYEPAHDR)
    ATT1      80                JOPT      00                TYPE      0D                STYP      50
    NAME      QMISRC    MIHELLO                             SPATT     80                SPIN      00
    SPSZ      00000810          PBAU      3F10              DVER      4100              TIME      07/27/00  20:09:55
    UPSG      0E6792A4C7 000000 AGSG      0000000000 000000 CTSG      173A0D5C03 000000 OSG       0419D793F7 000000
    RCV2      0002              ASP       0000              PERF      03000000          MDTS      09/03/00  21:06:14
    JPSG      0000000000 000000 CBSG      0000000000 000000 JID       00000000000000000000
    OWAU      FF1C              IPL NUM   00000073          AL1S      0000000000 000000 GRAU      0000
    GRP       000000000000      MAXS      0000              INFO      00000000000000000 ATT2      E0
    COLB      00                LEVL      00000000          USCNT     0000              USDAY     0000
    DJID      00                DENP      0000000000 000000 POSG      3130B03F48 000000 DIRP      0000000000 000000
    AUR       0071              JGEN      0000              TEMP      00000000000000000000000000

    000000   0001000800898000  0419D793F7000000  4001000000000000  0419D793F70007F0  *.........P.7... .........P.7..0*
    000020   80000D50D8D4C9E2  D9C340404040D4C9  C8C5D3D3D6404040  4040404040404040  *...&QMISRC    MIHELLO          *
    000040   4040800000000810  000000083F104100  81069260B4628000  0E6792A4C7000000  *  ................-........G...*
    000060   0000000000000000  173A0D5C03000000  0419D793F7000000  0002000003000000  *...........*.......P.7...........*
==> 000080   813665D6701C0000  0000000000000000  0000000000000000  0000000000000000  *...o.............................*
    0000A0   0000FF1C00000073  0000000000000000  0000000000000000  0000000000000000  *.............................*
    0000C0   0000000000000000  E000000000000000  0000000000000000  0000000000000000  *.............................*
    0000E0   3130B03F48000000  0000000000000000  0071000000000000  0000000000000000  *.............................*
```

Note the modify-timestamp (MDTS) at offset x'80'. This is one of the items we might want to change. The address of the associated space is at offset x'07F0'. We build a space pointer to the associated space from the resolved system pointer with the (now) familiar "Set Space Pointer From Pointer"-instruction:

```
    SETSPPFP    .CURSOR-SPACE, .CURSOR;
```

The Member Header

Investigating the associated space for the cursor we see that at offset x'04' there is a 4-byte binary value (x '00000490') that when added to the address of the associated space (x'…0007F0') points to an area (x'…000c80') where the member descriptive text, type and source change dates are stored. This area is known as the *member header*:

```
DCL SPCPTR .CURSOR-SPACE;
DCL SPC     CURSOR-SPACE BAS(.CURSOR-SPACE);
```

```
    DCL DD  CSR-MBR-HEADER BIN(4) DEF(CURSOR-SPACE) POS(5);
```

Here is the associated space (so you can see for yourself):

```
ASSOCIATED SPACE
   0007E0                                                 90000000 00000490  0000083C000000B0  *                ................*
   000800   00000140000001C6  0000028000000000   000002C000000000  0000000000000000  *... ...F.......................*
   000820 T 0000000000000000  0000000000000000   0000000000000000  0419D793F7000D10  *.........................P.7...*
   000840   0000000000000000  0000000000000000   0000000000000000  0000000000000000  *................................*
   000860   0000000000000000  0000000000000000   8000000000000000  0000000000000000  *................................*
   000880   0000000000000000  0000000000000000   2C00000000000000  0000000000000000  *................................*
   0008A0   C4C2000000000000  0000000000000000   0000000000000000  0000000000000000  *DB..............................*
   0008C0   0000000000000000  0000000000000000   D4C9C8C5D3D3D640  4040000000000000  *.............MIHELLO   ......*
   0008E0   0000000000000000  00000000000000C1   D900E800000000000 0000000000000000  *............AR.Y.............*
   000900   0000000000000000  0000000000000000   0000000000000000  0000000000000000  *................................*
   000920   0000000000000000  0000000000000000   00010001C4C1E3C1  C2C1E2C540400000  *................DATABASE  ..*
   000940   0000000000000000  0010000E000F06B0   0012004500130013  006F004500450045  *.....................?.....*
   000960   0013004500450045  000D0011000000D1   0000000000000000  0000000000000000  *................................*
   000980   0000000000000000  0000000000000000   0000000000000000  0000000000000000  *................................*
   0009A0   0000000000000000  0000000000000000   0000000000000090  0000000000000000  *................................*
   0009C0   0000000000000000  0000000000000000   0000000000000000  0000000000000000  *................................*
            3 LINES    0419D793F7 0009E0    TO    0419D793F7 000A20    SAME AS ABOVE
   000A40   0000000000000000  0022000000000000   0000000300000480  0000000000000000  *................................*
   000A60   0000000000000000  0000000000000000 T 0000000200300000  0000000000003000  *................................*
   000A80 T 0000000000000000  0419D793F7000D10   0000000000000000  34FE0DEF61000B00  *...........P.7.............../...*
   000AA0   8181000000000000  0000000000000000   0000000000000000  0000000000000000  *................................*
   000AC0 T 0000000000000000  0000000000000000   8000000000000000  0419D793F7000C80  *.........................P.7...*
   000AE0   0000000000000000  0000000000000000   0000000000000000  0000000000000000  *................................*
            4 LINES    0419D793F7 000B00    TO    0419D793F7 000B60    SAME AS ABOVE
   000B80   0000000000000000  0000000000000000   0000000000000000  0000000003000000  *................................*
   000BA0   2500000000C3D7C6  0000000000800000   0000003C00000000  0000000000000000  *.....CPF.......................*
   000BC0   0000000000000000  0000000000000000   0000000000000000  0000000000000000  *................................*
   000C20   0000000000000000  005C000000000001   D4C9E2D9C3404040  404000F1C6C4C6C3  *.........*......MISRC     .1FDFC*
   000C40   F9C2C6C4C5F5C5C5  00000000005C0001   0000000100000000  0000000300000000  *9BFDE5EE.....*.................*
   000C60   0000000000000000  0000000000000000 T 0000000000000000  0000000000000000  *................................*

   000C80 T 0000000000000000  26259A9FB5000D10 T 0000000000000000  171C8C8A02000D10  *................................*
   000CA0   0000000000000000  3130B03F48001910   0000000000000000  0000000000000000  *................................*
   000CC0   0000000000000000  0000000000000000   8018FFFFE3C5E7E3  4B4B4B4040404040  *.....................TEXT...      *
   000CE0   4040404040404040  4040404040404040   4040404040404040  4040404040404040  *                                *
   000D00   404040404040D4C9  4040404040404040   0000000000000000  0000F1F0F0F0F7F2  *      MI          .........100072*
   000D20   F7F2F0F0F5F5F5F1  F0F0F0F7F2F7F2F0   F0F9F5F500C4D500  00271003E801F3D5  *7200555 1000727200955.DN.....Y.3N*
   000D40   F1F0F0F0F9F0F5F0  F3F3F4F5F5404040   4040404040404040  4040000000000000  *1000905033455              ......*
   000D60   0000000000000000  0000000000000000   0000000000000000  0000000000000000  *................................*
```

Further analysis (which I'll spare you) reveals that the format of the member header is as follows:

```
DCL SPCPTR .MBR-HEADER;
DCL SPC MBR-HEADER BAS(.MBR-HEADER);
    DCL SYSPTR .MHDR-PREV-MCB          DIR;
    DCL SYSPTR .MHDR-NEXT-MCB          DIR;
    DCL SYSPTR .MHDR-FILE-CB           DIR;
    DCL SYSPTR .MHDR-SHARE-DIR         DIR;
    DCL SYSPTR .MHDR-DATA-DICT         DIR;

    DCL DD MHDR-STATUS      CHAR(2)    DIR;
    DCL DD *                CHAR(2)    DIR;
    DCL DD MHDR-TEXT        CHAR(50)   DIR;
    DCL DD MHDR-TYPE        CHAR(10)   DIR;
    DCL DD *                CHAR(10)   DIR;

    DCL DD MHDR-CHANGE-DATE CHAR(13)   DIR;
    DCL DD MHDR-CREATE-DATE CHAR(13)   DIR;

    DCL DD MHDR-PREFRD-UNIT CHAR(1)    DIR;
    DCL DD MHDR-ALLOC-TYPE  CHAR(2)    DIR;
    DCL DD MHDR-INIT-RECS   BIN(4)     DIR;
    DCL DD MHDR-RECS-EXTEND BIN(2)     DIR;
    DCL DD MHDR-NBR-EXTENDS BIN(2)     DIR;
    DCL DD MHDR-RECOVER-OPT CHAR(1)    DIR;

    DCL DD MHDR-SAVE-DATE   CHAR(13)   DIR;
    DCL DD MHDR-RSTR-DATE   CHAR(13)   DIR;
    DCL DD MHDR-%-DLT-ALLOW CHAR(1)    DIR;
    DCL DD MHDR.USER-AREA   BIN(4)     DIR;
    DCL DD MHDR-OLD-S-DATE  CHAR(13)   DIR;
    DCL DD MHDR-OLD-R-DATE  CHAR(13)   DIR;
    DCL DD MHDR...........  CHAR(1)    DIR;
```

We can get a pointer to the member header by using the "Add to Space Pointer"-instruction (**ADDSPP**) that adds the binary value at offset x'04' (which we'll use as the 3rd operand) to the space pointer to the associated space (which we'll use as the 2nd operand) yielding the required space pointer to the member header, returned to us by ADDSPP in the 1st operand:

```
    ADDSPP      .MBR-HEADER, .CURSOR-SPACE, CSR-MBR-HEADER;
```

Unfortunately, we cannot add a negative offset to get at the functional space of the object (where the system pointer is pointing to). If you try to, you get a run-time exception that the address is outside of the bounds of the associated space. What we really would like to do is to manufacture a space pointer directly from the system pointer. That problem we just solved in chapter 7 and we'll apply the solution here:

```
    CPYBWP      .POINTER, .CURSOR;
    CPYBREP      PTR-OFFSET, X'00';
    CALLX       .MIMAKPTR, MIMAKPTR, *;
```

And now we can access the modify-timestamp:

```
DCL SPC     MBR-CURSOR BAS(.POINTER);
    DCL DD  MBR-CHANGE-TIMESTAMP CHAR(8) DEF(MBR-CURSOR) POS(129);
```

Converting Dates to Timestamps

Back in chapter 4, we went through the data structures and algorithm needed for converting an MI-timestamp to a more readable date and time. The reverse conversion uses the same data structures and the code is straightforward (but still interesting). A new twist is that we don't want the real century, but only IBM's goofy "century-flag":

```
DCL DD CENTURY       ZND(2,0) DEF(YYYYMMDDHHMMSS) POS(1);
DCL DD CENTURY-FLAG  ZND(1,0) DEF(YYYYMMDDHHMMSS) POS(2);
DCL DD CYYMMDDHHMMSS CHAR(13) DEF(YYYYMMDDHHMMSS) POS(2);

DCL INSPTR .DATE-TO-TIMESTAMP;
ENTRY       DATE-TO-TIMESTAMP INT;
    ADDN        CENTURY, CENTURY-FLAG, 19;
    SUBN        NBR-YEARS, YEAR, 1925;  /* 1st period: 1925, 26, 27, 28 */
    DIVREM      NBR-PERIODS, NBR-YEARS, 4, ADD-YEARS; /* 0   1   2   3 */
    MULT        NBR-DAYS, NBR-PERIODS, 1461;
    MULT        D, ADD-YEARS, 365;
    ADDN(S)     NBR-DAYS, D;
    CPYNV       M, MONTH;
    ADDN(S)     NBR-DAYS, DAYS-ACCUM(M);
    ADDN(S)     NBR-DAYS, DAY;
    CMPNV(B)    ADD-YEARS, 3/NEQ(=+2);   /* leap year fiddling:   */
    CMPNV(B)    MONTH,     2/HI (=+2);:  /* February is one day   */
    SUBN(S)     NBR-DAYS,  1;:           /* longer in the 3rd year */
    MULT        NBR-SECONDS, NBR-DAYS, 86400;
    MULT        S, HOUR, 60; /* S = minutes */
    ADDN(S)     S, MIN;
    MULT(S)     S, 60;       /* S = seconds */
    ADDN(S)     S, SEC;
    ADDN(S)     NBR-SECONDS, S;
    SUBN(S)     NBR-SECONDS, 114955386; /* AUG 23, 1928, 12:03:06pm */
    MULT        TIMESTAMP, NBR-SECONDS, 4096000000;
    DIVREM      BIN-TIME-HI, TIMESTAMP, TWO**32, BIN-TIME-LO;
    B          .DATE-TO-TIMESTAMP;
```

We'll need this conversion because we'll want our API to work in date/time format rather than in 64-bit internal timestamp format.

Must be System State

Because we are trying to modify a system domain object, we need to (e.g. using SST to) make **MIMBRINF** a system state program as outlined in chapter 7. The **MIMAKPTR** program that we call should then also be a system state program (or at least an "inherit state"-program). You could avoid this proliferation of system programs by incorporating (the one-line) **MIMAKPTR** program into **MIMBRINF** and patching the resulting program accordingly. By now, you can do this in stride.

Parameters

For the final revision of **MIBMRINF**, we will modify it to accept two parameters; a control block that specifies the operation to perform and the member to perform it on; and an information block with the type, change dates/times, and the descriptive text. We'll also decide to have a separate operation for changing each of the pieces of the information block because this is often what we need in practice. It is rare that we need to change more than one piece at a time. In any case, the code setting up the parameters is trivial boilerplate code that you can easily change to fit your needs.

```
DCL SPCPTR .PARM-CONTROL   PARM;
DCL SPCPTR .PARM-INFO      PARM;

DCL OL PARAMETERS(.PARM-CONTROL, .PARM-INFO) EXT PARM MIN(2);

DCL DD PARM-CONTROL          CHAR(32)  BAS(.PARM-CONTROL);
    DCL DD PARM-OPERATION    CHAR(1)   DEF(PARM-CONTROL) POS(1);
    /* G - GET INFO FOR MBR      */
    /* S - SET SOURCE DATE/TIME  */
    /* O - SET OBJECT DATE/TIME  */
    /* T - SET MEMBER TYPE       */
    /* D - SET DESCRIPTIVE TEXT  */
    DCL DD PARM-FEEDBACK     CHAR(1)   DEF(PARM-CONTROL) POS(2);
    /* BLANK - OK                */
    /* E     - ERROR             */
    /* O     - OPERATION UNKNOWN */
    DCL DD PARM-LIBRARY      CHAR(10)  DEF(PARM-CONTROL) POS(3);
    /* *LIBL - LIBRARY LIST      */
    DCL DD PARM-FILE         CHAR(10)  DEF(PARM-CONTROL) POS(13);
    DCL DD PARM-MEMBER       CHAR(10)  DEF(PARM-CONTROL) POS(23);
    /* *FILE - MEMBER = FILE     */

DCL DD PARM-INFO             CHAR(86)  BAS(.PARM-INFO);
    DCL DD PARM-TYPE         CHAR(10)  DEF(PARM-INFO) POS(1);
    DCL DD PARM-DATE-SRC     CHAR(13)  DEF(PARM-INFO) POS(11);
    DCL DD PARM-DATE-OBJ     CHAR(13)  DEF(PARM-INFO) POS(24);
    DCL DD PARM-TEXT         CHAR(50)  DEF(PARM-INFO) POS(37);
```

Command CHGMBR and Command Processing Program CHGMBRCL

This time we'll make a simple command and an associated command [processing CL-program to set up the parameters (in particular to allow simple prompting). You can modify these simple programs to suit your needs. This is not rocket science. First the command (**CHGMBR** in file **QCDMSRC**):

```
CMD   PROMPT('Change File Member Information')
PARM  KWD(LIBRARY) TYPE(*CHAR) LEN(10) PROMPT('Library')
PARM  KWD(FILE)    TYPE(*CHAR) LEN(10) PROMPT('File')
PARM  KWD(MEMBER)  TYPE(*CHAR) LEN(10) PROMPT('Member')
PARM  KWD(TYPE)    TYPE(*CHAR) LEN(10) PROMPT('Type')
PARM  KWD(DATESRC) TYPE(*DATE) PROMPT('Source date')
PARM  KWD(TIMESRC) TYPE(*TIME) PROMPT('Source time')
PARM  KWD(DATEOBJ) TYPE(*DATE) PROMPT('Object date')
PARM  KWD(TIMEOBJ) TYPE(*TIME) PROMPT('Object time')
PARM  KWD(TEXT)    TYPE(*CHAR) LEN(50) PROMPT('Descriptive Text')
PARM  KWD(TEST)    TYPE(*CHAR) LEN( 1) PROMPT('Test, Yes/No')
```

Create the command:

```
===> CRTCMD CMD(CHGMBR) PGM(CHGMBRCL) REPLACE(*YES)
```

Here is a typical prompt screen:

```
                    Change File Member Information (CHGMBR)


Type choices, press Enter.

Library  . . . . . . . . . . . . > *LIBL       Character value
File . . . . . . . . . . . . . . > QMISRC      Character value
Member . . . . . . . . . . . . . > MIHELLO     Character value
Type . . . . . . . . . . . . . . > MI          Character value
Source date  . . . . . . . . . .   7/27/2000   Date
Source time  . . . . . . . . . .   11:22:33    Time
Object date  . . . . . . . . . .   9/02/2000   Date
Object time  . . . . . . . . . .   22:33:44    Time
Descriptive Text . . . . . . . . > 'Hello World'                        .
_________.
Test, Yes/No . . . . . . . . . . > Y           Character value
```

The last parameter is for testing. If "Y" the member information is retrieved and the information block is sent as a message to the requesting job's message queue. The full command line for the above prompt was:

```
 ===> CHGMBR LIBRARY(*LIBL) FILE(QMISRC) MEMBER(MIHELLO) TYPE(MI) DATESRC('7/27/
2000') TIMESRC('11:22:33') DATEOBJ('9/02/2000') TIMEOBJ('22:33:44') TEXT('Hello
World') TEST(Y)
```

Only parameters that are given will be changed by the program.

Command Processing Program

Compile the CL-program **CHGMBRCL** in **QCLSRC** that is to serve as command processing program:

```
PGM PARM(&PARMLIB  &PARMFILE &PARMMBR  &PARMTYPE &PARMDSRC &PARMTSRC +
                   &PARMDOBJ &PARMTOBJ &PARMTEXT &PARMTEST)

  DCL &PARMLIB     *CHAR LEN(10)
  DCL &PARMFILE    *CHAR LEN(10)
  DCL &PARMMBR     *CHAR LEN(10)
  DCL &PARMTYPE    *CHAR LEN(10)
  DCL &PARMDSRC    *CHAR LEN(07) /* DATE WHEN SOURCE CHANGED */
  DCL &PARMTSRC    *CHAR LEN(06) /* TIME WHEN SOURCE CHANGED */
  DCL &PARMDOBJ    *CHAR LEN(07) /* DATE WHEN OBJECT CHANGED */
  DCL &PARMTOBJ    *CHAR LEN(06) /* TIME WHEN OBJECT CHANGED */
  DCL &PARMTEXT    *CHAR LEN(50)
  DCL &PARMTEST    *CHAR LEN( 1) /* Y/N */

  DCL &CONTROL     *CHAR LEN(32)
  DCL &QUALNAME    *CHAR LEN(30)
  DCL &INFO        *CHAR LEN(86)

  CHGVAR &QUALNAME VALUE(&PARMLIB *CAT &PARMFILE *CAT &PARMMBR)
  IF (&PARMMBR  *EQ '          ') GOTO DONE

  IF (&PARMTYPE *EQ '*NONE     ') +
       THEN(DO)
          CHGVAR &CONTROL VALUE('T ' *CAT &QUALNAME)
          CHGVAR &INFO    VALUE('          ')
          CALL PGM(MIMBRINF) PARM(&CONTROL &INFO)
       ENDDO
  ELSE +
  IF (&PARMTYPE *NE '          ') +
       THEN(DO)
          CHGVAR &CONTROL VALUE('T ' *CAT &QUALNAME)
          CHGVAR &INFO    VALUE(&PARMTYPE)
          CALL PGM(MIMBRINF) PARM(&CONTROL &INFO)
       ENDDO

  IF (&PARMDSRC *NE '0000000') +
       THEN(DO)
          CHGVAR &CONTROL VALUE('S ' *CAT &QUALNAME)
          CHGVAR &INFO VALUE(&PARMTYPE *CAT &PARMDSRC *CAT &PARMTSRC)
          CALL PGM(MIMBRINF) PARM(&CONTROL &INFO)
       ENDDO

  IF (&PARMDOBJ *NE '0000000') +
       THEN(DO)
          CHGVAR &CONTROL VALUE('O ' *CAT &QUALNAME)
          CHGVAR &INFO VALUE(&PARMTYPE *CAT '             ' +
                                *CAT &PARMDOBJ *CAT &PARMTOBJ)
          CALL PGM(MIMBRINF) PARM(&CONTROL &INFO)
       ENDDO

  IF (&PARMTEXT *EQ '*NONE ') +
       THEN(DO)
          CHGVAR &CONTROL VALUE('D ' *CAT &QUALNAME)
          CHGVAR &INFO    VALUE(&PARMTYPE          +
            *CAT '                          '      +
            *CAT '                                                  ')
          CALL PGM(MIMBRINF) PARM(&CONTROL &INFO)
       ENDDO
  ELSE +
  IF (&PARMTEXT *NE '*NONE ') +
       THEN(DO)
          CHGVAR &CONTROL VALUE('D ' *CAT &QUALNAME)
          CHGVAR &INFO    VALUE(&PARMTYPE          +
            *CAT '                          '      +
            *CAT &PARMTEXT)
          CALL PGM(MIMBRINF) PARM(&CONTROL &INFO)
       ENDDO

DONE:
```

```
  IF (&PARMTEST *EQ 'Y') +
       THEN(DO)
          CHGVAR &CONTROL VALUE('G ' *CAT &QUALNAME)
          CALL PGM(MIMBRINF) PARM(&CONTROL &INFO)
          SNDMSG MSG(&INFO) TOUSR(*REQUESTER)
       ENDDO
ENDPGM
```

When you run the command, with `TEST(Y)`, the test message that is sent looks like this:

```
 Type reply (if required), press Enter.
   From  . . . :   LSVALGAARD     09/06/00   22:31:22
   MI        1000727112233100090222334Hello World
```

The Complete `MIMBRINF` Program

```
DCL SPCPTR .PARM-CONTROL   PARM;
DCL SPCPTR .PARM-INFO      PARM;

DCL OL PARAMETERS(.PARM-CONTROL, .PARM-INFO) EXT PARM MIN(2);

DCL DD PARM-CONTROL          CHAR(32)  BAS(.PARM-CONTROL);
    DCL DD PARM-OPERATION    CHAR(1)   DEF(PARM-CONTROL) POS(1);
    /* G - GET INFO FOR MBR       */
    /* S - SET SOURCE DATE/TIME   */
    /* O - SET OBJECT DATE/TIME   */
    /* T - SET MEMBER TYPE        */
    /* D - SET DESCRIPTIVE TEXT   */
    DCL DD PARM-FEEDBACK     CHAR(1)   DEF(PARM-CONTROL) POS(2);
    /* BLANK - OK                 */
    /* E     - ERROR              */
    /* O     - OPERATION UNKNOWN  */
    DCL DD PARM-LIBRARY      CHAR(10)  DEF(PARM-CONTROL) POS(3);
    /* *LIBL - LIBRARY LIST       */
    DCL DD PARM-FILE         CHAR(10)  DEF(PARM-CONTROL) POS(13);
    DCL DD PARM-MEMBER       CHAR(10)  DEF(PARM-CONTROL) POS(23);
    /* *FILE - MEMBER = FILE      */

DCL DD PARM-INFO             CHAR(86)  BAS(.PARM-INFO);
    DCL DD PARM-TYPE         CHAR(10)  DEF(PARM-INFO) POS(1);
    DCL DD PARM-DATE-SRC     CHAR(13)  DEF(PARM-INFO) POS(11);
    DCL DD PARM-DATE-OBJ     CHAR(13)  DEF(PARM-INFO) POS(24);
    DCL DD PARM-TEXT         CHAR(50)  DEF(PARM-INFO) POS(37);

DCL EXCM EXCEPTION-LIST EXCID(H'0000') BP(ERROR-DETECTED) IGN;

DCL SYSPTR .CONTEXT;
DCL SYSPTR .CURSOR;
DCL SPCPTR .CURSOR-SPACE;
DCL SPC     CURSOR-SPACE BAS(.CURSOR-SPACE);
    DCL DD  CSR-MBR-HEADER BIN(4) DEF(CURSOR-SPACE) POS(5);

DCL SPC     MBR-CURSOR BAS(.POINTER);
    DCL DD  MBR-CHANGE-TIMESTAMP CHAR(8) DEF(MBR-CURSOR) POS(129);

DCL SPCPTR .MBR-HEADER;
DCL SPC MBR-HEADER BAS(.MBR-HEADER);
    DCL SYSPTR .MHDR-PREV-MCB         DIR;
    DCL SYSPTR .MHDR-NEXT-MCB         DIR;
    DCL SYSPTR .MHDR-FILE-CB          DIR;
    DCL SYSPTR .MHDR-SHARE-DIR        DIR;
    DCL SYSPTR .MHDR-DATA-DICT        DIR;

    DCL DD MHDR-STATUS      CHAR(2)   DIR;
    DCL DD *                CHAR(2)   DIR;
    DCL DD MHDR-TEXT        CHAR(50)  DIR;
    DCL DD MHDR-TYPE        CHAR(10)  DIR;
    DCL DD *                CHAR(10)  DIR;

    DCL DD MHDR-CHANGE-DATE CHAR(13)  DIR;
    DCL DD MHDR-CREATE-DATE CHAR(13)  DIR;

    DCL DD MHDR-PREFRD-UNIT CHAR(1)   DIR;
    DCL DD MHDR-ALLOC-TYPE  CHAR(2)   DIR;
    DCL DD MHDR-INIT-RECS   BIN(4)    DIR;
    DCL DD MHDR-RECS-EXTEND BIN(2)    DIR;
    DCL DD MHDR-NBR-EXTENDS BIN(2)    DIR;
    DCL DD MHDR-RECOVER-OPT CHAR(1)   DIR;
```

```
    DCL DD MHDR-SAVE-DATE   CHAR(13)  DIR;
    DCL DD MHDR-RSTR-DATE   CHAR(13)  DIR;
    DCL DD MHDR-%-DLT-ALLOW CHAR(1)   DIR;
    DCL DD MHDR.USER-AREA   BIN(4)    DIR;
    DCL DD MHDR-OLD-S-DATE  CHAR(13)  DIR;
    DCL DD MHDR-OLD-R-DATE  CHAR(13)  DIR;
    DCL DD MHDR...........  CHAR(1)   DIR;

DCL SPCPTR .ARG1 INIT(POINTER);
DCL DD  POINTER CHAR(16) BDRY(16);
    DCL PTR .POINTER           DEF(POINTER) POS( 1);

    DCL DD PTR-TYPE    CHAR(8) DEF(POINTER) POS( 1);
    DCL DD PTR-SEGMENT CHAR(5) DEF(POINTER) POS( 9);
    DCL DD PTR-OFFSET  CHAR(3) DEF(POINTER) POS(14);

DCL OL      MIMAKPTR (.ARG1) ARG;
DCL SYSPTR .MIMAKPTR;

DCL DD RESOLVE CHAR(34);
    DCL DD RESOLVE-TYPE CHAR( 2) DEF(RESOLVE) POS( 1) INIT(X'0000');
    DCL DD RESOLVE-NAME CHAR(30) DEF(RESOLVE) POS( 3);
    DCL DD RESOLVE-AUTH CHAR( 2) DEF(RESOLVE) POS(33) INIT(X'0000');

/*************************************************************/

ENTRY * (PARAMETERS) EXT;
    CPYBLA       PARM-FEEDBACK, " ";
    CMPBLA(B)    RESOLVE-TYPE, X'0000'/NEQ(TEST-LIBRARY);
RESOLVE-TO-MAKE-POINTER-PGM:
    CPYBLA       RESOLVE-TYPE, X'0201';
    CPYBLAP      RESOLVE-NAME, "MIMAKPTR", " ";
    RSLVSP      .MIMAKPTR, RESOLVE, *, *;

TEST-LIBRARY:
    CMPBLAP(B)   PARM-LIBRARY, "*LIBL", " "/NEQ(GET-CONTEXT);
    CPYBWP      .CONTEXT, *; /* NULL */
    B            TEST-MEMBER;

GET-CONTEXT:
    CPYBLA       RESOLVE-TYPE, X'0401';
    CPYBLAP      RESOLVE-NAME, PARM-LIBRARY, " ";
    RSLVSP      .CONTEXT, RESOLVE, *, *;

TEST-MEMBER:
    CPYBLAP      RESOLVE-NAME( 1:30), PARM-FILE, " ";
    CPYBLA       RESOLVE-NAME(11:10), PARM-FILE;
    CMPBLAP(B)   PARM-MEMBER, "*FILE", " "/EQ(=+2);
    CPYBLA       RESOLVE-NAME(11:10), PARM-MEMBER;:

RESOLVE-TO-CURSOR:
    CPYBLA       RESOLVE-TYPE, X'0D50';
    RSLVSP      .CURSOR, RESOLVE, .CONTEXT, *;
    SETSPPFP    .CURSOR-SPACE, .CURSOR;
    ADDSPP      .MBR-HEADER, .CURSOR-SPACE, CSR-MBR-HEADER;

TEST-OPERATION:
    CMPBLA(B)    PARM-OPERATION, "S"/EQ(SET-SOURCE-INFO);
    CMPBLA(B)    PARM-OPERATION, "T"/EQ(SET-MEMBER-TYPE);
    CMPBLA(B)    PARM-OPERATION, "D"/EQ(SET-DESCRIPTIVE-TEXT);

GET-MBR-OBJECT:
    CPYBWP      .POINTER, .CURSOR;
    CPYBREP      PTR-OFFSET, X'00';
    CALLX       .MIMAKPTR, MIMAKPTR, *;

    CMPBLA(B)    PARM-OPERATION, "O"/EQ(SET-OBJECT-INFO);
    CMPBLA(B)    PARM-OPERATION, "G"/EQ(GET-INFO-FOR-MBR);
    B            OPERATION-ERROR;

GET-INFO-FOR-MBR:
    CPYBLA       PARM-TYPE, MHDR-TYPE;
    CPYBLA       PARM-TEXT, MHDR-TEXT;
    CPYBLA       BIN-TIMESTAMP, MBR-CHANGE-TIMESTAMP;
    CALLI        TIMESTAMP-TO-DATE, *, .TIMESTAMP-TO-DATE;
    CPYBLA       PARM-DATE-OBJ, CYYMMDDHHMMSS;
    CPYBLA       PARM-DATE-SRC, MHDR-CHANGE-DATE;
    B            RETURN;

SET-SOURCE-INFO:
```

```
    CPYBLA       MHDR-CHANGE-DATE, PARM-DATE-SRC;
    B            RETURN;

SET-OBJECT-INFO:
    CPYBLA       CYYMMDDHHMMSS, PARM-DATE-OBJ;
    CALLI        DATE-TO-TIMESTAMP, *, .DATE-TO-TIMESTAMP;
    CPYBLA       MBR-CHANGE-TIMESTAMP, BIN-TIMESTAMP;
    B            RETURN;

SET-MEMBER-TYPE:
    CPYBLA       MHDR-TYPE, PARM-TYPE;
    B            RETURN;

SET-DESCRIPTIVE-TEXT:
    CPYBLA       MHDR-TEXT, PARM-TEXT;
    B            RETURN;

OPERATION-ERROR:
    CPYBLA       PARM-FEEDBACK, "O";
RETURN:
    RTX          *;

ERROR-DETECTED:
    CPYBLA       PARM-FEEDBACK, "E";
    RTX          *;
```

Internal Sorting, *Combsort*

Sorting Internal Data

In this chapter we shall use MI to develop a very fast sorting program for internal data held in an array (a *table*) and show how to get the CPU time the processor spends sorting tables of different sizes. In doing this we'll have occasion to use floating point variables computing logarithms. We start with a description of the simple, well-known Bubble sort algorithm and show the algorithm first in COBOL.

Bubble Sort

As the name suggests, bubble sort moves items up a table like bubbles in a tube. The algorithm can be explained as follows: pass over the data, comparing and exchanging items so that the largest item ends up at the end of the table. Repeat for the remaining items until the table is sorted, that is, no exchanges were made during the latest pass.

The following COBOL code performs a bubble sort:

```
01  TABLE-TO-SORT.
    02  TABLE-SIZE              PIC S9(5)  COMP.
    02  TABLE-MAX               PIC S9(5)  COMP  VALUE +1000.
    02  TABLE-ITEM                         OCCURS 1000 TIMES.
        03  TABLE-KEY           PIC X(9).
        03  TABLE-DATA          PIC X(11).

01  VARIOUS-INDICES.
    02  ITEM-NBR                PIC S9(5)  COMP.
    02  SWAP-NBR                PIC S9(5)  COMP.
    02  JUMP-SIZE               PIC S9(5)  COMP.
    02  UPPER-LIMIT             PIC S9(5)  COMP.

01  VARIOUS-VALUES.
    02  SWAP-ITEM               PIC X(20).
    02  SWAP-INDICATOR          PIC X(1).
        88  NO-MORE-SWAPS           VALUE IS SPACE.

BUBBLE-SORT-THE-ARRAY.
    MOVE  nnnn  TO TABLE-SIZE
    MOVE "SWAP" TO SWAP-INDICATOR
    MOVE   1    TO JUMP-SIZE
    PERFORM BUBBLE-SORT
      UNTIL NO-MORE-SWAPS
    .

BUBBLE-SORT.
    MOVE SPACE TO SWAP-INDICATOR
    COMPUTE UPPER-LIMIT = TABLE-SIZE - JUMP-SIZE

    PERFORM COMPARE-AND-SWAP-KEYS
      VARYING ITEM-NBR FROM 1 BY 1
        UNTIL ITEM-NBR > UPPER-LIMIT
    .

COMPARE-AND-SWAP-KEYS.
    COMPUTE SWAP-NBR = ITEM-NBR + JUMP-SIZE
    IF TABLE-KEY (ITEM-NBR) > TABLE-KEY (SWAP-NBR)
        MOVE TABLE-ITEM (ITEM-NBR) TO SWAP-ITEM
        MOVE TABLE-ITEM (SWAP-NBR) TO TABLE-ITEM (ITEM-NBR)
        MOVE  SWAP-ITEM TO TABLE-ITEM (SWAP-NBR)
        MOVE "SWAP" TO SWAP-INDICATOR
    .
```

The code sorts a table of a given size assuming that is have already been initialized with values. Even if COBOL is not your favorite language, the code should be easy enough to follow. Study the above code so you understand how it works. It has long been known that bubble sort is the *worst* sorting algorithm and that one should never use it for anything more that about 20 values. The best general-purpose sorting algorithm was known to be quicksort. Bubble sort's only redeeming feature is its simplicity. Sorting is a problem that was solved years ago.

Combsort

Just as we thought that the last word had been said about sorting, a breakthrough comes along and spoils everything. In the April 1991 issue of BYTE magazine, Stephen Lacey and Richard Box show that a simple modification to bubble sort makes it a fast and efficient sort method on par with heapsort and quicksort.

In a bubble sort, each item is compared to the next; if the two are out of order, they are swapped. This method is slow because it is susceptible to the appearance of what Box and Lacey call *turtles*. A turtle is a relatively low value located near the end of the table. During a bubble sort, this element moves only one position for each pass, so a single turtle can cause maximal slowing. Almost every long table of items contains a turtle.

Their simple modification of bubble sort, which they call 'combsort', eliminates turtles quickly by allowing the distance between compared items to be greater than one. This distance - the `JUMP-SIZE` - is initially set to the `TABLE-SIZE`. Before each pass, the `JUMP-SIZE` is divided by 1.3 (the *shrink factor*). If this causes it to become less than 1, it is simply set to 1, collapsing combsort into bubble sort. An exchange of items moves items by `JUMP-SIZE` positions rather than only one position, causing turtles to jump rather than crawl. As with any sort method where the displacement of an element can be larger than one position, combsort is not stable - like elements do not keep their relative positions. This is rarely a problem in practice and could, if necessary, be fixed by adding a sequence number to the key.

Successively shrinking the `JUMP-SIZE` is analogous to combing long, tangled hair - stroking first with your fingers alone, then with a pick comb that has widely spaced teeth, followed by finer combs with progressively closer teeth - hence the name ***combsort***. Lacey and Box came up with a shrink factor of 1.3 empirically by testing combsort on over 200,000 random tables. There is at present no theoretical justification for this particular value; it just works...

Here is then the magic code. It is clearly correct, as it (unless the table is empty) ends with `JUMP-SIZE = 1` (ensured by the `+3') and therefore degenerates into bubble sort:

```
COMBSORT-THE-ARRAY.
    MOVE TABLE-SIZE TO JUMP-SIZE
    PERFORM COMBSORT
      UNTIL NO-MORE-SWAPS
        AND JUMP-SIZE NOT > 1
    .

COMBSORT.
    COMPUTE JUMP-SIZE = (10 * JUMP-SIZE + 3) / 13
    COMPUTE UPPER-LIMIT = TABLE-SIZE - JUMP-SIZE
    MOVE SPACE TO SWAP-INDICATOR
    PERFORM COMPARE-AND-SWAP-KEYS
      VARYING ITEM-NBR FROM 1 BY 1
        UNTIL ITEM-NBR > UPPER-LIMIT
    .
```

The careful termination test (`JUMP-SIZE NOT > 1`) also caters for the case where the table is empty.

MI-Version of Combsort

We'll make the MI-version of Combsort, **`MICMBSRT`**, a separate program that we can call from our test program. This nicely separates the sorting algorithm from the test scaffolding. For simplicity, we make the key the same as the table element value. Then we generalize a bit and make the code able to handle both ascending keys and descending keys. The first parameter is a template, that gives the number of elements to be sorted and the sort direction:

```
DCL SPCPTR .PARM1 PARM;
DCL DD PARM1 CHAR(5) BAS(.PARM1);
    DCL DD PARM-NBR-ELEMENTS PKD(7,0) DEF(PARM1) POS(1);
    DCL DD PARM-DIRECTION    CHAR(1)  DEF(PARM1) POS(5); /* A or D */

DCL SPCPTR .PARM2 PARM;
DCL DD ELEMENT(1) CHAR(10) BAS(.PARM2);
DCL DD KEY    (1) CHAR(10) BAS(.PARM2); /* overlays ELEMENT */

DCL OL PARMS(.PARM1, .PARM2) EXT PARM MIN(2);
```

We do not want to know how large the table is, so it is declared here to contain only one element. We'll tell the compiler to omit subscript checking. This is done with the "Override Program Attributes"-instruction (**OVRPGATR**), which really is executed at compile-time and is in effect until overridden by another **OVRPGATR**.

Override Program Attribute

The format is: `OVRPGATR` *`Attribute ID, Attribute modifier`*

Attribute ID	Description	Modifier
1	Array constrainment	1 = Constrain array references
		2 = Do not constrain array references
		3 = Fully unconstrain array references
		4 = Resume attribute given in template
2	String constrainment	1 = Constrain string references
		2 = Do not constrain string references
		3 = Resume attribute given in template
3	Suppress binary size error	1 = Suppress binary size exceptions
		2 = Do not suppress binary size exceptions
		3 = Resume attribute given in template
4	Suppress decimal data error	1 = Suppress decimal data exceptions
		2 = Do not suppress decimal data exceptions
		3 = Resume attribute given in template
5	CPYBWP alignment	1 = Require like alignment
		2 = Do not require like alignment
6	CMPSPAD null pointer	1 = Signal pointer does not exist exception
		2 = Do not signal pointer does not exist exception

```
ENTRY * (PARMS) EXT;
    OVRPGATR   1, 2; /* Don't Constrain Array Refs */
```

The effect of the **OVRPGATR** in this case allows us to access the array as an *unbounded* array, this allows us to declare the variable as only having one element, `ELEMENT(1)`, but in our code we can use any positive value as the index to access any element with in the array. This is a very useful facility, however as the compiler is not now checking whether the index is within the bounds of the array, it can be quite easy to supply an index whose value is so large, that we try to access data outside the storage space for the array. Don't worry though, OS/400 has a polite way of telling you, with a MCH error.

The Sort Double Loop

Declare the various local variables:

```
DCL DD SWAP-FLAG  CHAR(1);
DCL DD JUMP-SIZE   BIN(4);
DCL DD SWEEP-END   BIN(4);
DCL DD ITEM-NBR    BIN(4);
DCL DD COMP-NBR    BIN(4);
```

Start the loops:

```
    CPYNV      JUMP-SIZE, PARM-NBR-ELEMENTS;
SORT-JUMP:                                    <-----------------------.
    CMPNV(B)   JUMP-SIZE,  1 /HI(SORT-SWEEP);                          |
    CMPBLA(B)  SWAP-FLAG, "S"/NEQ(RETURN);                             |
SORT-SWEEP:                                                            |
    MULT(S)    JUMP-SIZE, 10;                                          |
    ADDN(S)    JUMP-SIZE,  3;                                          |
    DIV(S)     JUMP-SIZE, 13; /* JUMP-SIZE = (10 * JUMP-SIZE + 3)/13 */  |
    SUBN       SWEEP-END, PARM-NBR-ELEMENTS, JUMP-SIZE;                |
    CPYBLA     SWAP-FLAG, " ";                                         |
    CPYNV(B)   ITEM-NBR, 0/ZER(SORT-COMPARE);                          |
                                                                       |
SORT-SWAP:                                      <-------------.        |
    EXCHBY     ELEMENT(ITEM-NBR), ELEMENT(COMP-NBR);            |        |
    CPYBLA     SWAP-FLAG, "S";                                  |        |
SORT-COMPARE:                                   <-------------|        |
    ADDN(S)    ITEM-NBR, 1;                                     |        |
    CMPNV(B)   ITEM-NBR, SWEEP-END/HI(SORT-JUMP);  ------------------------'
    ADDN       COMP-NBR, ITEM-NBR, JUMP-SIZE;                   |
    CMPBLA(B)  PARM-DIRECTION, "D"/EQ(DESCENDING-SORT-COMPARE);  |
                                                                |
ASCENDING-SORT-COMPARE:                                         |
```

```
    CMPBLA(B)  KEY(ITEM-NBR), KEY(COMP-NBR)/                    |
               HI(SORT-SWAP), NHI(SORT-COMPARE);   --------------'

DESCENDING-SORT-COMPARE:
    CMPBLA(B)  KEY(ITEM-NBR), KEY(COMP-NBR)/
               LO(SORT-SWAP), NLO(SORT-COMPARE);
RETURN:
    RTX        *;
```

Note the handy swap instruction: **EXCHBY** (Exchange Bytes). Note also the use of more than one branch extender:

```
    CMPBLA(B)  KEY(ITEM-NBR), KEY(COMP-NBR)/      /* if condition is 'HI' goto SORT-SWAP
*/
               HI(SORT-SWAP), NHI(SORT-COMPARE);  /* else goto SORT-COMPARE
*/
```

Since the second condition is the negation of the first, a branch is always taken.

Complete Combsort Code

```
DCL SPCPTR .PARM1 PARM;
DCL DD PARM1 CHAR(5) BAS(.PARM1);
    DCL DD PARM-NBR-ELEMENTS PKD(7,0) DEF(PARM1) POS(1);
    DCL DD PARM-DIRECTION    CHAR(1)  DEF(PARM1) POS(5); /* A OR D */

DCL SPCPTR .PARM2 PARM;
DCL DD ELEMENT(1) CHAR(10) BAS(.PARM2);
DCL DD KEY    (1) CHAR(10) BAS(.PARM2);

DCL OL PARMS(.PARM1, .PARM2) EXT PARM MIN(2);

DCL DD SWAP-FLAG  CHAR(1);
DCL DD JUMP-SIZE   BIN(4);
DCL DD SWEEP-END   BIN(4);
DCL DD ITEM-NBR    BIN(4);
DCL DD COMP-NBR    BIN(4);

ENTRY * (PARMS) EXT;
    OVRPGATR   1, 2; /* DON'T CONSTRAIN ARRAY REFS */
    CPYNV      JUMP-SIZE, PARM-NBR-ELEMENTS;
SORT-JUMP:
    CMPNV(B)   JUMP-SIZE,  1 /HI(SORT-SWEEP);
    CMPBLA(B)  SWAP-FLAG, "S"/NEQ(RETURN);
SORT-SWEEP:
    MULT(S)    JUMP-SIZE, 10;
    ADDN(S)    JUMP-SIZE,  3;
    DIV(S)     JUMP-SIZE, 13;
    SUBN       SWEEP-END, PARM-NBR-ELEMENTS, JUMP-SIZE;
    CPYBLA     SWAP-FLAG, " ";
    CPYNV(B)   ITEM-NBR, 0/ZER(SORT-COMPARE);

SORT-SWAP:
    EXCHBY     ELEMENT(ITEM-NBR), ELEMENT(COMP-NBR);
    CPYBLA     SWAP-FLAG, "S";
SORT-COMPARE:
    ADDN(S)    ITEM-NBR, 1;
    CMPNV(B)   ITEM-NBR, SWEEP-END/HI(SORT-JUMP);
    ADDN       COMP-NBR, ITEM-NBR, JUMP-SIZE;
    CMPBLA(B)  PARM-DIRECTION, "D"/EQ(DESCENDING-SORT-COMPARE);

ASCENDING-SORT-COMPARE:
    CMPBLA(B)  KEY(ITEM-NBR), KEY(COMP-NBR)/
               HI(SORT-SWAP), NHI(SORT-COMPARE);
DESCENDING-SORT-COMPARE:
    CMPBLA(B)  KEY(ITEM-NBR), KEY(COMP-NBR)/
               LO(SORT-SWAP), NLO(SORT-COMPARE);

RETURN:
    RTX        *;
```

Using Instruction Pointers

The test of the sort direction in the inner loop can be avoided using an instruction pointer with the "Set Instruction Pointer"-instruction, **SETIP** *pointer*, *label*:

```
DCL INSPTR KEY-COMPARE;
```

```
ENTRY * (PARMS) EXT;
    OVRPGATR   1, 2; /* DON'T CONSTRAIN ARRAY REFS */

    SETIP      KEY-COMPARE, ASCENDING-SORT-COMPARE;
    CMPBLA(B)  PARM-DIRECTION, "D"/NEQ(=+2);
    SETIP      KEY-COMPARE, DESCENDING-SORT-COMPARE;:

…
SORT-COMPARE:
    ADDN(S)    ITEM-NBR, 1;
    CMPNV(B)   ITEM-NBR, SWEEP-END/HI(SORT-JUMP);
    ADDN(B)    COMP-NBR, ITEM-NBR, JUMP-SIZE/POS(KEY-COMPARE);

ASCENDING-SORT-COMPARE:
    CMPBLA(B)  KEY(ITEM-NBR), KEY(COMP-NBR)/
               HI(SORT-SWAP), NHI(SORT-COMPARE);
DESCENDING-SORT-COMPARE:
    CMPBLA(B)  KEY(ITEM-NBR), KEY(COMP-NBR)/
               LO(SORT-SWAP), NLO(SORT-COMPARE);
```

Testing Combsort

The test program (**MITSTCBM**) shall run through a range of table sizes, say from 50,000 items to 1,000,000 items. For each choice of table size, N, we initialize the table with random numbers (giving us a chance of showing how to do that too), then get the processor time spent before and after calling the Combsort program, **MICMBSRT**. Finally, we calculate the time difference and compare it to the theoretical optimal value kN log_2 N, where k is a constant (the time to decide if a given item is in place and move it if not).

First the interface to Combsort:

```
DCL SPCPTR .CONTROL INIT(CONTROL);
DCL DD      CONTROL CHAR(5);
    DCL DD  CTRL-NBR-ELEMENTS PKD(7,0) DEF(CONTROL) POS(1);
    DCL DD  CTRL-DIRECTION    CHAR(1)  DEF(CONTROL) POS(5);

DCL SPCPTR .TABLE INIT(TABLE);
DCL DD      TABLE(1000000) ZND(10,0); /* MAX 16MB */

DCL SYSPTR .COMBSORT INIT("MICMBSRT", TYPE(PGM));
DCL OL COMBSORT(.CONTROL, .TABLE) ARG;
```

Generating Random Numbers

We use the linear congruential method to generate pseudo-random numbers. The following formula generates evenly distributed integers between 0 and 2,099,862 and suits us fine (seed *Random* with a suitable number first):

$$Random = (1005973 * Random + 443771) \bmod 2099863$$

```
DCL DD RND-RESULT           PKD( 7,0); /* Also SEED Value */
DCL DD RND-PRODUCT          PKD(15,0);
DCL DD RND-MODULUS          PKD( 7,0) INIT(P'+2099863');
DCL DD RND-MULTIPLIER       PKD( 7,0) INIT(P'+1005973');
DCL DD RND-INCREMENT        PKD( 7,0) INIT(P'+443771');

DCL DD NBR   BIN(4);
DCL DD N     BIN(4); /* Number of elements in the table */

    CPYNV       RND-RESULT, 314159; /* SEED Value */
    CPYNV       NBR, 0;
NEXT-ELEMENT:
    MULT        RND-PRODUCT, RND-MULTIPLIER, RND-RESULT;
    ADDN(S)     RND-PRODUCT, RND-INCREMENT;
    REM         RND-RESULT , RND-PRODUCT, RND-MODULUS; /* REMainder */

    ADDN(S)     NBR, 1;
    CPYNV       TABLE(NBR), RND-RESULT;
    CMPNV(B)    NBR, N/LO(NEXT-ELEMENT);
```

Measuring Processor Time Spent

The "Materialize Process Attributes"-instruction (**MATPRATR**) can return all kinds of data about a running process. If called with a null 2nd operand it returns information about the current process (the one your program is running in): The 3rd operand selects what information to materialize; x'**21**' selects the processor time spent:

```
DCL SPCPTR .MAT-PROC-ATTRS INIT(MAT-PROC-ATTRS);
DCL DD      MAT-PROC-ATTRS CHAR(22);
    DCL DD  MAT-BYTES-PROVIDED   BIN(4) DEF(MAT-PROC-ATTRS) POS( 1);
    DCL DD  MAT-BYTES-AVAILABLE  BIN(4) DEF(MAT-PROC-ATTRS) POS( 5);
    DCL DD  MAT-TOTAL-BYTES-USED BIN(4) DEF(MAT-PROC-ATTRS) POS( 9);
    DCL DD  MAT-CPU-TIME-USED   CHAR(8) DEF(MAT-PROC-ATTRS) POS(13);
    DCL DD  MAT-NBR-OF-LOCKS     BIN(2) DEF(MAT-PROC-ATTRS) POS(21);

    CPYNV       MAT-BYTES-PROVIDED, 20;    /* don't care about the LOCKS */
    MATPRATR   .MAT-PROC-ATTRS, *, X'21';  /* get CPU-time used          */
```

Store the 8-character timestamp-format CPU-TIME-USED in BEFORE and AFTER variables:

```
DCL DD CPU-TIMES CHAR(24) BDRY(8);
    DCL DD CPU-BEFORE       CHAR(8) DEF(CPU-TIMES) POS( 1);
    DCL DD CPU-AFTER        CHAR(8) DEF(CPU-TIMES) POS( 9);
    DCL DD CPU-DIFFERENCE   CHAR(8) DEF(CPU-TIMES) POS(17);

    CPYBLA      CPU-BEFORE, MAT-CPU-TIME-USED;
```

Calling Combsort

```
SORT:
    CPYNV       CTRL-NBR-ELEMENTS, N;
    CPYBLA      CTRL-DIRECTION, "ASCENDING"; /* only stores the 'A' */
    BRK "1";

    CPYNV       MAT-BYTES-PROVIDED, 20;
    MATPRATR   .MAT-PROC-ATTRS, *, X'21';
    CPYBLA      CPU-BEFORE, MAT-CPU-TIME-USED;

    CALLX      .COMBSORT, COMBSORT, *;

    MATPRATR   .MAT-PROC-ATTRS, *, X'21';
    CPYBLA      CPU-AFTER , MAT-CPU-TIME-USED;
    BRK "2";
```

The two break points are placed such that you can observe the table before and after sorting it. We should also test if the table *is* actually sorted by comparing each item with the previous one:

```
DCL DD PREV  BIN(4);

TEST-IF-SORTED:
    CPYNV       NBR, N;
COMPARE-WITH-PREVIOUS:
    SUBN(B)     PREV, NBR, 1/NPOS(=+3);
    CMPBLA(B)   TABLE(NBR), TABLE(PREV)/LO(NOT-SORTED); /* Notify if bad */
    SUBN(SB)    NBR, 1/POS(COMPARE-WITH-PREVIOUS);:
```

Computing N log_2 N

The machine has a built-in function to calculate the natural logarithm (to base *e*) of a number. The logarithm to base 2 is readily found from: $ln\,(x) = ln\,(2^{log_2 x}) = log_2(x)\; ln\,(2)$, *i.e.*: $log_2(x) = ln(x)/ln(2)$. All these calculations must be done in floating-point:

```
DCL DD LN-N       FLT(8);
DCL DD LOG2-N     FLT(8);
DCL DD LN-2       FLT(8);
DCL DD FLT-N      FLT(8);
DCL DD N*LOG2-N   FLT(8);
DCL DD RESULT     ZND(10,0);

COMPUTE-N*LOG2-N:
    CPYNV       RESULT, N;
    CPYBLAP     MSG-TEXT, RESULT, " "; /* MSG-TEXT = N     */
```

```
    CPYNV       FLT-N, N;
    CMF1        LN-N, X'0011', FLT-N;
    CMF1        LN-2, X'0011', E'2';
    DIV         LOG2-N, LN-N, LN-2;
    MULT        N*LOG2-N, N, LOG2-N;
    CPYNV(R)    RESULT, N*LOG2-N;
    CPYBLA      MSG-TEXT(13:10), RESULT; /* MSG-TEXT append N*log2 N */
```

Compute Mathematical Function with 1 Argument

The **CMF1**-instruction has this format:

```
    CMF1        Result, 2-Character Function Code, Input Argument
```

The *Result* and the *Argument* must both be *floating-point* numbers. This is a (sad) departure from the polymorphic nature of the other instructions that work on numeric operands. So, we need to convert N ourselves:

```
    CPYNV       FLT-N, N;  /* converting to floating point */
    CMF1        LN-N, X'0011', FLT-N;
```

Function x'**0011**' is the Natural Logarithm. Other functions are:

x'0001'	Sine	$-1 \le \sin(x) \le +1$
x'0002'	Arc sine	$-\pi/2 \le \arcsin(x) \le +\pi/2$
x'0003'	Cosine	$-1 \le \cos(x) \le +1$
x'0004'	Arc cosine	$0 \le \arccos(x) \le \pi$
x'0005'	Tangent	$-\infty \le \tan(x) \le +\infty$
x'0006'	Arc tangent	$-\pi/2 \le \arctan(x) \le +\pi/2$
x'0007'	Cotangent	$-\infty \le \cot(x) \le +\infty$
x'0010'	Exponential function	$0 \le \exp(x) \le +\infty$
x'0011'	Natural logarithm	$-\infty \le \ln(x) \le +\infty$
x'0012'	Sine hyperbolic	$-\infty \le \sinh(x) \le +\infty$
x'0013'	Cosine hyperbolic	$+1 \le \cosh(x) \le +\infty$
x'0014'	Tangent hyperbolic	$-1 \le \tanh(x) \le +1$
x'0015'	Arc tangent hyperbolic	$-\infty \le \text{arctanh}(x) \le +\infty$
x'0020'	Square root	$0 \le \text{sqrt}(x) \le +\infty$

Calculating the natural logarithm of 2, we can use the long floating-point literal E'2':

```
    CMF1        LN-2, X'0011', E'2';
```

Computing Time Differences

Working with 64-bit long integers is complicated by there not being a `BIN(8)` variable type. We'll use the same technique as in chapter 2 of going through intermediate packed numbers:

```
DCL DD CPU-TIMES CHAR(24) BDRY(8);
    DCL DD CPU-BEFORE       CHAR(8) DEF(CPU-TIMES) POS( 1);
    DCL DD CPU-AFTER        CHAR(8) DEF(CPU-TIMES) POS( 9);
    DCL DD CPU-DIFFERENCE   CHAR(8) DEF(CPU-TIMES) POS(17);

DCL DD CPU-DIFF-HI   BIN(4) UNSGND DEF(CPU-DIFFERENCE) POS(1);
DCL DD CPU-DIFF-LO   BIN(4) UNSGND DEF(CPU-DIFFERENCE) POS(5);

DCL DD TIMESTAMP     PKD(21,0); /* Can hold 64-bit unsigned integer */
DCL DD TIMESTAMP-HI  PKD(11,0); /* Can hold 32-bit unsigned integer */
DCL DD TIMESTAMP-LO  PKD(11,0); /* Can hold 32-bit unsigned integer */
DCL DD TWO**32       PKD(11,0) INIT(P'4294967296');  /* unsigned 2^32 */
```

The 8-character time difference can be computed with the "Subtract Logical Character"-instruction (**SUBLC**):

```
COMPUTE-TIME-USED:
    SUBLC       CPU-DIFFERENCE, CPU-AFTER, CPU-BEFORE; /* diff = after - before */
    CPYNV       TIMESTAMP-HI, CPU-DIFF-HI;             /* copy UNSIGNED values  */
    CPYNV       TIMESTAMP-LO, CPU-DIFF-LO;
    MULT        TIMESTAMP, TIMESTAMP-HI, TWO**32;
    ADDN(S)     TIMESTAMP, TIMESTAMP-LO; /*  complete 64-bit timestamp as PKD   */
```

To convert the timestamp to microseconds, we divide by 4096:

```
    DIV         RESULT, TIMESTAMP, 4096; /* MICROSECONDS */
    CPYBLA      MSG-TEXT(25:10), RESULT; /* Append to message */
```

If the relationship is linear the time taken to sort N items divided by N log_2 N should be the constant (k) slope of the line we get by plotting the two quantities against each other. We compute the slope to four decimal places:

```
DCL DD SLOPE     ZND(10,4);

COMPUTE-SLOPE:
    DIV         SLOPE, RESULT, N*LOG2-N;
    CPYBLA      MSG-TEXT(37:10), SLOPE; /* Append to message */
```

Simple Numeric Editing

The message that we have built so far displays each number as a raw 10-digit zoned value complete with leading zeroes, such as: "0000050000 0000780482 0005101264 0000065360". In order to make the result a little more pleasing to the eye, let's remove the leading zeroes. We set up an 11-character substring into the message, identified by the index value `START`. We then let `START` run through the values 37, 25, 13, and 1 (step of 12) and count the leading zeroes, before replacing them with blanks. No rocket science here. How do we count leading zeroes? The "Verify"-instruction (**`VERIFY`**) comes in handy:

```
    VERIFY      Where, Source, Class
```

Each character of the *source* operand is checked to verify that it is among the characters in the *class* operand. If a match exists, the next character is checked. When a mismatch is found its character position in the source operand is returned in the *where* operand. So, if the instruction `VERIFY WHERE, "00001234", "01"` would return with `WHERE` = 6. If no mismatch is found, the *where* operand is set to zero. `VERIFY WHERE, "00001234", "0"` would return `WHERE` = 5. The number of leading zeroes is then one less. The code below does the trick. There are some careful additional tests to cater for various end-conditions (no leading zeroes, all zeroes…):

```
DCL DD WHERE BIN(4);
DCL DD START BIN(4);

    CPYNV       START, 37;
EDIT-RESULT:
    VERIFY      WHERE, MSG-TEXT(START:11), "0";
    SUBN(SB)    WHERE, 1/NPOS(=+2);
    CPYBLAP     MSG-TEXT(START:WHERE), " ", " ";:
    SUBN(SB)    START, 12/POS(EDIT-RESULT);
```

I would have preferred `CPYBREP MSG-TEXT(START:WHERE), " "` for replacing the leading zeroes, but unfortunately `CPYBREP` does not support substrings ☹.

Finally, show the result and continue the loop until N is not lower than the 1,000,000 we had selected as our maximum table size:

```
SHOW-RESULT:
    CALLI       SHOW-MESSAGE, *, .SHOW-MESSAGE;
    CMPNV(B)    N, 1000000/LO(NEXT-SIZE);
    RTX         *;

NOT-SORTED:
    CPYBLAP     MSG-TEXT, "Not Sorted!", " ";
    CALLI       SHOW-MESSAGE, *, .SHOW-MESSAGE;
    RTX         *;

%INCLUDE SHOWMSG
```

The Complete MITSTCMB Test Program

Here is the complete **`MITSTCMB`** test program:

```
DCL SPCPTR .CONTROL INIT(CONTROL);
DCL DD      CONTROL CHAR(5);
    DCL DD  CTRL-NBR-ELEMENTS PKD(7,0) DEF(CONTROL) POS(1);
    DCL DD  CTRL-DIRECTION    CHAR(1)  DEF(CONTROL) POS(5);
```

```
DCL SPCPTR .TABLE INIT(TABLE);
DCL DD      TABLE(1000000) ZND(10,0); /* MAX 16MB */

DCL SYSPTR .COMBSORT INIT("MICMBSRT", TYPE(PGM));
DCL OL COMBSORT(.CONTROL, .TABLE) ARG;

DCL DD PREV  BIN(4);
DCL DD NBR   BIN(4);
DCL DD N     BIN(4);
DCL DD WHERE BIN(4);
DCL DD START BIN(4);

DCL DD LN-N      FLT(8);
DCL DD LOG2-N    FLT(8);
DCL DD LN-2      FLT(8);
DCL DD FLT-N     FLT(8);
DCL DD N*LOG2-N  FLT(8);
DCL DD RESULT    ZND(10,0);
DCL DD SLOPE     ZND(10,4);

DCL DD RND-RESULT           PKD( 7,0); /* ALSO SEED VALUE */
DCL DD RND-PRODUCT          PKD(15,0);
DCL DD RND-MODULUS          PKD( 7,0) INIT(P'+2099863');
DCL DD RND-MULTIPLIER       PKD( 7,0) INIT(P'+1005973');
DCL DD RND-INCREMENT        PKD( 7,0) INIT(P'+443771');

DCL DD CPU-TIMES CHAR(24) BDRY(8);
    DCL DD CPU-BEFORE       CHAR(8) DEF(CPU-TIMES) POS( 1);
    DCL DD CPU-AFTER        CHAR(8) DEF(CPU-TIMES) POS( 9);
    DCL DD CPU-DIFFERENCE   CHAR(8) DEF(CPU-TIMES) POS(17);

DCL DD CPU-DIFF-HI   BIN(4) UNSGND DEF(CPU-DIFFERENCE) POS(1);
DCL DD CPU-DIFF-LO   BIN(4) UNSGND DEF(CPU-DIFFERENCE) POS(5);

DCL DD TIMESTAMP     PKD(21,0);
DCL DD TIMESTAMP-HI  PKD(11,0);
DCL DD TIMESTAMP-LO  PKD(11,0);
DCL DD TWO**32       PKD(11,0) INIT(P'4294967296');

DCL SPCPTR .MAT-PROC-ATTRS INIT(MAT-PROC-ATTRS);
DCL DD      MAT-PROC-ATTRS CHAR(22);
    DCL DD  MAT-BYTES-PROVIDED   BIN(4) DEF(MAT-PROC-ATTRS) POS( 1);
    DCL DD  MAT-BYTES-AVAILABLE  BIN(4) DEF(MAT-PROC-ATTRS) POS( 5);
    DCL DD  MAT-TOTAL-BYTES-USED BIN(4) DEF(MAT-PROC-ATTRS) POS( 9);
    DCL DD  MAT-CPU-TIME-USED   CHAR(8) DEF(MAT-PROC-ATTRS) POS(13);
    DCL DD  MAT-NBR-OF-LOCKS     BIN(2) DEF(MAT-PROC-ATTRS) POS(21);

ENTRY * EXT;
    CPYNV       N, 0;
NEXT-SIZE:
    ADDN(S)     N, 50000;
    CPYNV       RND-RESULT, 314159;
    CPYNV       NBR, 0;
NEXT-ELEMENT:
    MULT        RND-PRODUCT, RND-MULTIPLIER, RND-RESULT;
    ADDN(S)     RND-PRODUCT, RND-INCREMENT;
    REM         RND-RESULT , RND-PRODUCT, RND-MODULUS;

    ADDN(S)     NBR, 1;
    CPYNV       TABLE(NBR), RND-RESULT;
    CMPNV(B)    NBR, N/LO(NEXT-ELEMENT);

SORT:
    CPYNV       CTRL-NBR-ELEMENTS, N;
    CPYBLA      CTRL-DIRECTION, "ASCENDING";
    BRK "1";

    CPYNV       MAT-BYTES-PROVIDED, 20;
    MATPRATR   .MAT-PROC-ATTRS, *, X'21';
    CPYBLA      CPU-BEFORE, MAT-CPU-TIME-USED;

    CALLX      .COMBSORT, COMBSORT, *;

    MATPRATR   .MAT-PROC-ATTRS, *, X'21';
    CPYBLA      CPU-AFTER , MAT-CPU-TIME-USED;
    BRK "2";

TEST-IF-SORTED:
    CPYNV       NBR, N;
COMPARE-WITH-PREVIOUS:
```

```
    SUBN(B)     PREV, NBR, 1/NPOS(=+3);
    CMPBLA(B)   TABLE(NBR), TABLE(PREV)/LO(NOT-SORTED);
    SUBN(SB)    NBR, 1/POS(COMPARE-WITH-PREVIOUS);:

COMPUTE-N*LOG2-N:
    CPYNV       RESULT, N;
    CPYBLAP     MSG-TEXT, RESULT, " ";

    CPYNV       FLT-N, N;
    CMF1        LN-N, X'0011', FLT-N;
    CMF1        LN-2, X'0011', E'2';
    DIV         LOG2-N, LN-N, LN-2;
    MULT        N*LOG2-N, N, LOG2-N;
    CPYNV(R)    RESULT, N*LOG2-N;
    CPYBLA      MSG-TEXT(13:10), RESULT;

COMPUTE-TIME-USED:
    SUBLC       CPU-DIFFERENCE, CPU-AFTER, CPU-BEFORE;
    CPYNV       TIMESTAMP-HI, CPU-DIFF-HI;
    CPYNV       TIMESTAMP-LO, CPU-DIFF-LO;
    MULT        TIMESTAMP, TIMESTAMP-HI, TWO**32;
    ADDN(S)     TIMESTAMP, TIMESTAMP-LO;

    DIV         RESULT, TIMESTAMP, 4096; /* MICROSECONDS */
    CPYBLA      MSG-TEXT(25:10), RESULT;

COMPUTE-SLOPE:
    DIV         SLOPE, RESULT, N*LOG2-N;
    CPYBLA      MSG-TEXT(37:10), SLOPE;

    CPYNV       START, 37;
EDIT-RESULT:
    VERIFY      WHERE, MSG-TEXT(START:11), "0";
    SUBN(SB)    WHERE, 1/NPOS(=+2);
    CPYBLAP     MSG-TEXT(START:WHERE), " ", " ";:
    SUBN(SB)    START, 12/POS(EDIT-RESULT);

SHOW-RESULT:
    CALLI       SHOW-MESSAGE, *, .SHOW-MESSAGE;
    CMPNV(B)    N, 1000000/LO(NEXT-SIZE);
    RTX         *;

NOT-SORTED:
    CPYBLAP     MSG-TEXT, "Not Sorted!", " ";
    CALLI       SHOW-MESSAGE, *, .SHOW-MESSAGE;
    RTX         *;

%INCLUDE SHOWMSG
```

Performance Results

Below are the results of running our test program. Amazingly, Combsort seems to be a true '$kN \log_2 N$' sorting algorithm, with the constant k being 6.725 microseconds on the average (for the 170 box I was running on):

N	*N log₂ N*	Time (µs)	Ratio
50000	780482	5101264	6.5360
100000	1660964	10797200	6.5006
150000	2579190	17325928	6.7176
200000	3521928	23176840	6.5807
250000	4482892	30809992	6.8728
300000	5458381	37077600	6.7928
350000	6445948	42506240	6.5943
400000	7443856	49615200	6.6653
450000	8450804	56946104	6.7385
500000	9465784	63365320	6.6941
550000	10487990	72378584	6.9011
600000	11516762	77620296	6.7398
650000	12551552	85657896	6.8245
700000	13591897	92395368	6.7978
750000	14637398	98955256	6.7604
800000	15687712	103698432	6.6102
850000	16742538	114234688	6.8230
900000	17801609	121012736	6.7979
950000	18864690	127639992	6.7661
1000000	19931569	134387000	6.7424

$$\text{Time} = 6.725 * N * log_2 N \text{ μsec}$$

The “optimization” using an instruction pointer in the inner loop is actually slightly slower for an ascending sort because a branch to the instruction pointer is always taken while the original code with compare and branch only actually branched when the test failed. Since a branch is an expensive operation on a pipelined RISC machine with pre-fetching of instructions, if you can avoid the branch you gain some speed. For a descending sort where the branch must be taken, both versions run at the same speed.

Input/Output in MI Using the SEPT

A File Compressor/Decompressor

In this chapter we'll solve the problem of compressing a large database file for transfer to a different system. Once there, we'll need to decompress (i.e. expand) the file to the same format it had before the compression. We'll use MI to read and write the database file. There are also convenient MI-instructions to do the actual compression/decompression. These instructions work on areas in memory rather than files so we have to read/write to/from these areas as needed.

The User File Control Block

Strictly speaking a *file* is not an MI concept. At the MI-level, data is stored in *spaces*, not files. Files are opened, read/written, then closed. At the MI-level there is no concept of opening/closing of files. Files are defined at the OS/400 level (CPF) above the MI. A program (even an MI-program) accesses a file through a *User File Control Block* (the UFCB). The UFCB defines the filename, library name, possibly a member name, buffer areas and all necessary other control information needed to manage the file. It also provides pointers to the various feedback areas and access to various control structures, such as the *Open Data Path* (the ODP). The main purpose of the UFCB is to provide *device independence* for I/O. You use a UFCB for database files, display files, spool files, save files, tape files, etc. The fact, that these files often have very different internal structure (the actual objects making up the files) is transparent to the programmer.

The UFCB consists of a 208-character fixed part, followed by a variable number of parameter items. Here is the structure of a typical UFCB (including one parameter item):

```
DCL SPCPTR .UFCB INIT(UFCB);
DCL DD UFCB CHAR(214) BDRY(16); /* must be 16-byte aligned */
    DCL SPCPTR .UFCB-ODP              DEF(UFCB) POS(  1);
    DCL SPCPTR .UFCB-INBUF            DEF(UFCB) POS( 17);
    DCL SPCPTR .UFCB-OUTBUF           DEF(UFCB) POS( 33);
    DCL SPCPTR .UFCB-OPEN-FEEDBACK    DEF(UFCB) POS( 49);
    DCL SPCPTR .UFCB-IO-FEEDBACK      DEF(UFCB) POS( 65);
    DCL SPCPTR .UFCB-NEXT-UFCB        DEF(UFCB) POS( 81);

    DCL DD *                  CHAR(32) DEF(UFCB) POS( 97);
    DCL DD UFCB-FILE          CHAR(10) DEF(UFCB) POS(129);
    DCL DD UFCB-LIB-ID        BIN(2)   DEF(UFCB) POS(139);
    DCL DD UFCB-LIBRARY       CHAR(10) DEF(UFCB) POS(141);
    DCL DD UFCB-MBR-ID        BIN(2)   DEF(UFCB) POS(151);
    DCL DD UFCB-MEMBER        CHAR(10) DEF(UFCB) POS(153);

    DCL DD UFCB-DEVICE-NAME   CHAR(10) DEF(UFCB) POS(163);
    DCL DD UFCB-DEVICE-INDEX  BIN(2)   DEF(UFCB) POS(173);

    DCL DD UFCB-FLAGS-1       CHAR(1)  DEF(UFCB) POS(175);
    DCL DD UFCB-FLAGS-2       CHAR(1)  DEF(UFCB) POS(176);

    DCL DD UFCB-REL-VERSION   CHAR(4)  DEF(UFCB) POS(177);
    DCL DD UFCB-INV-MK-COUNT  BIN (4)  DEF(UFCB) POS(181);
    DCL DD UFCB-MORE-FLAGS    CHAR(1)  DEF(UFCB) POS(185);
    DCL DD TAPE-END-OPTION    CHAR(1)  DEF(UFCB) POS(186);
    DCL DD *                  CHAR(22) DEF(UFCB) POS(187);
```

Each item of the variable part is headed by a 2-byte binary number identifying the type of the parameter item. The record-length parameter has an identifier of **1**:

```
    DCL DD UFCB-LENGTH-ID     BIN (2)  DEF(UFCB) POS(209) INIT(1);
    DCL DD UFCB-RECORD-LENGTH BIN (2)  DEF(UFCB) POS(211) INIT(132);
```

The end of the parameter list is signaled by an identifier with the value 32767 or X'7FFF':

```
    DCL DD UFCB-NO-MORE-PARMS BIN (2)  DEF(UFCB) POS(213) INIT(32767);
```

How Do We Know This Stuff?

A good source of information is the generated MI-code produced by the various compilers. Here is an example. Write the following RPG-program (incomplete, but so what?):

```
    FMYFILE  IP  F     132           DISK
```

Compile it with the *LIST generation option (use the RPGIII compiler, as this facility is no longer provided with the ILE/RPG compiler):

```
===> CRTRPGPGM PGM(RPGMIN) GENLVL(50) GENOPT(*LIST)
```

In the spooled listing you'll find all kinds of interesting things, *e.g.* (edited for presentation):

```
/* UFCB DSECT */

DCL SPC .UFCB BAS(.UFCBPTR)
DCL SPCPTR .UCBODPB DIR                    /* OPEN DATA PATH */
DCL SPCPTR .UCBBUFI DIR                    /* INPUT BUFFER */
DCL SPCPTR .UCBBUFO DIR                    /* OUTPUT BUFFER */
DCL SYSPTR .UCBOFBK DIR                    /* OPEN FEEDBACK AREA*/
DCL SPCPTR .UCBIFBK DIR                    /* I/O FEEDBACK AREA */
DCL SPCPTR .UCBNXTU DIR                    /* NEXT */
DCL SPCPTR .UCBSIA DIR                     /* SEP IND AREA */

/*FIXED DATA AREA*/

DCL DD * CHAR(16) DIR                      /* UNUSED */
DCL DD .UCBFNAM CHAR(10) DIR               /* FILENAME */
DCL DD .UCBLBLN BIN(2) DIR                 /* ID FOR LIBRARY NAME */
DCL DD .UCBLIBN CHAR(10) DIR               /* LIBRARY NAME */
DCL DD .UCBMBID BIN(2) DIR                 /* ID FOR MEMBER */
DCL DD .UCBMBER CHAR(10) DIR               /* MEMBER NAME */
DCL DD .UCBLSTD CHAR(10) DIR               /* LAST DEVICE USED */
DCL DD .UCBINDX BIN(2) DIR                 /* LINKAGE TABLE INDEX */
DCL DD .UCBFLG1 CHAR(2) DIR                /* FLAGS */

/*BIT 0-2 CLOSE OPTIONS */
/*BIT 3-4 SHARE ODP OPTIONS */
/*BIT 5-6 SECURE OPTIONS */
/*BIT 7-9  UFCB STATE */
/*BIT 10-13 INPUT, OUTPUT, UPDATE, DELETE */
/*BIT 14-15 UNUSED */

DCL DD * CHAR(4) DIR                       /* VER REL */
DCL DD .UCBIMRK BIN(4) DIR                 /* INVOC MARK CNT */
DCL DD .UCBFLG2 CHAR(1) DIR                /* FLAGS */

/*BIT 0 COUNT INVOC MARKS */
/*BIT 1 TAPE CLOSE PARMS */
/*BIT 2 MULTRCD SPECIFIED */
/*BIT 3-7  UNSED (sic) */

DCL DD * CHAR(1) DIR                       /* TAPE CLOSE OPTION */
DCL DD * CHAR(22) DIR                      /* UNUSED */

/* END OF COMMON PART OF UFCB */
```

Setting Library/File/Member

The name (including type/subtype) of an MI-object is 32 characters long. The name of an MI-object is often the concatenation of several CPF-level names. At the CPF level most names are 10 characters long. A typical CPF-style file name would consist of the filename proper, the library in which the file resides, and a member name, such as `FILE(`*`MYFILE`*`) LIB(*LIBL) MBR(*FIRST)`. The library and member names could be special names such as `*LIBL` and `*FIRST`, or they could refer to a specific library and member. This complexity is dealt with as follows. First, First, let's recap on the part within the UFCB that contains the filename information:

```
DCL DD UFCB-FILE          CHAR(10) DEF(UFCB) POS(129);
DCL DD UFCB-LIB-ID        BIN(2)   DEF(UFCB) POS(139);
DCL DD UFCB-LIBRARY       CHAR(10) DEF(UFCB) POS(141);
DCL DD UFCB-MBR-ID        BIN(2)   DEF(UFCB) POS(151);
DCL DD UFCB-MEMBER        CHAR(10) DEF(UFCB) POS(153);
```

Note the 10-character file, library, and member names. The library and member names are preceded by an *introducer* value. If the library name refers to a specific library, the `LIB-ID` introducer has the value 72. If the name is a special name, the `LIB-ID` introducer is *negative*, -72. The corresponding values for the `MBR-ID` introducer are 73 and -73.

Some AS/400-S/38 History

The introducer values given above are actually the ones used on the AS/400's predecessor, the S/38. The early AS/400 *was* just a S/38 in a differently shaped and colored box. IBM was very proud of the (obvious and vacuous) capability of AS/400 to run S/38-programs, but wanted to prevent AS/400 programs to run on the older box. The devious device IBM used to ensure this was that the compilers on the AS/400 produced code that used introducer values that were *different* from the S/38 values, namely 75 and 71 instead of 72 and 73. The old values still worked (and still do!). To salute the S/38 (the S/38 lives!) I sometimes still use the original introducer values.

It is handy to define some *constants* for the introducer values. Note the use of the **`CON`** keyword rather than of the `DD` keyword:

```
DCL CON *LIBL       BIN(2) INIT(-75); /* S/38:  -72 */
DCL CON *FIRST      BIN(2) INIT(-71); /* S/38:  -73 */
DCL CON LIB-ID      BIN(2) INIT(72);  /* AS/400  75 */
DCL CON MBR-ID      BIN(2) INIT(73);  /* AS/400  71 */
```

You can then initialize the name section in the UFCB like this

```
    CPYBLA       UFCB-FILE,      FILE;
    CPYNV        UFCB-LIB-ID,    THE-LIB;
    CPYBLA       UFCB-LIBRARY,   LIB;
    CPYNV        UFCB-MBR-ID,    THE-MBR;
    CPYBLA       UFCB-MEMBER,    MBR;
```

Or like this

```
    CPYBLA       UFCB-FILE,      FILE;
    CPYNV        UFCB-LIB-ID,   *LIBL;
    CPYBLAP      UFCB-LIBRARY, "*LIBL" , " ";
    CPYNV        UFCB-MBR-ID,   *FIRST;
    CPYBLAP      UFCB-MEMBER,  "*FIRST", " ";
```

Control Flags

A set of sixteen flags controls the operation of the UFCB:

```
    DCL DD UFCB-FLAGS-1       CHAR(1)  DEF(UFCB) POS(175);
    DCL DD UFCB-FLAGS-2       CHAR(1)  DEF(UFCB) POS(176);
```

The flags are grouped into several option fields:

```
    /*  BIT 0-2    CLOSE OPTIONS                                */
    /*  BIT 3-4    SHARE OPTIONS, shared  = B'11'               */
    /*  BIT 5-6    SECURE OPTIONS, secure = B'11'               */
    /*  BIT 7-9    UFCB STATE                                   */
    /*  BIT 10-13  INPUT, OUTPUT, UPDATE, DELETE                */
    /*  BIT 14-15  USER BUFFER OPTIONS, used = B'11'            */
```

The top bit of the close options determines if the file is a temporary file (0) or a permanent file (1). Bits 10-13 control the operation you want to perform:

- Bit 10 1 = Input
- Bit 11 1 = Output
- Bit 12 1 = Update
- Bit 13 1 = Delete

For normal operation just leave all the other flag bits off. We need two UFCBs. One for input (`IFCB`) and one for output (`OFCB`). The flag settings should then be:

```
    DCL DD IFCB-FLAGS-1     CHAR(1)  DEF(IFCB) POS(175) INIT(X'80'); /* permanent file */
    DCL DD IFCB-FLAGS-2     CHAR(1)  DEF(IFCB) POS(176) INIT(X'20'); /* input          */

    DCL DD OFCB-FLAGS-1     CHAR(1)  DEF(OFCB) POS(175) INIT(X'80'); /* permanent file */
    DCL DD OFCB-FLAGS-2     CHAR(1)  DEF(OFCB) POS(176) INIT(X'10'); /* output         */
```

More Control Flags

There is yet another control flag field, UFCB-MORE-FLAGS, with the following bit settings:

Bit 0 If set, user must set the invocation mark, otherwise QDMOPEN sets the invocation mark.
Bit 1 If set indicates that QDMCLOSE is to merge the close parameter into the ODP.
Bit 2 If set, blocked records will be transferred to the users buffer on each IO.
Bit 3 If set, IO routines will assume that the indicators are in a separate indicator area.
Bit 4 If set, all program devices defined to the file are to be acquired when the file is opened.
Bit 5 If set and the file is open for input and output, a single buffer is used for both input and output.
Bit 6-7 Unused

This flag is a "grab-bag" of various options. Setting it to all zeroes will ordinarily suffice. Setting bit 2, that controls blocking, can sometimes improve I/O performance dramatically.

The FCMPRS (File Compress) Command

We need a simple user interface to specify the files, something like this:

```
                        Compress/Decompress File (FCMPRS)

Type choices, press Enter.

(C)ompress/(D)ecompress  . . . . FCT            C
In File  . . . . . . . . . . . . INFILE         _________.
In Library . . . . . . . . . . . INLIB          _________.
Out File . . . . . . . . . . . . OUTFILE        _________.
Out Library  . . . . . . . . . . OUTLIB         _________.
```

Or from the command line:

```
===> FCMPRS FCT(C) INFILE(MINE) INLIB(MYLIB) OUTFILE(YOURS) OUTLIB(YOURLIB)
```

We envision five parameters, the first being an operation code: "C" for compress or "D" for decompress. This book is not about how to write commands or CL-programs, so we take a simplistic approach in order to concentrate on the machine-level coding. Here is the simple command source (**FCMPRS**/QCMDSRC):

```
CMD   PROMPT('Compress/Decompress File')
PARM  KWD(FCT)     TYPE(*CHAR) LEN(1)  PROMPT('(C)ompress/(D)ecompress')
PARM  KWD(INFILE)  TYPE(*CHAR) LEN(10) PROMPT('In File')
PARM  KWD(INLIB)   TYPE(*CHAR) LEN(10) PROMPT('In Library')
PARM  KWD(OUTFILE) TYPE(*CHAR) LEN(10) PROMPT('Out File')
PARM  KWD(OUTLIB)  TYPE(*CHAR) LEN(10) PROMPT('Out Library')
```

Command-Processing Program

Compile the command specifying the following CL-Program (**CLFCMPRS**/CLSRC) as command-processing program:

```
      PGM PARM(&FCT &INFILE &INLIB &OUTFILE &OUTLIB)
      DCL VAR(&FCT)     TYPE(*CHAR) LEN(1)
      DCL VAR(&INFILE)  TYPE(*CHAR) LEN(10)
      DCL VAR(&INLIB)   TYPE(*CHAR) LEN(10)
      DCL VAR(&OUTFILE) TYPE(*CHAR) LEN(10)
      DCL VAR(&OUTLIB)  TYPE(*CHAR) LEN(10)
      DLTF    FILE(&OUTLIB/&OUTFILE)
      MONMSG  MSGID(CPF2105)  /* FILE NOT FOUND */
      CRTPF   FILE(&OUTLIB/&OUTFILE) RCDLEN(132) +
                SIZE(*NOMAX) LVLCHK(*NO)
      CALL PGM(MIFCMPRS) PARM(&FCT &INFILE &INLIB &OUTFILE &OUTLIB)
      ENDPGM
```

Parameter Block

The MI-program, **MIFCMPRS**, is where the real work is done. As you can see, it has five parameters:

```
DCL SPCPTR .PARM1 PARM;
    DCL DD PARM-OPERATION CHAR(1) BAS(.PARM1); /* better name would be FCT, why? */
```

```
DCL SPCPTR .PARM2 PARM;
    DCL DD PARM-IN-FILE  CHAR(10) BAS(.PARM2);

DCL SPCPTR .PARM3 PARM;
    DCL DD PARM-IN-LIB   CHAR(10) BAS(.PARM3);

DCL SPCPTR .PARM4 PARM;
    DCL DD PARM-OUT-FILE CHAR(10) BAS(.PARM4);

DCL SPCPTR .PARM5 PARM;
    DCL DD PARM-OUT-LIB  CHAR(10) BAS(.PARM5);

DCL OL PARMS(.PARM1, .PARM2, .PARM3, .PARM4, .PARM5) EXT PARM MIN(5);
```

We shall assume that the libraries are specified explicitly and that the physical files have only one member with the same name as the file. It is straightforward to extend this example to deal with the library list and/or a different member, so we'll leave that as an exercise for the reader. We'll also omit checking for parameter validity and the like; these things are easy, but tedious, and add 'clutter' to the example program which will detract from the purpose of this chapter.

The System Entry Point Table (SEPT)

CPF and API calls are done by using the CALLX instruction referencing a system pointer that is obtained from the *System Entry Point Table*. No call by name should be used because of library list considerations. The SEPT is a table of pre-resolved system pointers. The table is a special space object (`QINSEPT/QSYS` type `X'19C3'`) that you can dump using the `DUMPSYSOBJ` command:

```
 ===> DMPSYSOBJ OBJ(QINSEPT) CONTEXT(QSYS)
```

```
DMPSYSOBJ PARAMETERS
OBJ- QINSEPT                          CONTEXT- QSYS
TYPE- *ALL SUBTYPE-*ALL
OBJECT TYPE-           SPACE                                        *SEPT
NAME-        QINSEPT                        TYPE-          19   SUBTYPE-          C3
LIBRARY-     QSYS                           TYPE-          04   SUBTYPE-          01
CREATION-    01/29/98  06:34:12             SIZE-          000001B000
OWNER-       QSYS                           TYPE-          08   SUBTYPE-          01
ATTRIBUTES-          0800                   ADDRESS-       2816099C68 000000
```

When you do that and select the `QPSRVDMP` spool file with `WRKSPLF` you see something like the following; find the `POINTERS` section:

```
                          Display Spooled File
 File  . . . . . :   QPSRVDMP                          Page/Line   54/47
 Control . . . . .                                     Columns     1 - 78
 Find  . . . . . .   POINTERS

  1  000000   SYP 02 01 QT3REQIO     04 01 QSYS    3F10  0000  *PGM
  2  000010   SYP 02 01 QWSCLOSE     04 01 QSYS    3F10  0000  *PGM
  3  000020   SYP 02 01 QSFGET       04 01 QSYS    3F10  0000  *PGM
  4  000030   SYP 02 01 QWSOPEN      04 01 QSYS    3F10  0000  *PGM
  5  000040   SYP 02 01 QWSPBDVR     04 01 QSYS    3F10  0000  *PGM
  6  000050   SYP 02 01 QWSRST       04 01 QSYS    3F10  0000  *PGM
  7  000060   SYP 02 01 QWSRTSFL     04 01 QSYS    3F10  0000  *PGM
  8  000070   SYP 02 01 QSFCRT       04 01 QSYS    3F10  0000  *PGM
  9  000080   SYP 02 01 QWSSPEND     04 01 QSYS    3F10  0000  *PGM
 10  000090   SYP 02 01 QDCVRX       04 01 QSYS    3F10  0000  *PGM
 11  0000A0   SYP 02 01 QDMCLOSE     04 01 QSYS    3F10  8000  *PGM
 12  0000B0   SYP 02 01 QDMCOPEN     04 01 QSYS    3F10  8000  *PGM
 13  0000C0   SYP 02 01 QDBCLOSE     04 01 QSYS    3F10  0000  *PGM
 14  0000D0   SYP 02 01 QDBGETDR     04 01 QSYS    3F10  8000  *PGM
 15  0000E0   SYP 02 01 QDBGETKY     04 01 QSYS    3F10  8000  *PGM
 16  0000F0   SYP 02 01 QDBGETSQ     04 01 QSYS    3F10  8000  *PGM
 17  000100   SYP 02 01 QDBOPEN      04 01 QSYS    3F10  0000  *PGM
 18  000110   SYP 02 01 QDBPUT       04 01 QSYS    3F10  8000  *PGM
 19  000120   SYP 02 01 QDBUDR       04 01 QSYS    3F10  8000  *PGM
 20  000130   SYP 02 01 QSPBPPRT     04 01 QSYS    3F10  0000  *PGM
 21  000140   SYP 02 01 QOESETEX     04 01 QSYS    3F10  0000  *PGM
 22  000150   SYP 02 01 QWSPUT       04 01 QSYS    3F10  8000  *PGM
 23  000160   SYP 02 01 QWSMEEH      04 01 QSYS    3F10  0000  *PGM
 24  000170   SYP 02 01 QWSMSG       04 01 QSYS    3F10  0000  *PGM
 25  000180   SYP 02 01 QWSPTMSG     04 01 QSYS    3F10  0000  *PGM
 26  000190   SYP 02 01 QLPCTLIN     04 01 QSYS    3F10  0000  *PGM
 27  0001A0   SYP 02 01 QLPTRANS     04 01 QSYS    3F10  0000  *PGM
 28  0001B0   SYP 02 01 QWCITUNR     04 01 QSYS    3F10  0000  *PGM
…etc
```

Highlighted **entries** are programs in the user domain that can be called from user-state programs. The others are system-domain programs that are internal to OS/400. Entry numbers 11 (QDMCLOSE) and 12 (QDMCOPEN) are the entry points to the Data Management Close/Open functions.

The Process Communication Object (PCO) contains at it very beginning a pointer to SEPT:

```
DCL DD PCO CHAR(80) BASPCO;           /* recall the BASPCO, ch. 9   */
   DCL SPCPTR ..SEPT DEF(PCO) POS(1); /* pointer to space with SEPT */
```

The SEPT itself contains thousands of system pointers. At last count about 6440 were found, and this number keeps going up from release to release:

```
DCL SYSPTR .SEPT(6440) BAS(..SEPT);
```

Only Entries in the user domain have fixed positions within the SEPT. System domain entry positions may (and occasionally do) change from release to release.

Opening a File

The operand list for QDMCOPEN contains as its sole argument a space pointer to the file control block (*x*FCB) to identify the file to open:

```
DCL SPCPTR .IFCB INIT(IFCB); /* input FCB  */
DCL SPCPTR .OFCB INIT(OFCB); /* output FCB */

DCL OL OPEN-I(.IFCB) ARG;   /* for the input file  */
DCL OL OPEN-O(.OFCB) ARG;   /* for the output file */
```

We can now open the files:

```
DCL CON OPEN-ENTRY BIN(2) INIT(12); /* entry number for OPEN function */

OPEN-INPUT-FILE:
    CPYBLA       IFCB-FILE,    PARM-IN-FILE;
    CPYNV        IFCB-LIB-ID,  THE-LIB;
    CPYBLA       IFCB-LIBRARY, PARM-IN-LIB;
    CPYNV        IFCB-MBR-ID,  THE-MBR;
    CPYBLA       IFCB-MEMBER,  PARM-IN-FILE;
    CALLX       .SEPT(OPEN-ENTRY), OPEN-I, *;

OPEN-OUTPUT-FILE:
    CPYBLA       OFCB-FILE,    PARM-OUT-FILE;
    CPYNV        OFCB-LIB-ID,  THE-LIB;
    CPYBLA       OFCB-LIBRARY, PARM-OUT-LIB;
    CPYNV        OFCB-MBR-ID,  THE-MBR;
    CPYBLA       OFCB-MEMBER,  PARM-OUT-FILE;
    CALLX       .SEPT(OPEN-ENTRY), OPEN-O, *;
```

If you place breakpoints before and after the open-call:

```
    BRK "1";
    CALLX       .SEPT(OPEN-ENTRY), OPEN-I, *;
    BRK "2";
```

you can see the effects of the open (filling in of pointers to buffers, etc):

```
                            Display Breakpoint
  Variable  . . . . . . . . . . . . . . . . : IFCB
    Type  . . . . . . . . . . . . . . . . . :   CHARACTER
    Length  . . . . . . . . . . . . . . . . :   214
    * . . . + . . . . 1 . . . . + .    *...+....1....+.
    00000000000000000000000000000000  '                '
    00000000000000000000000000000000  '                '
    00000000000000000000000000000000  '                '
    00000000000000000000000000000000  '                '
    00000000000000000000000000000000  '                '
    00000000000000000000000000000000  '                '
    00000000000000000000000000000000  '                '
    00000000000000000000000000000000  '                '
    E3D6D7C34040404040400048D3E2E5C1  'TOPC       ÇLSVA'
    D3C7C1C1D9C40049E3D6D7C340404040  'LGAARD ñTOPC     '
    4040000000000000000000000000008020  '               Ø '
    00000000000000000000000000000000  '                 '
    00000000000000000000000000000000  '                 '
    000100847FFF                      '    d" '
```

```
Variable  . . . . . . . . . . . . . . . . : IFCB
  Type  . . . . . . . . . . . . . . . . . :   CHARACTER
  Length  . . . . . . . . . . . . . . . . :   214
  * . . . + . . . . 1 . . . . + .    *...+....1....+.
  8000000000000000D333631263000650  'Ø       L Ä Ä  &' open data path
  8000000000000000D333631263000E60  'Ø       L Ä Ä  -' input buffer
  00000000000000000000000000000000  '                '
  8000000000000000D333631263000700  'Ø       L Ä Ä    ' open feedback area
  8000000000000000D333631263000816  'Ø       L Ä Ä    ' i/o feedback area
  00000000000000000000000000000000  '                '
  00000000000000000000000000000000  '                '
  00000000000000000000000000000000  '                '
  E3D6D7C3404040404040400048D3E2E5C1  'TOPC       çLSVA'
  D3C7C1C1D9C40049E3D6D7C340404040  'LGAARD ñTOPC    '
  40404040404040404040404000018120  '               a ' device name, flags (state)
  0000000000000D970000000000000000  '       p         ' invocation mark count
  00000000000000000000000000000000  '                '
  000100847FFF                      '    d" '
```

Note that the state field in the control flags is now set to "1" indicating an open in progress.
Although we could use the pointer to the buffer provided in the FCB we prefer to use a copy of it (gives us more flexibility - e.g. if we want to re-use the FCB), so:

```
DCL SPCPTR .INBUF;
DCL DD INBUF  CHAR(132) BAS(.INBUF);   /* base the input buffer  */

DCL SPCPTR .OUTBUF;
DCL DD OUTBUF CHAR(132) BAS(.OUTBUF);  /* base the output buffer */

    CPYBWP      .INBUF,  .IFCB-INBUF;  /* make copy of pointer to input  */
    CPYBWP      .OUTBUF, .OFCB-OUTBUF; /* make copy of pointer to output */
```

We now have open files with buffers defined. The SEPT contains pointers to the routines that read and write records, but these routines are different for different devices, so how do we know which ones to use? Finding the correct pointer is done through the *Data Management Entry Point Table* (DMEPT) in the *Open Data Path* (the ODP). After a file has been opened, this table has the correct indices into the SEPT for each I/O routine as fitting for the device in question. This allows the Data Management component of OS/400 to direct the I/O to the correct device/file consistent with user requests including overrides (e.g., to a different device or file type). Only this method for calling I/O routines allows Data Management to handle overrides and error conditions correctly.

The Open Data Path

There is an interesting trend on IBM's website for AS/400 documentation. For release V3R2, I count 41 books with a reference to the ODP, in V4R5 I only see 28. In either case the information is scant. Apart from a passing reference here and there, the following is the only substantial (?) information available:

> "Open file operations result in the creation of a temporary resource called an open data path (ODP). The open function can be started by using HLL open verbs, the Open Query File (OPNQRYF) command, or the Open Data Base File (OPNDBF) command. The ODP is scoped to the activation group of the program that opened the file. For OPM or ILE programs that run in the default activation group, the ODP is scoped to the call-level number. To change the scoping of HLL open verbs, an override may be used. You can specify scoping by using the open scope (OPNSCOPE) parameter on all override commands, the OPNDBF command, and the OPNQRYF command. For more information about open file operations, see the *Data Management* book."

This is even more surprising because the ODP control block is *the* central control block of an opened device file or database member. The first part of the ODP (often called the ODP *root*) has a fixed format The information contained in the root section includes:

- Status information
- Various size information about the space in which the ODP control block resides
- Offset to the Device File/Database MCB (Member Control Block)
- Offsets to various sections within the ODP
- Information (such as names and "open" options) common to several components

Internally there are two types of ODPs, a *permanent* ODP (sometimes called the prototype ODP) and an *active* ODP. When a device file/database member is created, an associated ODP is also created. When the device/member is opened, a temporary duplicate of the associated is created. This duplicate is the active ODP. The duplicate is destroyed when the file is closed or the job ends.

For database members, the permanent ODP at the MI-level is a *cursor* object (type/subtype X'0D50'). It is simply the database member object. Its name is the concatenation of filename and member name. The associated ODP is also a cursor (type/subtype X'0DEF') called the *operational cursor*. Its name is the concatenation of filename, library name, and member name. Refer to chapter 8 for more information about member cursors. Here is a dump of an operational cursor for database member **TOPC** of file **TOPC** in library **LSVALGAARD**:

```
Address D911A9DBA5 000000
000000  00010008 00810001  D911A9DB A5000000    ·····a··R·zûv···
000010  00010000 00000000  D911A9DB A5000650    ········R·zûv··&
000020  10000DEF E3D6D7C3  40404040 4040D3E2    ···ÕTOPC      LS
000030  E5C1D3C7 C1C1D9C4  E3D6D7C3 40404040    VALGAARDTOPC
000040  40408000 000009B0  00000008 FF1C4100      Ø····^·········
000050  81509637 A5D48000  00000000 00000000    a&o·vMØ·········
000060  DB27F422 1A000000  00000000 00000000    û·4·············
000070  D911A9DB A5000000  00020000 01000000    R·zûv···········
000080  81509637 A5D48000  00000000 00000000    a&o·vMØ·········
000090  00000000 00000000  00000000 00000000    ················
0000A0  00000000 00000073  00000000 00000000    ·······Ë········
0000B0  00000000 00000000  00000000 00000000    ················
0000C0  00000000 00000000  E0000000 00000000    ········\·······
0000D0  00000000 00000000  00000000 00000000    ················
0000E0  00000000 00000000  00000000 00000000    ················
0000F0  00730000 00000000  00000000 00000000    ·Ë··············
000100  00000000 00000000  3ECAFBBD C2000000    ·········-Û¨B··· <= address of member
000110  52040000 0000EB80  00010000 00060000    ê·····ÔØ·····
000120  D911A9DB A5000400  00000039 40000000    R·zûv·····  ···

Address 3ECAFBBDC2 000000
000000  00010008 00898000  3ECAFBBD C2000000    ·····iØ··-Û¨B···
000010  40010000 00000000  3ECAFBBD C2000830      ·····-Û¨B···
000020  80000D50 E3D6D7C3  40404040 4040E3D6    Ø··&TOPC      TO
000030  D7C34040 40404040  40404040 40404040    PC
000040  40408000 000007D0  00000008 3F104100      Ø····}·····
000050  808C50FA E3AF8000  0E6792A4 C7000000    øð&³T®Ø··ÅkuG···
000060  00000000 00000000  173A0D5C 03000000    ··········*····
000070  3ECAFBBD C2000000  00020000 03000000    ·-Û¨B··········
000080  814EEC7F 47228000  00000000 00000000    a·Ö"å·Ø·········
000090  00000000 00000000  00000000 00000000    ················
```

The ODP is strictly speaking located in the associated space of the cursor objects. We prefer to have our own copy of the pointer to the ODP (we'll use the same pointer in turn for both the ODP of the input file and for the ODP of the output file), so

```
    CPYBWP      .ODP, .IFCB-ODP;  /* copy pointer to input ODP */
```

Here is the layout of the fixed part of the ODP (note the use of a **SPC**). :

```
DCL SPCPTR .ODP;
DCL SPC     ODP  BAS(.ODP);
    DCL DD ODP-STATUS       CHAR(4)   DIR;
    DCL DD ODP-DEV-LENGTH   BIN(4)    DIR;
    DCL DD ODP-OPEN-SIZE    BIN(4)    DIR;
    DCL DD ODP.OPEN-FEEDBCK BIN(4)    DIR;
    DCL DD ODP.DEV-CTRLBLK  BIN(4)    DIR;  <=== discussed below
    DCL DD ODP.IO-FEEDBACK  BIN(4)    DIR;
    DCL DD ODP.LOCK-LIST    BIN(4)    DIR;
    DCL DD ODP.SPOOL-OUTPUT BIN(4)    DIR;

    DCL DD ODP.MBR-DESCR    BIN(4)    DIR;
    DCL DD ODP.CUR-IN-REC   BIN(4)    DIR;
    DCL DD ODP.CUR-OUT-REC  BIN(4)    DIR;
    DCL DD ODP.OPEN-DMCQ    BIN(4)    DIR;
    DCL DD ODP.OUTSTANDINGS BIN(4)    DIR;
    DCL DD *                CHAR(12)  DIR;

    DCL SYSPTR .ODP-CURSOR            DIR;
    DCL DD      *           CHAR(16)  DIR;
    DCL SPCPTR .ODP-CDM-ERROR         DIR;
```

```
    DCL SPCPTR .ODP-INPUT-BUFFER      DIR;
    DCL SPCPTR .ODP-OUTPUT-BUFFER     DIR;

    DCL DD ODP.CDM-CLOSING  BIN(2)    DIR;
    DCL DD ODP-DEV-NAME-IDX BIN(2)    DIR;
    DCL DD ODP-NBR-OF-DEVS  BIN(2)    DIR;

    DCL DD ODP-SEQUENCE-NBR BIN(4)    DIR;
    DCL DD ODP-REC-LENGTH   BIN(2)    DIR;
    DCL DD ODP-REC-LENGTH2  BIN(2)    DIR;
    DCL DD ODP-NBR-OF-*RDS  BIN(2)    DIR;
    DCL DD ODP-RELEASE-NBR  BIN(2)    DIR;
    DCL DD ODP-OPEN-POSN    CHAR(1)   DIR;
    DCL DD ODP-OVR-REC-LEN  BIN(2)    DIR;
    DCL DD ODP-COM-DEV-CNT  BIN(2)    DIR;

    DCL DD ODP.INPUT-BPCA   BIN(4)    DIR;
    DCL DD ODP.OUTPUT-BPCA  BIN(4)    DIR;
    … stuff omitted
```

Looking at the ODP with the Debugger shows:

```
  Variable  . . . . . . . . . . . . . . . . : ODP
    Type  . . . . . . . . . . . . . . . . . :   CHARACTER
    Length  . . . . . . . . . . . . . . . . :   32767
    * . . . + . . . . 1 . . . . + . .    *...+....1....+.
    81000000000008A0000009B0000000B0  'a      µ   ^   ^'
    00000140000001C60000028000000000  '       F   Ø     ' <== offset to namelist
    000002C0000000000000000000002E0   '   {           \'
    00000000000000000000000000000000  '                 '
    0000800000000000CFE48EE96E000DFF  '  Ø     õUþZ>    ' <== temporary copy of active ODP
    0000051000000002000004900000000   '                 '
    80000000000000000EA12C952D364DC10  'Ø       ² IêLÀü '
    80000000000000000CFE48EE96E000E60  'Ø       õUþZ>  -' <== input buffer
    00000000000000000000000000000000  '                 '
    01900000000000000000008400000000  ' °          d    '
    2C0000000000000000007F00000000000  '            0   '
    C4C2E3D6D7C3404040404040D3E2E5C1  'DBTOPC      LSVA'
    D3C7C1C1D9C400000000000000000000  'LGAARD            '
    00000000000000000000000000840000  '              d  '
    E3D6D7C340404040404000000008F0000  'TOPC          ±  '
    00000015000000000000000000004B7C1  '             ¼A'
    D900D58000000000000000000000000000  'R NØ             '
    00000000000000000000010000008F00  '              ±  '
    00043000000000000000000000000001  '                 '
    0000000100004C0000FFFF0000000000  '      <          '

    00010001C4C1E3C1C2C1E2C540400000  '    DATABASE     ' Device control block:
    00000000000000000010000E00450045  '              á á'
    0045004500450045006F004500450045  ' á á á á ? á á á'
    00450BFD00450045000D001100000001  ' á Ù á á         '
    00000000000000000000000000000000  '                 '
    00000000000000000000000000000000  '                 '
```

Data Management Entry Point Table

Here is not the place for a discussion of all the fields of the ODP. What we are interesting in at this point is the Data Management Entry Point table. The **ODP.DEV-CTRLBLK** is an offset to what we need. To get a pointer to this area we add the offset to the pointer to the ODP:

```
    ADDSPP      .DEV-CONTROL-BLOCK, .ODP, ODP.DEV-CTRLBLK;
```

Below is the format of the *Device Control Block* (values from the above dump are shown as /* comments */). The DMEPT starts 24 bytes into the DCB:

```
DCL SPCPTR .DEV-CONTROL-BLOCK;
DCL SPC     DEV-CONTROL-BLOCK BAS(.DEV-CONTROL-BLOCK);
    DCL DD  DCB-MAX-NBR-OF-DEVICES   BIN( 2) DIR;   /* 0001                        */
    DCL DD  DCB-DEVICES-IN-THE-ODP   BIN( 2) DIR;   /* 0001                        */
    DCL DD  DCB-DEVICE-NAME         CHAR(10) DIR;   /* DATABASE                    */
    DCL DD  DCB-OFFSET-TO-FM-WORK    BIN( 4) DIR;   /* 00000000                    */
    DCL DD  DCB-LENGTH-OF-FM-WORK    BIN( 4) DIR;   /* 00000000                    */
    DCL DD  DCB-INDEX-FOR-LUD-PTR    BIN( 2) DIR;   /* 0000                        */
    /*  DMEPT starts here */
    DCL DD  DCB-GET                  BIN( 2) DIR;   /* 0010   = 16     QDBGETSQ   */
    DCL DD  DCB-GET-BY-RRN           BIN( 2) DIR;   /* 000E   = 14     QDBGETDR   */
    DCL DD  DCB-GET-BY-KEY           BIN( 2) DIR;   /* 0045   = 69     QDMIFERR   */
```

```
    DCL DD  *                   BIN( 2) DIR;   /* 0045   = 69     QDMIFERR   */
    DCL DD  DCB-PUT             BIN( 2) DIR;   /* 0045   = 69     QDMIFERR   */
    DCL DD  DCB-PUT-GET         BIN( 2) DIR;   /* 0045   = 69     QDMIFERR   */
    DCL DD  DCB-UPDATE          BIN( 2) DIR;   /* 0045   = 69     QDMIFERR   */
    DCL DD  DCB-DELETE          BIN( 2) DIR;   /* 0045   = 69     QDMIFERR   */
    DCL DD  DCB-FORCE-EOD       BIN( 2) DIR;   /* 006F   = 111    QDBFEOD    */
    DCL DD  DCB-FORCE-EOV       BIN( 2) DIR;   /* 0045   = 69     QDMIFERR   */
    DCL DD  DCB-COMMIT          BIN( 2) DIR;   /* 0045   = 69     QDMIFERR   */
    DCL DD  DCB-ROLLBACK        BIN( 2) DIR;   /* 0045   = 69     QDMIFERR   */
    DCL DD  DCB-FREE-REC-LOCK   BIN( 2) DIR;   /* 0045   = 69     QDMIFERR   */
    DCL DD  *                   BIN( 2) DIR;   /* 0BFD   = 3069   QDBCHEOD   */
    DCL DD  *                   BIN( 2) DIR;   /* 0045   = 69     QDMIFERR   */
    DCL DD  *                   BIN( 2) DIR;   /* 0045   = 69     QDMIFERR   */
    DCL DD  DCB-CLOSE           BIN( 2) DIR;   /* 000D   = 13     QDBCLOSE   */
    DCL DD  DCB-OPEN            BIN( 2) DIR;   /* 0011   = 17     QDBOPEN    */
    DCL DD  *                   BIN( 2) DIR;   /* 0000               */
    DCL DD  *                   BIN( 2) DIR;   /* 0001               */
```

Each entry in the DMEPT is set to an index or subscript into the SEPT at open time. If an entry does not apply to a particular device/file type, these entry points within the SEPT will be to a program that will signal an appropriate exception saying that the requested operation is not supported or applicable for that device. As you can see, all the output operations for this open input file lead to the same error routine - SEPT entry number 69, `QDMIFERR`. We are interested in the read-routine, which is SEPT entry number 16 - `QDBGETSQ`. We save that SEPT entry number for later use:

```
    CPYNV        GET-ENTRY, DCB-GET;
```

Similar considerations apply for the output file, so opening the two files now proceeds like this:

```
DCL DD  GET-ENTRY  BIN(2);
DCL DD  PUT-ENTRY  BIN(2);

OPEN-INPUT-FILE:
    …
    CPYBWP      .INBUF, .IFCB-INBUF;
    CPYBWP      .ODP-ROOT, .IFCB-ODP;
    ADDSPP      .DEV-CONTROL-BLOCK, .ODP-ROOT, ODP.DEV-NAMELIST;
    CPYNV        GET-ENTRY, DCB-GET;

OPEN-OUTPUT-FILE:
    …
    CPYBWP      .OUTBUF, .OFCB-OUTBUF;
    CPYBWP      .ODP-ROOT, .OFCB-ODP;
    ADDSPP      .DEV-CONTROL-BLOCK, .ODP-ROOT, ODP.DEV-NAMELIST;
    CPYNV        PUT-ENTRY, DCB-PUT;
```

I/O Routines

All OS/400 CPF-level I/O routines accept three arguments: an open UFCB, an option list, and a control list. An argument may be left out or coded as a **`NULL`** pointer, if it is not applicable. Let's set up the operand lists for `GET` and `PUT` operations.

It would be convenient if we could initialize a null pointer or pass a null pointer as a parameter like this:

```
DCL SPCPTR .NULL INIT(*);           /* invalid initialization */
DCL OL GET-OPERATION(.IFCB, .GET-PARM, *); /* invalid operand */
```

but we can't because the MI-compiler complains, so we have to resort to the following method:

```
DCL SPCPTR .NULL;
DCL OL GET-OPERATION(.IFCB, .GET-OPTION, .NULL);
    CPYBWP .NULL, *; /* Make NULL ptr at runtime */
```

The Data Management Option List

The data management option list is passed to the I/O programs as a space pointer that points to a four-byte option list, which in turn is a fixed structure that has four one-byte substructures.

Operation option byte (byte 0):

X'00'	Release All
X'01'	Release First

X‘01’ Get First
X‘02’ Get Last
X‘03’ Get Next, Wait
X‘04’ Get Previous
X‘05’ GetK Next, Unique
X‘06’ GetK Previous, Unique
X‘07’ GetD RelCur
X‘08’ GetD Ordinal RelStr, wait

X‘13’ Get Next, No Wait
X‘23’ Get Next, Wait via Acc Input
X‘83’ Get Next, Wait via Event Handler
X‘0C’ Get Cancel
X‘0E’ Get Same

X‘18’ GetD Ordinal RelStr, No Wait
X‘0A’ GetD Next Mod, Wait
X‘1A’ GetD Next Mod, No Wait
X‘0B’ GetD Unblocked, Wait
X‘1B’ GetD Unblocked, No Wait
X‘0C’ GetD Cancel

X‘0B’ GetK Key Equal
X‘0A’ GetK Key Before or Equal
X‘09’ GetK Key Before
X‘0C’ GetK Key After or Equal
X‘0D’ GetK Key After

X‘00’ Put, Wait
X‘10’ Put, No Wait
X‘03’ PutGet Next, Wait
X‘13’ PutGet Next, No Wait
X‘0C’ PutGet Cancel
X‘0F’ Put Send Error Message

Share option byte (byte 1):

X‘00’ Get for Read Only, Normal
X‘03’ Get for Update, Normal
X‘10’ Get for Read Only, No Position
X‘13’ Get for Update, No Position

Data option byte (byte 2):

X‘00’ Data Access Record
X‘01’ No Data Access Record
X‘02’ Data Access All Record
X‘03’ No Data Access All Record

X‘10’ Get Prior, Data Access Record
X‘11’ Get Prior, No Data Access Record
X‘12’ Get Prior, Data Access All Record
X‘13’ Get Prior, No Data Access All Record

Device Support byte (byte 3):

X‘01’ Indicating a Get Request
X‘02’ Indicating a GetD Request
X‘05’ Indicating a Put Request
X‘06’ Indicating a PutGet Request
X‘07’ Indicating an Update Request
X‘08’ Indicating a Delete Request
X‘0B’ Indicating a Release Request

So, based on all that, we set up the option lists as follows:

```
DCL DD GET-OPTION BIN(4) INIT(H'03000001'); /* Get Next, Wait.   Get Request */
DCL SPCPTR .GET-OPTION INIT(GET-OPTION);

DCL DD PUT-OPTION BIN(4) INIT(H'10000005'); /* Put, No Wait.     Put Request */
DCL SPCPTR .PUT-OPTION INIT(PUT-OPTION);
```

The Data Management Control List

The control list is used to pass additional information to the I/O programs as needed. The control list is a variable length structure made up of elements with the following format:

Entry ID, CHAR(1)	Entry Value Length, BIN(2)	Entry Value, CHAR(nn)

The last entry must have a single type byte of X'FF' to indicate the end of the list. For the simple operations we are performing here on the files, no control list is needed, so we code that argument as a **NULL** pointer:

```
DCL OL GET-OPERATION(.IFCB, .GET-PARM, .NULL);
```

Filling the Input Area

Because the data compression works best on largish amounts of data we'll define an input area capable of holding 5000 132-character records:

```
DCL SPCPTR .INPUT-SPACE INIT(INPUT-SPACE);
DCL DD      INPUT-SPACE (5000)  CHAR(132) BDRY(16);
```

We'll keep reading records until either the area is full or we get and end-of-file exception:

```
DCL EXCM * EXCID(H'5001') BP(EOF-DETECTED) CV("CPF") IMD;  ----------------.
                                                                           |
    CPYNV        INPUT-RECS, 0;   /* counting input records    */          |
    CPYBLA       INPUT-EOF, " ";  /* End-of-file indicator off */          |
                                                                           |
READ-UNCOMPRESSED-RECORD:       <-------------------------------------.    |
    CMPNV(B)     INPUT-RECS, 5000/EQ(COMPRESS-CHUNK);   ----.         |    |
    CALLX       .SEPT(GET-ENTRY), GET-OPERATION, *;         |         |    |
    ADDN(S)      INPUT-RECS, 1;                             |         |    |
    CPYBLA       INPUT-SPACE(INPUT-RECS), INBUF;            |         |    |
    B            READ-UNCOMPRESSED-RECORD;     ------------------'    |    |
                                                           |          |    |
EOF-DETECTED:                        <---------------------|----------'    |
    CPYBLA       INPUT-EOF, "Y";                           |               |
                                                           |               |
COMPRESS-CHUNK:                           <----------------'               |
```

Compressing the Input

The "Compress Data"-instruction, **CPRDATA**, compresses data of a specified length. The operand template identifies the data to be compressed and the result to receive the compressed data. The template has this format:

```
DCL SPCPTR .COMPRESS INIT(COMPRESS);
DCL DD      COMPRESS CHAR(64) BDRY(16); /* must be 16-byte aligned */
    DCL DD  CMPR-INPUT-LENGTH    BIN(4) DEF(COMPRESS) POS( 1);
    DCL DD  CMPR-OUTPUT-LENGTH   BIN(4) DEF(COMPRESS) POS( 5); /* max length of result */
    DCL DD  CMPR-ACTUAL-LENGTH   BIN(4) DEF(COMPRESS) POS( 9); /* length of compressed */
    DCL DD  CMPR-ALGORITHM       BIN(2) DEF(COMPRESS) POS(13);
    DCL DD  *                  CHAR(18) DEF(COMPRESS) POS(15);
    DCL SPCPTR .CMPR-INPUT              DEF(COMPRESS) POS(33); /* pointer to input     */
    DCL SPCPTR .CMPR-OUTPUT             DEF(COMPRESS) POS(49); /* pointer to result    */
```

We have two choices for the compression algorithm:

1 Simple TERSE algorithm (whatever that is)
2 IBM version of LZ1 dictionary based algorithm

```
COMPRESS-CHUNK:
    MULT         CMPR-INPUT-LENGTH, INPUT-RECS, 132; /* calculate length     */
    CPYNV        INPUT-RECS, 0;                      /* reset record counter */
    CPYNV        CMPR-OUTPUT-LENGTH, 660000;         /* max = 132 * 5000     */
    CPYNV        CMPR-ALGORITHM, 2;                  /* use LZ1 algorithm    */
    CPYBWP      .CMPR-INPUT,  .INPUT-SPACE;          /* set input area       */
```

```
    CPYBWP      .CMPR-OUTPUT, .OUTPUT-SPACE;         /* set output area      */
    CPRDATA     .COMPRESS;                           /* compress the data    */
```

We have chosen the popular LZ1 algorithm (also called LZ77 after Lempel ands Ziv's 1977 landmark paper). The LZ77 algorithm is used in many popular compression packages, such as PKZip, WinZip, PNG, and ARJ. In the LZ77 approach, the dictionary is simply a portion of the previously encoded sequence. The encoder examines the input sequence through a sliding window. The window consists of two parts, a *search buffer* that contains a portion of the recently encoded sequence, and a *look-ahead* buffer that contains the next portion of the sequence to be encoded.

To encode the sequence in the look-ahead buffer, the encoder moves a search pointer back through the search buffer until it encounters a match to the first symbol in the look-ahead buffer. The distance of the pointer from the look-ahead buffer is called the *offset*. The encoder then examines the symbols at the pointer location to see if they match consecutive symbols in the look-ahead buffer. The number of consecutive symbols in the search buffer that match consecutive symbols in the look-ahead buffer, starting with the first symbol, is called the *length* of the match. Once the longest match has been found, the encoder encodes it with a triple {*offset, length*, *code*} where the code is the codeword corresponding to the symbol in the look-ahead buffer.

Write the Compressed Records

When outputting the compressed data, we'll precede each compressed chunk by a header record, that contains information about the file and of the size of the current chunk:

```
DCL SPCPTR .OUTBUF;
DCL DD OUTBUF CHAR(132) BAS(.OUTBUF);
    DCL DD OUTBUF-SYSTEM     CHAR(10) DEF(OUTBUF) POS( 1);
    DCL DD OUTBUF-LIB        CHAR(10) DEF(OUTBUF) POS(12);
    DCL DD OUTBUF-FILE       CHAR(10) DEF(OUTBUF) POS(23);
    DCL DD OUTBUF-BYTES     ZND(10,0) DEF(OUTBUF) POS(34);
    DCL DD OUTBUF-RECS      ZND(10,0) DEF(OUTBUF) POS(45);

WRITE-HEADER-RECORD:
    CPYBREP      OUTBUF, " ";
    CPYBLA       OUTBUF-SYSTEM, NWA-SYSNAME;  /* the name of the AS/400 system   */
    CPYBLA       OUTBUF-LIB,  IFCB-LIBRARY;   /* name of library for input file  */
    CPYBLA       OUTBUF-FILE, IFCB-FILE;      /* name of file being compressed   */

    CPYNV        OUTBUF-BYTES, CMPR-ACTUAL-LENGTH;  /* number of bytes in chunk  */
    ADDN         OUTPUT-BYTES, CMPR-ACTUAL-LENGTH, 131;  /* prepare to round up  */
    DIV          OUTPUT-RECS, OUTPUT-BYTES, 132;    /* compute number of records */

    CPYNV        OUTBUF-RECS, OUTPUT-RECS;
    CALLX       .SEPT(PUT-ENTRY), PUT-OPERATION, *; /* write header record       */
    CPYNV        CURRENT-REC, 0;                    /* reset record counter      */

WRITE-COMPRESSED-RECORD:                          <-----------------------.
    ADDN(S)      CURRENT-REC, 1;                                            |
    CPYBLA       OUTBUF, OUTPUT-SPACE(CURRENT-REC);                         |
    CALLX       .SEPT(PUT-ENTRY), PUT-OPERATION, *;                         |
    SUBN(SB)     OUTPUT-RECS, 1/NZER(WRITE-COMPRESSED-RECORD);   ---------'
```

If we are not yet at End-Of-File we go back and read the next batch of records to compress:

```
    CMPBLA(B)    INPUT-EOF, "Y"/NEQ(READ-UNCOMPRESSED-RECORD); -----------------'
```

Otherwise go close the files and exit the program:

```
    B            CLOSE-ALL-FILES;
…

CLOSE-ALL-FILES:
    CALLX       .SEPT(CLOSE-ENTRY), CLOSE-I, *;
    CALLX       .SEPT(CLOSE-ENTRY), CLOSE-O, *;
    RTX          *;
```

Getting the System Name

There is a handy API, **QWCRNETA**, that we can use to retrieve the name of the local system in which you are running. It is entry number 4938 in the SEPT. We call it like this (after having defined the operand list as shown below):

```
    CALLX       .SEPT(4938), QWCRNETA, *;
```

First the Network Attribute Template:

```
DCL SPCPTR .NETWORK-ATTR INIT(NETWORK-ATTR);
DCL DD      NETWORK-ATTR CHAR(32);
    DCL DD  NETWORK-ATTR-NBR    BIN(4)  DEF(NETWORK-ATTR) POS( 1);
    DCL DD  NETWORK-ATTR-OFFSET BIN(4)  DEF(NETWORK-ATTR) POS( 5);
    DCL DD  NWA-ATTR-NAME      CHAR(10) DEF(NETWORK-ATTR) POS( 9);
    DCL DD  NWA-ATTR-TYPE      CHAR(1)  DEF(NETWORK-ATTR) POS(19);
    DCL DD  NWA-ATTR-STS       CHAR(1)  DEF(NETWORK-ATTR) POS(20);
    DCL DD  NWA-ATTR-SIZE       BIN(4)  DEF(NETWORK-ATTR) POS(21);
    DCL DD  NWA-SYSNAME        CHAR(8)  DEF(NETWORK-ATTR) POS(25);

DCL SPCPTR .LENGTH-NETWORK-ATTR INIT(LENGTH-NETWORK-ATTR);
DCL DD      LENGTH-NETWORK-ATTR BIN(4) INIT(32);

DCL SPCPTR .NBR-OF-NETWORK-ATTR INIT(NBR-OF-NETWORK-ATTR);
DCL DD      NBR-OF-NETWORK-ATTR BIN(4) INIT(1);

DCL SPCPTR .NAME-NETWORK-ATTR INIT(NAME-NETWORK-ATTR);
DCL DD      NAME-NETWORK-ATTR CHAR(10) INIT("SYSNAME");

DCL SPCPTR .ERROR-CODE INIT(ERROR-CODE);
DCL DD      ERROR-CODE BIN(4) INIT(0);  /* if 0, don't handle errors */

DCL OL QWCRNETA(.NETWORK-ATTR,
                .LENGTH-NETWORK-ATTR,
                .NBR-OF-NETWORK-ATTR,
                .NAME-NETWORK-ATTR,
                .ERROR-CODE);
```

Decompressing the Data

Decompression consists of reading the compressed chunks and decompressing each one, writing the decompressed data back out. We first read a header record, then as many records as it says in the header, decompress the data with the "Decompress Data"-instruction, **DCPDATA**, write out the result, then read the next header record, etc, until all the data has been processed. The code is straightforward:

```
DECOMPRESS-FILE:
READ-COMPRESSED-HEADER:
    CALLX       .SEPT(GET-ENTRY), GET-OPERATION, *;
    CPYNV(B)     NBR-OF-RECS,  INBUF-RECS/EQ(CLOSE-ALL-FILES);
    CPYNV        NBR-OF-BYTES, INBUF-BYTES;
    CPYNV        INPUT-RECS, 0;

READ-COMPRESSED-RECORD:
    CMPNV(B)     INPUT-RECS, NBR-OF-RECS/EQ(DECOMPRESS-CHUNK);
    CALLX       .SEPT(GET-ENTRY), GET-OPERATION, *;
    ADDN(S)      INPUT-RECS, 1;
    CPYBLA       INPUT-SPACE(INPUT-RECS), INBUF;
    B            READ-COMPRESSED-RECORD;

DECOMPRESS-CHUNK:
    CPYNV        CMPR-INPUT-LENGTH,  0;
    CPYNV        CMPR-OUTPUT-LENGTH, 660000;
    CPYNV        CMPR-ALGORITHM, 0;
    CPYBWP      .CMPR-INPUT,  .INPUT-SPACE;
    CPYBWP      .CMPR-OUTPUT, .OUTPUT-SPACE;
    DCPDATA     .COMPRESS; /* uses same template as CPRDATA */

    DIV          OUTPUT-RECS, CMPR-ACTUAL-LENGTH, 132;
    CPYNV        CURRENT-REC, 0;

WRITE-UNCOMPRESSED-RECORD:                  <--------------------------.
    ADDN(S)      CURRENT-REC, 1;                                       |
    CPYBLA       OUTBUF, OUTPUT-SPACE(CURRENT-REC);                    |
    CALLX       .SEPT(PUT-ENTRY), PUT-OPERATION, *;                    |
    SUBN(SB)     OUTPUT-RECS, 1/NZER(WRITE-UNCOMPRESSED-RECORD);  ------'
    B            READ-COMPRESSED-HEADER;
```

Compression Ratio

A typical listing file had 1207 records totaling 172,032 bytes; the compressed file had 233 records totaling 40960 bytes. The compression ratio was thus 172032/40960 = 4.2, which is typical for listing text.

The Complete Program

Here is then the complete **MIFCMPS** program:

```
DCL SPCPTR .PARM1 PARM;
    DCL DD PARM-OPERATION CHAR(1) BAS(.PARM1);

DCL SPCPTR .PARM2 PARM;
    DCL DD PARM-IN-FILE CHAR(10) BAS(.PARM2);

DCL SPCPTR .PARM3 PARM;
    DCL DD PARM-IN-LIB  CHAR(10) BAS(.PARM3);

DCL SPCPTR .PARM4 PARM;
    DCL DD PARM-OUT-FILE CHAR(10) BAS(.PARM4);

DCL SPCPTR .PARM5 PARM;
    DCL DD PARM-OUT-LIB  CHAR(10) BAS(.PARM5);

DCL OL PARMS(.PARM1, .PARM2, .PARM3, .PARM4, .PARM5) EXT PARM MIN(5);

DCL SPCPTR .INPUT-SPACE INIT(INPUT-SPACE);
DCL DD      INPUT-SPACE (5000)  CHAR(132) BDRY(16);

DCL SPCPTR .OUTPUT-SPACE INIT(OUTPUT-SPACE);
DCL DD      OUTPUT-SPACE (5000) CHAR(132) BDRY(16);

DCL SPCPTR .COMPRESS INIT(COMPRESS);
DCL DD      COMPRESS CHAR(64) BDRY(16);
    DCL DD  CMPR-INPUT-LENGTH    BIN(4) DEF(COMPRESS) POS( 1);
    DCL DD  CMPR-OUTPUT-LENGTH   BIN(4) DEF(COMPRESS) POS( 5);
    DCL DD  CMPR-ACTUAL-LENGTH   BIN(4) DEF(COMPRESS) POS( 9);
    DCL DD  CMPR-ALGORITHM       BIN(2) DEF(COMPRESS) POS(13);
    DCL DD  *                  CHAR(18) DEF(COMPRESS) POS(15);
    DCL SPCPTR .CMPR-INPUT              DEF(COMPRESS) POS(33);
    DCL SPCPTR .CMPR-OUTPUT             DEF(COMPRESS) POS(49);

DCL SPCPTR .ODP;
DCL SPC     ODP  BAS(.ODP);
    DCL DD ODP-STATUS       CHAR(4)   DIR;
    DCL DD ODP-DEV-LENGTH   BIN(4)    DIR;
    DCL DD ODP-OPEN-SIZE    BIN(4)    DIR;
    DCL DD ODP.OPEN-FEEDBCK BIN(4)    DIR;
    DCL DD ODP.DEV-CTRLBLK  BIN(4)    DIR;
    DCL DD ODP.IO-FEEDBACK  BIN(4)    DIR;
    DCL DD ODP.LOCK-LIST    BIN(4)    DIR;
    DCL DD ODP.SPOOL-OUTPUT BIN(4)    DIR;

    DCL DD ODP.MBR-DESCR    BIN(4)    DIR;
    DCL DD ODP.CUR-IN-REC   BIN(4)    DIR;
    DCL DD ODP.CUR-OUT-REC  BIN(4)    DIR;
    DCL DD ODP.OPEN-DMCQ    BIN(4)    DIR;
    DCL DD ODP.OUTSTANDINGS BIN(4)    DIR;
    DCL DD *                CHAR(12)  DIR;

    DCL SYSPTR .ODP-CURSOR            DIR;
    DCL SPCPTR *                      DIR;
    DCL SPCPTR .ODP-CDM-ERROR         DIR;
    DCL SPCPTR .ODP-INPUT-BUFFER      DIR;
    DCL SPCPTR .ODP-OUTPUT-BUFFER     DIR;

    DCL DD ODP.CDM-CLOSING  BIN(2)    DIR;
    DCL DD ODP-DEV-NAME-IDX BIN(2)    DIR;
    DCL DD ODP-NBR-OF-DEVS  BIN(2)    DIR;

    DCL DD ODP-SEQUENCE-NBR BIN(4)    DIR;
    DCL DD ODP-REC-LENGTH   BIN(2)    DIR;
    DCL DD ODP-REC-LENGTH2  BIN(2)    DIR;
    DCL DD ODP-NBR-OF-*RDS  BIN(2)    DIR;
    DCL DD ODP-RELEASE-NBR  BIN(2)    DIR;
    DCL DD ODP-OPEN-POSN    CHAR(1)   DIR;
    DCL DD ODP-OVR-REC-LEN  BIN(2)    DIR;
    DCL DD ODP-COM-DEV-CNT  BIN(2)    DIR;
```

```
    DCL DD ODP.INPUT-BPCA   BIN(4)    DIR;
    DCL DD ODP.OUTPUT-BPCA  BIN(4)    DIR;
    DCL DD ODP............  CHAR(1)   DIR;

DCL SPCPTR .DEV-CONTROL-BLOCK;
DCL SPC     DEV-CONTROL-BLOCK BAS(.DEV-CONTROL-BLOCK);
    DCL DD  DCB-MAX-NBR-OF-DEVICES   BIN( 2) DIR;
    DCL DD  DCB-DEVICES-IN-THE-ODP   BIN( 2) DIR;
    DCL DD  DCB-DEVICE-NAME         CHAR(10) DIR;
    DCL DD  DCB-OFFSET-TO-FM-WORK    BIN( 4) DIR;
    DCL DD  DCB-LENGTH-OF-FM-WORK    BIN( 4) DIR;
    DCL DD  DCB-INDEX-FOR-LUD-PTR    BIN( 2) DIR;
    DCL DD  DCB-GET                  BIN( 2) DIR;
    DCL DD  DCB-GET-BY-RRN           BIN( 2) DIR;
    DCL DD  DCB-GET-BY-KEY           BIN( 2) DIR;
    DCL DD  *                        BIN( 2) DIR;
    DCL DD  DCB-PUT                  BIN( 2) DIR;
    DCL DD  DCB-PUT-GET              BIN( 2) DIR;
    DCL DD  DCB-UPDATE               BIN( 2) DIR;
    DCL DD  DCB-DELETE               BIN( 2) DIR;
    DCL DD  DCB-FORCE-EOD            BIN( 2) DIR;
    DCL DD  DCB-FORCE-EOV            BIN( 2) DIR;
    DCL DD  DCB-COMMIT               BIN( 2) DIR;
    DCL DD  DCB-ROLLBACK             BIN( 2) DIR;
    DCL DD  DCB-FREE-REC-LOCK        BIN( 2) DIR;
    DCL DD  *                        BIN( 2) DIR;
    DCL DD  *                        BIN( 2) DIR;
    DCL DD  *                        BIN( 2) DIR;
    DCL DD  DCB-CLOSE                BIN( 2) DIR;
    DCL DD  DCB-OPEN                 BIN( 2) DIR;
    DCL DD  *                        BIN( 2) DIR;
    DCL DD  *                        BIN( 2) DIR;

/* THE I/O IS DONE BY USING THE CALLX INSTRUCTION REFERENCING */
/* A SYSTEM POINTER  THAT IS OBTAINED  FROM  THE ENTRY POINT  */
/* TABLE.  THE ENTRY POINT TABLE CONTAINS PRE-RESOLVED SYSTEM */
/* POINTERS (THOUSANDS...).  THE SYSTEM ENTRY POINT TABLE     */
/* IS ADDRESSED BY THE POINTER BASED ON THE PROCESS COMMUNI-  */
/* CATION OBJECT (PCO):                                       */
/*   PCO POINTER --> POINTER TO SEPT --> PTR TO OS FUNCTION 1 */
/*                                       PTR TO OS FUNCTION 2 */
/*                                       ...                  */

DCL SYSPTR .SEPT(6440) BAS(@SEPT);
DCL DD PCO CHAR(80) BASPCO;
   DCL SPCPTR @SEPT DEF(PCO) POS( 1);

/* THE USER FILE CONTROL BLOCK (UFCB) DEFINES THE FILE NAME,  */
/* BUFFER SPACES AND ALL NECESSARY CONTROL INFORMATION NEEDED */
/* TO MANAGE THE FILE.  IT ALSO PROVIDES THE FEEDBACKS NEEDED */
/* TO ACCESS VARIOUS STRUCTURES, SUCH AS THE ODP  (THE OPEN   */
/* DATA PATH).                                                */

DCL DD IFCB CHAR(214) BDRY(16);
    DCL SPCPTR .IFCB-ODP               DEF(IFCB) POS(  1);
    DCL SPCPTR .IFCB-INBUF             DEF(IFCB) POS( 17);
    DCL SPCPTR .IFCB-OUTBUF            DEF(IFCB) POS( 33);
    DCL SPCPTR .IFCB-OPEN-FEEDBACK     DEF(IFCB) POS( 49);
    DCL SPCPTR .IFCB-IO-FEEDBACK       DEF(IFCB) POS( 65);
    DCL SPCPTR .IFCB-NEXT-UFCB         DEF(IFCB) POS( 81);

    DCL DD *                  CHAR(32) DEF(IFCB) POS( 97);
    DCL DD IFCB-FILE          CHAR(10) DEF(IFCB) POS(129);
    DCL DD IFCB-LIB-ID        BIN(2)   DEF(IFCB) POS(139);
    DCL DD IFCB-LIBRARY       CHAR(10) DEF(IFCB) POS(141);
    DCL DD IFCB-MBR-ID        BIN(2)   DEF(IFCB) POS(151);
    DCL DD IFCB-MEMBER        CHAR(10) DEF(IFCB) POS(153);

    DCL DD IFCB-DEVICE-NAME   CHAR(10) DEF(IFCB) POS(163);
    DCL DD IFCB-DEVICE-INDEX  BIN(2)   DEF(IFCB) POS(173);

    DCL DD IFCB-FLAGS-1       CHAR(1)  DEF(IFCB) POS(175) INIT(X'80');
    DCL DD IFCB-FLAGS-2       CHAR(1)  DEF(IFCB) POS(176) INIT(X'20');

    DCL DD IFCB-REL-VERSION   CHAR(4)  DEF(IFCB) POS(177);
    DCL DD IFCB-INV-MK-COUNT  BIN (4)  DEF(IFCB) POS(181);
    DCL DD IFCB-MORE-FLAGS    CHAR(1)  DEF(IFCB) POS(185);
    DCL DD *                  CHAR(23) DEF(IFCB) POS(186);

    DCL DD IFCB-LENGTH-ID     BIN (2)  DEF(IFCB) POS(209) INIT(1);
```

```
    DCL DD IFCB-RECORD-LENGTH BIN (2)  DEF(IFCB) POS(211) INIT(132);
    DCL DD IFCB-NO-MORE-PARMS BIN (2)  DEF(IFCB) POS(213) INIT(32767);

DCL SPCPTR .IFCB INIT(IFCB);
DCL OL  OPEN-I(.IFCB);
DCL OL CLOSE-I(.IFCB);

DCL DD OFCB CHAR(214) BDRY(16);
    DCL SPCPTR .OFCB-ODP               DEF(OFCB) POS(  1);
    DCL SPCPTR .OFCB-INBUF             DEF(OFCB) POS( 17);
    DCL SPCPTR .OFCB-OUTBUF            DEF(OFCB) POS( 33);
    DCL SPCPTR .OFCB-OPEN-FEEDBACK     DEF(OFCB) POS( 49);
    DCL SPCPTR .OFCB-IO-FEEDBACK       DEF(OFCB) POS( 65);
    DCL SPCPTR .OFCB-NEXT-UFCB         DEF(OFCB) POS( 81);

    DCL DD *                  CHAR(32) DEF(OFCB) POS( 97);
    DCL DD OFCB-FILE          CHAR(10) DEF(OFCB) POS(129);
    DCL DD OFCB-LIB-ID        BIN(2)   DEF(OFCB) POS(139);
    DCL DD OFCB-LIBRARY       CHAR(10) DEF(OFCB) POS(141);
    DCL DD OFCB-MBR-ID        BIN(2)   DEF(OFCB) POS(151);
    DCL DD OFCB-MEMBER        CHAR(10) DEF(OFCB) POS(153);

    DCL DD OFCB-DEVICE-NAME   CHAR(10) DEF(OFCB) POS(163);
    DCL DD OFCB-DEVICE-INDEX  BIN(2)   DEF(OFCB) POS(173);

    DCL DD OFCB-FLAGS-1       CHAR(1)  DEF(OFCB) POS(175) INIT(X'80');
    DCL DD OFCB-FLAGS-2       CHAR(1)  DEF(OFCB) POS(176) INIT(X'10');

    DCL DD OFCB-REL-VERSION   CHAR(4)  DEF(OFCB) POS(177);
    DCL DD OFCB-INV-MK-COUNT  BIN (4)  DEF(OFCB) POS(181);
    DCL DD OFCB-MORE-FLAGS    CHAR(1)  DEF(OFCB) POS(185);
    DCL DD *                  CHAR(23) DEF(OFCB) POS(186);

    DCL DD OFCB-LENGTH-ID     BIN (2)  DEF(OFCB) POS(209) INIT(1);
    DCL DD OFCB-RECORD-LENGTH BIN (2)  DEF(OFCB) POS(211) INIT(132);
    DCL DD OFCB-NO-MORE-PARMS BIN (2)  DEF(OFCB) POS(213) INIT(32767);

DCL SPCPTR .OFCB INIT(OFCB);
DCL OL  OPEN-O(.OFCB) ARG;
DCL OL CLOSE-O(.OFCB) ARG;

DCL DD  GET-ENTRY   BIN(2);
DCL DD  PUT-ENTRY   BIN(2);

DCL CON CLOSE-ENTRY BIN(2)   INIT(11);
DCL CON OPEN-ENTRY  BIN(2)   INIT(12);
DCL CON *LIBL       BIN(2)   INIT(-75); /* S/38: -72 */
DCL CON *FIRST      BIN(2)   INIT(-71); /* S/38: -73 */
DCL CON THE-LIB     BIN(2)   INIT(72);
DCL CON THE-MBR     BIN(2)   INIT(73);

DCL EXCM * EXCID(H'5001') BP(EOF-DETECTED) CV("CPF") IMD;

DCL SPCPTR .INBUF;
DCL DD INBUF CHAR(132) BAS(.INBUF);
    DCL DD INBUF-SYSTEM     CHAR(10) DEF(INBUF) POS( 1);
    DCL DD INBUF-LIB        CHAR(10) DEF(INBUF) POS(12);
    DCL DD INBUF-FILE       CHAR(10) DEF(INBUF) POS(23);
    DCL DD INBUF-BYTES     ZND(10,0) DEF(INBUF) POS(34);
    DCL DD INBUF-RECS      ZND(10,0) DEF(INBUF) POS(45);

DCL SPCPTR .OUTBUF;
DCL DD OUTBUF CHAR(132) BAS(.OUTBUF);
    DCL DD OUTBUF-SYSTEM     CHAR(10) DEF(OUTBUF) POS( 1);
    DCL DD OUTBUF-LIB        CHAR(10) DEF(OUTBUF) POS(12);
    DCL DD OUTBUF-FILE       CHAR(10) DEF(OUTBUF) POS(23);
    DCL DD OUTBUF-BYTES     ZND(10,0) DEF(OUTBUF) POS(34);
    DCL DD OUTBUF-RECS      ZND(10,0) DEF(OUTBUF) POS(45);

DCL SPCPTR .NULL;

DCL DD GET-OPTION BIN(4) INIT(H'03000001');
DCL SPCPTR .GET-OPTION INIT(GET-OPTION);
DCL OL GET-OPERATION(.IFCB, .GET-OPTION, .NULL);

DCL DD PUT-OPTION BIN(4) INIT(H'10000005');
DCL SPCPTR .PUT-OPTION INIT(PUT-OPTION);
DCL OL PUT-OPERATION(.OFCB, .PUT-OPTION, .NULL);

DCL DD INPUT-EOF   CHAR(1);
DCL DD INPUT-RECS   BIN(4);
```

```
DCL DD CURRENT-REC  BIN(4);
DCL DD OUTPUT-RECS  BIN(4);
DCL DD OUTPUT-BYTES BIN(4);
DCL DD NBR-OF-RECS  BIN(4);
DCL DD NBR-OF-BYTES BIN(4);

DCL SPCPTR .NETWORK-ATTR INIT(NETWORK-ATTR);
DCL DD      NETWORK-ATTR CHAR(32);
    DCL DD  NETWORK-ATTR-NBR    BIN(4)  DEF(NETWORK-ATTR) POS( 1);
    DCL DD  NETWORK-ATTR-OFFSET BIN(4)  DEF(NETWORK-ATTR) POS( 5);
    DCL DD  NWA-ATTR-NAME      CHAR(10) DEF(NETWORK-ATTR) POS( 9);
    DCL DD  NWA-ATTR-TYPE      CHAR(1)  DEF(NETWORK-ATTR) POS(19);
    DCL DD  NWA-ATTR-STS       CHAR(1)  DEF(NETWORK-ATTR) POS(20);
    DCL DD  NWA-ATTR-SIZE       BIN(4)  DEF(NETWORK-ATTR) POS(21);
    DCL DD  NWA-SYSNAME        CHAR(8)  DEF(NETWORK-ATTR) POS(25);

DCL SPCPTR .LENGTH-NETWORK-ATTR INIT(LENGTH-NETWORK-ATTR);
DCL DD      LENGTH-NETWORK-ATTR BIN(4) INIT(32);

DCL SPCPTR .NBR-OF-NETWORK-ATTR INIT(NBR-OF-NETWORK-ATTR);
DCL DD      NBR-OF-NETWORK-ATTR BIN(4) INIT(1);

DCL SPCPTR .NAME-NETWORK-ATTR INIT(NAME-NETWORK-ATTR);
DCL DD      NAME-NETWORK-ATTR CHAR(10) INIT("SYSNAME");

DCL SPCPTR .ERROR-CODE INIT(ERROR-CODE);
DCL DD      ERROR-CODE BIN(4) INIT(0);

DCL OL QWCRNETA(.NETWORK-ATTR,
                .LENGTH-NETWORK-ATTR,
                .NBR-OF-NETWORK-ATTR,
                .NAME-NETWORK-ATTR,
                .ERROR-CODE);

/*************************************************************/

ENTRY * (PARMS) EXT;
    CALLX       .SEPT(4938), QWCRNETA, *;
    CPYBWP      .NULL, *; /* MAKE NULL PTR */

OPEN-INPUT-FILE:
    CPYBLA       IFCB-FILE,    PARM-IN-FILE;
    CPYNV        IFCB-LIB-ID,  THE-LIB;
    CPYBLA       IFCB-LIBRARY, PARM-IN-LIB;
    CPYNV        IFCB-MBR-ID,  THE-MBR;
    CPYBLA       IFCB-MEMBER,  PARM-IN-FILE;
    CALLX       .SEPT(OPEN-ENTRY), OPEN-I, *;

    CPYBWP      .INBUF, .IFCB-INBUF;
    CPYBWP      .ODP, .IFCB-ODP;
    ADDSPP      .DEV-CONTROL-BLOCK, .ODP, ODP.DEV-CTRLBLK;
    CPYNV        GET-ENTRY, DCB-GET;
    CPYNV        INPUT-RECS, 0;
    CPYBLA       INPUT-EOF, " ";

OPEN-OUTPUT-FILE:
    CPYBLA       OFCB-FILE,    PARM-OUT-FILE;
    CPYNV        OFCB-LIB-ID,  THE-LIB;
    CPYBLA       OFCB-LIBRARY, PARM-OUT-LIB;
    CPYNV        OFCB-MBR-ID,  THE-MBR;
    CPYBLA       OFCB-MEMBER,  PARM-OUT-FILE;
    CALLX       .SEPT(OPEN-ENTRY), OPEN-O, *;

    CPYBWP      .OUTBUF, .OFCB-OUTBUF;
    CPYBWP      .ODP, .OFCB-ODP;
    ADDSPP      .DEV-CONTROL-BLOCK, .ODP, ODP.DEV-CTRLBLK;
    CPYNV        PUT-ENTRY, DCB-PUT;

    CMPBLA(B)    PARM-OPERATION, "D"/EQ(DECOMPRESS-FILE);

COMPRESS-FILE:
READ-UNCOMPRESSED-RECORD:
    CMPNV(B)     INPUT-RECS, 5000/EQ(COMPRESS-CHUNK);
    CALLX       .SEPT(GET-ENTRY), GET-OPERATION, *;
    ADDN(S)      INPUT-RECS, 1;
    CPYBLA       INPUT-SPACE(INPUT-RECS), INBUF;
    B            READ-UNCOMPRESSED-RECORD;

EOF-DETECTED:
    CPYBLA       INPUT-EOF, "Y";
    CMPBLA(B)    PARM-OPERATION, "D"/EQ(CLOSE-ALL-FILES);
```

```
COMPRESS-CHUNK:
    MULT         CMPR-INPUT-LENGTH, INPUT-RECS, 132;
    CPYNV        INPUT-RECS, 0;
    CPYNV        CMPR-OUTPUT-LENGTH, 660000;
    CPYNV        CMPR-ALGORITHM, 2;
    CPYBWP      .CMPR-INPUT,  .INPUT-SPACE;
    CPYBWP      .CMPR-OUTPUT, .OUTPUT-SPACE;
    CPRDATA     .COMPRESS;

WRITE-HEADER-RECORD:
    CPYBREP      OUTBUF, " ";
    CPYBLA       OUTBUF-SYSTEM, NWA-SYSNAME;
    CPYBLA       OUTBUF-LIB,  IFCB-LIBRARY;
    CPYBLA       OUTBUF-FILE, IFCB-FILE;
    CPYNV        OUTBUF-BYTES, CMPR-ACTUAL-LENGTH;

    ADDN         OUTPUT-BYTES, CMPR-ACTUAL-LENGTH, 131;
    DIV          OUTPUT-RECS, OUTPUT-BYTES, 132;
    CPYNV        OUTBUF-RECS, OUTPUT-RECS;
    CALLX       .SEPT(PUT-ENTRY), PUT-OPERATION, *;
    CPYNV        CURRENT-REC, 0;

WRITE-COMPRESSED-RECORD:
    ADDN(S)      CURRENT-REC, 1;
    CPYBLA       OUTBUF, OUTPUT-SPACE(CURRENT-REC);
    CALLX       .SEPT(PUT-ENTRY), PUT-OPERATION, *;
    SUBN(SB)     OUTPUT-RECS, 1/NZER(WRITE-COMPRESSED-RECORD);
    CMPBLA(B)    INPUT-EOF, "Y"/NEQ(READ-UNCOMPRESSED-RECORD);

CLOSE-ALL-FILES:
    CALLX       .SEPT(CLOSE-ENTRY), CLOSE-I, *;
    CALLX       .SEPT(CLOSE-ENTRY), CLOSE-O, *;
    RTX          *;

DECOMPRESS-FILE:
READ-COMPRESSED-HEADER:
    CALLX       .SEPT(GET-ENTRY), GET-OPERATION, *;
    CPYNV(B)     NBR-OF-RECS,  INBUF-RECS/EQ(CLOSE-ALL-FILES);
    CPYNV        NBR-OF-BYTES, INBUF-BYTES;
    CPYNV        INPUT-RECS, 0;
READ-COMPRESSED-RECORD:
    CMPNV(B)     INPUT-RECS, NBR-OF-RECS/EQ(DECOMPRESS-CHUNK);
    CALLX       .SEPT(GET-ENTRY), GET-OPERATION, *;
    ADDN(S)      INPUT-RECS, 1;
    CPYBLA       INPUT-SPACE(INPUT-RECS), INBUF;
    B            READ-COMPRESSED-RECORD;

DECOMPRESS-CHUNK:
    CPYNV        CMPR-INPUT-LENGTH,  0;
    CPYNV        CMPR-OUTPUT-LENGTH, 660000;
    CPYNV        CMPR-ALGORITHM, 0;
    CPYBWP      .CMPR-INPUT,  .INPUT-SPACE;
    CPYBWP      .CMPR-OUTPUT, .OUTPUT-SPACE;
    DCPDATA     .COMPRESS;

    DIV          OUTPUT-RECS, CMPR-ACTUAL-LENGTH, 132;
    CPYNV        CURRENT-REC, 0;
WRITE-UNCOMPRESSED-RECORD:
    ADDN(S)      CURRENT-REC, 1;
    CPYBLA       OUTBUF, OUTPUT-SPACE(CURRENT-REC);
    CALLX       .SEPT(PUT-ENTRY), PUT-OPERATION, *;
    SUBN(SB)     OUTPUT-RECS, 1/NZER(WRITE-UNCOMPRESSED-RECORD);
    B            READ-COMPRESSED-HEADER;
```

Calculating Archimedes' Constant, π

An Amazing Formula for π

People have been calculating the value of the ratio of a circle's perimeter to its diameter for millennia. In the 3rd century B.C., Archimedes (287 B.C. - 212 B.C.) considered inscribed and circumscribed regular polygons of 96 sides and deduced that $3^{10}/_{71} < \pi < 3^{1}/_{7}$. Today hundreds of billions of digits of π are known. Yet, people keep on calculating π. No book about computing is complete without rehashing this subject. Even my old S/38 MI-assembler manual exhibits a program to calculate π. In keeping with that proud tradition we'll include one here as well. Tremendous progress has occurred in the 25 years since the S/38 appeared. The speed of our hardware has increased by a factor of many thousands. What is often less appreciated is that many algorithms have been improved by an even greater factor. The best algorithm for calculating π being no exception. The current record-holder is the following quartically (the number of correct digits *quadruples* with each iteration) convergent algorithm, which is related to Ramanujan's work on elliptic integrals:

$$\alpha_0 = 6 - 4\cdot 2^{1/2}, \quad z_0 = 2^{1/2} - 1$$
$$z_{n+1} = (1 - (1 - z_n^4)^{1/4})/(1 + (1 - z_n^4)^{1/4})$$
$$\alpha_{n+1} = (1 + z_{n+1})^4 \alpha_n - 2^{2n+3} z_{n+1} [1 + z_{n+1} + z_{n+1}^2]$$
$$1/\alpha_n \Rightarrow \pi \text{ as } n \Rightarrow \infty$$

This algorithm is the basis for Kanada's record-breaking evaluation of π to over 200 billion digits. More information about this and other related algorithms can be found at http://www.mathsoft.com/asolve/constant/pi/pi.html. The first 10,000 digits of π can be found at http://www.lacim.uqam.ca/piDATA/pi.html. Here are the first thirty-five digits: 3.14159 26535 89793 23846 26433 83279 50288…

We'll present two implementations, one using packed decimal numbers and one using double-precision floating-point numbers. As a final touch, we'll show the code from the old S/38 MI-assembler manual.

Computing the Square Root

The algorithm contains several places where a square root must be computed. Firstly in the initial values where $2^{1/2}$ is needed, but more importantly in the calculation of z_{n+1} where the fourth root (the square root of the square root) is called for. The square root must be computed correctly. Any error here propagates into the remainder of the calculation and is not reduced during the iteration.

The square root of N is calculated using Newton's iterative method with six iterations:

$s_0 = 1$ (initial guess)

$s_{n+1} = (s_n + N/s_n)/2$

We use packed numbers with maximal precision (31 digits):

```
DCL DD N    PKD(31,30);
DCL DD SQRT PKD(31,30); /* SQRT = square root of N */

DCL DD K BIN(2);
DCL DD WRK PKD(31,30);
DCL INSPTR .GET-SQUARE-ROOT;
ENTRY       GET-SQUARE-ROOT INT;
    CPYNV         SQRT, 1; /* initial guess  */
    CPYNV         K, 6;    /* six iterations */
NEWTON-SQRT-ITERATION:
    DIV           WRK, N, SQRT; /* NO ROUNDING */
    ADDN(S)       SQRT, WRK;
    DIV(SR)       SQRT, 2;  /* ROUNDING MATTERS */
    SUBN(SB)      K, 1/HI(NEWTON-SQRT-ITERATION);
    B            .GET-SQUARE-ROOT;
```

It is important for maximum accuracy that the division by 2 is performed with rounding.

The *MIPIPKD* Program

We'll iterate through the algorithm thrice, although, as we shall see, two iterations already give us the best approximation that we are going to obtain with the accuracy chosen:

```
DCL DD P  PKD(31, 0); /* power of 2 */
DCL DD Y  PKD(31,30);
DCL DD A  PKD(31,30);
DCL DD B  PKD(31,30);
DCL DD C  PKD(31,30);

    CPYNV        P, 4;
    CPYNV        N, 2;
    CALLI        GET-SQUARE-ROOT, *, .GET-SQUARE-ROOT;

    SUBN         Y, SQRT, 1;
    MULT         B, SQRT, 4;
    SUBN         A, 6, B;

    CPYNV        M, 3; /* iterate 3 times */
:   CALLI        ITERATE-PI, *, .ITERATE-PI;
    SUBN(SB)     M, 1/HI(=-1);
    RTX          *;

DCL DD M  BIN(2);
DCL DD PI ZND(31,30);
DCL INSPTR .ITERATE-PI;
ENTRY       ITERATE-PI INT;
    MULT         B, Y, Y;
    MULT(S)      B, B;
    SUBN         N, 1, B;
    CALLI        GET-SQUARE-ROOT, *, .GET-SQUARE-ROOT;
    CPYNV        N, SQRT;
    CALLI        GET-SQUARE-ROOT, *, .GET-SQUARE-ROOT;
    CPYNV        B, SQRT;

    SUBN         Y, 1, B;
    ADDN         C, 1, B;
    DIV(SR)      Y, C;   /* ROUNDING IS IMPORTANT */

    ADDN         B, 1, Y;
    MULT         C, Y, Y;
    ADDN(S)      C, B;
    MULT(S)      C, Y;
    MULT(S)      C, P;

    MULT(S)      P, 4;
    MULT(S)      B, B;
    MULT(S)      B, B;
    MULT(S)      A, B;

    SUBN(S)      A, C;
    SUBN(S)      A, C;

    DIV(R)       PI, 1, A;
    CPYBLAP      MSG-TEXT, PI, " ";
    CALLI        SHOW-MESSAGE, *, .SHOW-MESSAGE;
    B           .ITERATE-PI;

%INCLUDE SHOWMSG
```

The results after each of the three iterations are:

3.**14159 26462 13542 28214 93444 32024** ; 7 correct decimals
3.**14159 26535 89793 23846 26433 83260** ; 28 correct decimals
3.14159 26535 89793 23846 26433 83260 ; no change

If our machine could have handled it (or if we had programmed our own arithmetical operations on very long numbers), the third iteration would have given us 112 correct decimals, the next iteration 448, … This is a truly remarkable performance.

The *MIPIFLT* Program

And now for the floating-point version. We can use the square root function directly supported by the CMF1-instruction:

```
DCL DD N    FLT(8);
DCL DD SQRT FLT(8);

DCL DD P  FLT(8);
DCL DD Y  FLT(8);
DCL DD A  FLT(8);
DCL DD B  FLT(8);
DCL DD C  FLT(8);

    CPYNV        P, 4;
    CPYNV        N, 2;
    CMF1         SQRT, X'0020', N;

    SUBN         Y, SQRT, 1;
    MULT         B, SQRT, 4;
    SUBN         A, 6, B;

    CPYNV        M, 3;
:   CALLI        ITERATE-PI, *, .ITERATE-PI;
    SUBN(SB)     M, 1/HI(=-1);

    RTX          *;

DCL DD M  BIN(2);
DCL DD PI FLT(8);
DCL INSPTR .ITERATE-PI;
ENTRY       ITERATE-PI INT;
    MULT         B, Y, Y;
    MULT(S)      B, B;
    SUBN         N, 1, B;
    CMF1         SQRT, X'0020', N;
    CMF1         B,    X'0020', SQRT;

    SUBN         Y, 1, B;
    ADDN         C, 1, B;
    DIV(S)       Y, C;

    ADDN         B, 1, Y;
    MULT         C, Y, Y;
    ADDN(S)      C, B;
    MULT(S)      C, Y;
    MULT(S)      C, P;

    MULT(S)      P, 4;
    MULT(S)      B, B;
    MULT(S)      B, B;
    MULT(S)      A, B;

    SUBN(S)      A, C;
    SUBN(S)      A, C;

    DIV          PI, 1, A;
    CPYNV        QQ, PI;
    CPYBLAP      MSG-TEXT, QQ, " ";
    CALLI        SHOW-MESSAGE, *, .SHOW-MESSAGE;
    B           .ITERATE-PI;

DCL DD QQ ZND(31,30);
%INCLUDE SHOWMSG
```

Apart from simply changing the data type from `PKD` to `FLT`, the only real difference is that we cannot use the rounding options with floating-point instructions. Here is the result:

3.**14159 26462 13546 83555 40979 24614** ; 7 correct
3.**14159 26535 89809 54729 87279 20861** ; only 12 correct
3.14159 26535 89809 54729 87279 20861 ; no change

Even though the algorithm is quartic (so we should get four times as many correct digits in each iteration) we only get 12 correct decimals in the second iteration because the accuracy of floating-point numbers is only about 13-14 decimal digits.

The Original S/38 Program, MIPIS38

The MI-assembler program from page 62 of my old S/38 notes (manual is too big a word) used the following formula:

$$\pi/4 = \arctan(1/7) + 2\arctan(1/3)$$

where arctan(x) is calculated from the series expansion:

$$\arctan(x) = x - x^3/3 + x^5/5 - x^7/7 + x^9/9\ldots$$

Here is the program with its *original* orthography intact:

```
 DCL DD X     FLT(8)  AUTO INIT(E'+1.0E00')                    ;
 DCL DD Y     FLT(8)  AUTO INIT(E'7')                          ;
 DCL DD XS    FLT(8)  AUTO                                     ;
 DCL DD A     FLT(8)  AUTO INIT(E'1')                          ;
 DCL DD DENOM FLT(8)  AUTO INIT(E'1.0')                        ;
 DCL DD SUM   FLT(8)  AUTO                                     ;

 DCL DD TEMP  FLT(8)                                           ;
 DCL DD FOUR  FLT(8)  INIT(E'4.0E0')                           ;
 DCL DD PI    FLT(8)                                           ;

 DIV(S) X,3;               /* X = 1/3                        */
 DIV Y,1,Y;                /* Y = 1/7                        */
 MULT XS,X,X;              /* XS = X**2                      */
 NEG(S)   XS;
 ADDN A,X,0;

LOOP1:
 MULT(S)   X,XS    ;
 ADDN(S )  DENOM,2           ;
 DIV      TEMP,X,DENOM;
 ADDN(S) A   , TEMP;
 CMPNV(B)  DENOM,25 /LO(LOOP1),EQ(LOOP1) ;

 MULT XS,Y,Y            ;
 NEG(S)   XS;
 ADDN SUM,Y,0;
 ADDN DENOM,0,1;
LOOP2:                   /* LOOP TO CALCULATE ATN(1/7) */
 MULT(S) Y,XS          ;
 ADDN(S) DENOM,2        ;
 DIV TEMP,Y,DENOM        ;
 ADDN(S) SUM,TEMP         ;
 CMPNV(B) DENOM,25 /LO(LOOP2),EQ(LOOP2)  ;
 ADDN PI,A,A;
 ADDN(S) PI,SUM;
 MULT(S) PI,FOUR;

 CPYNV ZZ, PI;
 CPYBLAP MSG-TEXT, ZZ, " ";
 CALLI SHOW-MESSAGE, *, .SHOW-MESSAGE;
 RTX *;

 DCL DD ZZ ZND(31,30);
%INCLUDE SHOWMSG
```

with the result

3.**14159 26535 89789 56328 42846 68043**

correct to 13 decimals. If we have learnt nothing else in the intervening 17 years it would be to appreciate the usefulness (indeed, necessity) of a neat program layout:

"Ugly programs are like ugly suspension bridges: they're much more liable to collapse than pretty ones, because the way humans perceive beauty is intimately related to our ability to process and understand complexity."
- Eric S. Raymond

Exception Handling

Exceptions and Events

At the hardware level, the processor deals with exceptions by issuing *interrupts*. An exception at this level is a condition (note how we keep hiding the real world behind yet another concept) that calls for changing the normal flow of instruction execution. At the MI-level, there is no concept of interrupts. MI distinguishes between *exceptions* and *events*. An exception at the MI is not the same as an exception at the hardware level (although the latter may cause the former), but is rather a 'formally architected process message' (F. Soltis).

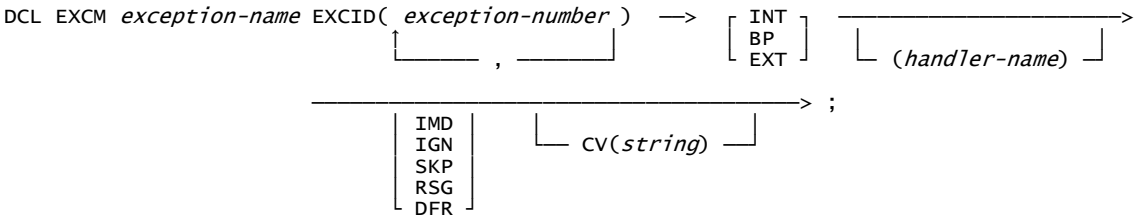
An MI-exception is defined as either a machine-defined error detected during the execution of an instruction, or as a user-defined condition detected by a user-program. An event, on the other hand, is defined as an activity that occurs during machine operation that may be of interest to machine users. Exceptions are synchronous, meaning that they are caused by the execution of an instruction, while events are asynchronous, meaning that they are caused by actions outside the currently executing instruction. An example of a synchronous exception is an attempt to divide by zero. An example of an asynchronous event is the completion of an I/O operation.

Just as there are two types of exceptions (errors and user-defined conditions), there are also two types of events: object-related events and machine-related events. An MI-process can monitor the occurrence of a set of events and take appropriate action on some or all of them. Exceptions can also be monitored. Multiple monitors can be enabled at the same time. Each monitor has a priority controlling exception searching and handling when more than one monitor is active. Associated with each monitor is an exception handling routine; exceptions are formatted as messages and sent to the appropriate queue space.

In this chapter we shall only consider exceptions. Events are dealt with in later chapters. Superficially however, there are many similarities between exceptions and events.

Declaring an Exception Monitor

An exception monitor is installed by declaring an *Exception Description*. The syntax is as follows:



where

Element	Range	Description
exception-name	MI-name or *	Name of exception description
exception-number	Unsigned integer from 0 to 65535	Exception identifier
handler-name	MI-name	Name of exception handler
string	1 to 32 bytes	Compare value

Handler type	Description
INT	Name refers to an internal entry point
BP	Name is label for branch point
EXT	Name refers to system pointer for external program or entry point

The *exception action* determines what the system does when the exception is encountered:

Handler action	Description
IMD	Control passes immediately to the specified exception handler (default)
IGN	Ignore the exception and continue processing
SKP	Skip the current description, continue search for another description
RSG	Re-signal the exception and continue to search for a description
DFR	Postpone (defer) handling, saving exception data for later

The Exception Identifier

Two bytes combined into an unsigned integer serve to identify an exception. The first byte is a *group* number, while the second byte is a subtype within the group. As further qualification the exception may have a *compare value*. Although compare values may contain up to 32 characters, OS/400 only uses three letter combinations, such as "CPF" or "MCH". As exceptions are 'architected' as messages, they are assigned a textual message identifier consisting of the three-letter compare value followed by a four-digit *hexadecimal* (see below, though) representation of the exception identifier, such as "**CPF3CF1**" or "**MCH1202**". Exceptions signaled by the machine have an internal compare value consisting of four null characters; although these are translated into the three-letter (more readable) value "MCH". By historical accident (bummer…), machine exception IDs are converted byte for byte into *decimal* value message IDs. This silly convention has caused much grief and gnashing of teeth. For example, the dreaded "decimal data error" message we all hate and know as MCH**1202** is really MCH**0C02**, so declaring an exception description to monitor for "decimal data error" (MCH1202), "zero divide" (MCH1210) and "size too small for result" (MCH1211) would look like this:

```
DCL EXCM DATA-ERROR EXCID(H'0C02', H'0C0A', H'0C0B') INT(ERROR) IMD CV("MCH");
```

An exception description may monitor for an exception with a *generic* ID as follows:

H'0000' - Any exception ID results in a match.
H'*gg*00' - Any exception ID in group *gg* results in a match.
H'*ggnn*' - The exception ID must match exactly *ggnn* in order for a match to occur.

Searching for Exception Descriptions

When an exception occurs, the exception descriptions of the current invocation are searched in the sequence in which they were declared. If an exception ID in an exception description corresponds to the occurring exception, the corresponding compare values are checked. If the compare value *length* in the exception description is less than the compare value length of the exception occurring, the length of the compare value in the exception description is used for matching purposes. If it is greater, an automatic mismatch results. As machine exception compare values have a length of four bytes, the three-character compare values used by OS/400 just squeak by.

Materialize an Exception Description

You can materialize a named exception description by using the **MATEXCPD** instruction. The format is:

```
MATEXCPD    .Receiver, Exception-description, Materialization-option;
```

The *.Receiver* operand is a space pointer to the materialization template. The materialization option is a one-character data item with the value x'00' meaning full materialization and higher values meaning partial materialization to various degrees. Here is the layout of the full template:

```
DCL SPCPTR .EXCP-DESCR INIT(EXCP-DESCR);
DCL DD      EXCP-DESCR CHAR(96) BDRY(16);
    DCL DD  EXCP-DESCR-PROV  BIN(4) DEF(EXCP-DESCR) POS(1) INIT(96);
    DCL DD  EXCP-DESCR-AVAIL BIN(4) DEF(EXCP-DESCR) POS(5);
    DCL DD  EXCP-MAT       CHAR(88) DEF(EXCP-DESCR) POS(9);
        DCL DD EXCP-MAT-CONTROL     CHAR(2) DEF(EXCP-MAT) POS( 1);
        DCL DD EXCP-MAT-INSTR-NBR    BIN(2) DEF(EXCP-MAT) POS( 3);
        DCL DD EXCP-MAT-CMPVAL-SIZE  BIN(2) DEF(EXCP-MAT) POS( 5);
        DCL DD EXCP-MAT-CMPVAL     CHAR(32) DEF(EXCP-MAT) POS( 7);
        DCL DD EXCP-MAT-NBR-OF-IDS   BIN(2) DEF(EXCP-MAT) POS(39);
        DCL SYSPTR .EXCP-MAT-HND-PGM        DEF(EXCP-MAT) POS(41);
        DCL DD EXCP-MAT-ID(16)      CHAR(2) DEF(EXCP-MAT) POS(57);
```

The last data item, EXCP-MAT-ID, is an array of exception IDs. It is here configured for 16 entries, but can have many more (albeit a rare occurrence). As is customary with templates, it starts with two binary numbers showing how many bytes are provided in the template and how many were actually materialized.

The control flags, EXCP-MAT-CONTROL, determine the further treatment of the exception as follows

Bits	Value	Meaning
0-2	000	Exception Handler Action: Ignore
	001	Exception Handler Action: Skip
	010	Exception Handler Action: Resignal
	100	Exception Handler Action: Defer
	101	Exception Handler Action: Pass control immediately
3	0	Return exception data
	1	Do not return exception data
4	0	Reserved, must be zero
5	0	User data not present
	1	User data present
6-7	00	Reserved, must be zeroes
8-9	00	External entry point handler
	01	Internal entry point handler
	10	Branch point handler
10-15	000000	Reserved, must be zeroes

The instruction number, EXCP-MAT-INSTR-NBR, is the MI-instruction number to be given control when this exception occurs. If the exception handler is external, EXCP-MAT-INSTR-NBR will be set to zero (as it is not to be used). The size and contents of the compare value, EXCP-MAT-CMPVAL-SIZE and EXCP-MAT-CMPVAL, have their obvious meanings.

Modifying an Exception Description

You can *modify* an exception description to alter the action to be taken. The simple form of the "Modify Exception Description" (**MODEXCPD**) instruction takes this format:

```
MODEXCPD    Exception-description, New-Control-Flags, X'01';
```

allowing you to specify a new set of control flags. You can also use a space pointer to an altered materialization template as operand 2 to change the exception description. You often modify an exception description after determining that the error causing the exception is indeed fatal and that no further repair of the situation makes sense or is even possible. In this case you should disable the exception monitor and retry the instruction (see below how to) causing the exception to occur again, but this time, due to your exception monitor having been disabled, it will cause the exception to be sent to the caller of your program. Here is how to disable the exception monitor:

```
MODEXCPD    Exception-description, X'2000', X'01';
```

Monitoring Exceptions: The MIDECEXC Program

To illustrate some of the features of exception handling, we'll write a program to repair various decimal data errors. Whether such errors should be ignored, repaired, reported, or cause the program to fail is completely application dependent. We are, of course, not advocating that errors be repaired in all cases. What our particular example program will do is:

- In case of invalid decimal data (MCH1202), set the data to zeroes
- In case of divide by zero (MCH1210), set the data to the highest value possible ("infinity")
- In case of result too large (MCH1211), set the data to the highest value possible ("infinity")

For illustration, the program starts by materializing an exception description. Debug the program with a break point set at "1" to see the result (EXCP-DESCR). The program then continues to force the various errors, accumulating the result of each operation in a message that is issued at the end of the program. You

can set a break point at "2" and look at (EXCP-INFO) to see the exception data returned for each exception as it occurs.

```
DCL SPCPTR .EXCP-DESCR INIT(EXCP-DESCR);
DCL DD      EXCP-DESCR CHAR(96) BDRY(16);
    DCL DD  EXCP-DESCR-PROV  BIN(4) DEF(EXCP-DESCR) POS(1) INIT(96);
    DCL DD  EXCP-DESCR-AVAIL BIN(4) DEF(EXCP-DESCR) POS(5);
    DCL DD  EXCP-MAT       CHAR(88) DEF(EXCP-DESCR) POS(9);
        DCL DD EXCP-MAT-CONTROL     CHAR(2) DEF(EXCP-MAT) POS( 1);
        DCL DD EXCP-MAT-INSTR-NBR    BIN(2) DEF(EXCP-MAT) POS( 3);
        DCL DD EXCP-MAT-CMPVAL-SIZE  BIN(2) DEF(EXCP-MAT) POS( 5);
        DCL DD EXCP-MAT-CMPVAL     CHAR(32) DEF(EXCP-MAT) POS( 7);
        DCL DD EXCP-MAT-NBR-OF-IDS   BIN(2) DEF(EXCP-MAT) POS(39);
        DCL SYSPTR .EXCP-MAT-HND-PGM        DEF(EXCP-MAT) POS(41);
        DCL DD EXCP-MAT-ID(16)      CHAR(2) DEF(EXCP-MAT) POS(57);

DCL EXCM DEC-ERROR EXCID(H'0C02', H'0C0A', H'0C0B')
                                            INT(ERROR) IMD CV("MCH");
DCL DD DATA CHAR(3);
    DCL DD NUMBER PKD(5,0) DEF(DATA) POS(1);

    MATEXCPD   .EXCP-DESCR, DEC-ERROR, X'00';
    BRK "1"; /* TO SHOW DESCRIPTION */

    CPYBREP     DATA, " "; /* MAKE INVALID PACKED NUMBER */
    CALLI       SHOW-DATA, *, .SHOW-DATA;

    ADDN(S)     NUMBER, 1; /* FORCE 'DECIMAL DATA ERROR' */
    CALLI       SHOW-DATA, *, .SHOW-DATA;

    DIV(S)      NUMBER, 0; /* FORCE 'ZERO DIVIDE  ERROR' */
    CALLI       SHOW-DATA, *, .SHOW-DATA;

    ADDN(S)     NUMBER, 1; /* FORCE 'NUMERIC SIZE ERROR' */
    CALLI       SHOW-DATA, *, .SHOW-DATA;

    SUBN(S)     NUMBER, 1; /* NO ERRORS, RESULT = 99998  */
    CALLI       SHOW-DATA, *, .SHOW-DATA;
    CALLI       SHOW-MESSAGE, *, .SHOW-MESSAGE;

    RTX         *;
```

Exception General and Specific Data

When an exception occurs, general exception information about the location and nature of the exception can be retrieved. In addition, many exceptions also have specific data associated with them giving further details about the error or condition that has occurred. The MI Functional Reference Manual chapter 27 provides a lot of details about this specific data. The exceptions we are monitoring for in our test program happen not to return any specific data, but we'll structure the code to be prepared for specific data in order to make it easier to adapt the code for other exceptions. The "Retrieve Exception Data" (**RETEXCPD**) instruction with this format

```
RETEXCPD    .Exception-info, Exception-handler-type;
```

retrieves the data related to the occurrence of the exception into the materialization area given by the space pointer .*Exception-info*. Here is the format of the information area:

```
DCL SPCPTR .EXCP-INFO INIT(EXCP-INFO);
DCL DD      EXCP-INFO CHAR(304) BDRY(16);
    DCL DD EXCP-INFO-PROV      BIN(4) DEF(EXCP-INFO) POS( 1) INIT(304);
    DCL DD EXCP-INFO-AVAIL     BIN(4) DEF(EXCP-INFO) POS( 5);
    DCL DD EXCP-INFO-ID       CHAR(2) DEF(EXCP-INFO) POS( 9);
    DCL DD EXCP-INFO-CMP-SIZE  BIN(2) DEF(EXCP-INFO) POS(11);
    DCL DD EXCP-INFO-CMPVAL   CHAR(32) DEF(EXCP-INFO) POS(13);
    DCL DD EXCP-INFO-REFKEY    BIN(4) DEF(EXCP-INFO) POS(45);
    DCL DD EXCP-DATA        CHAR(256) DEF(EXCP-INFO) POS(49);
```

The exception data, EXCP-DATA, contains both variable-size exception *specific* data and a 46-byte fixed-size data part related to the invocation and location of where the exception occurred. Unfortunately, the variable-size data comes first. To access the fixed-part, it is thus convenient to declare a *based* structure:

```
DCL DD INVOCATION-PART BIN(2);
DCL SPCPTR .EXCP-INVOC;
DCL DD      EXCP-INVOC CHAR(46) BAS(.EXCP-INVOC);
```

```
    DCL PTR .EXCP-SOURCE-INVOC          DEF(EXCP-INVOC) POS( 1);
    DCL PTR .EXCP-TARGET-INVOC          DEF(EXCP-INVOC) POS(17);
    DCL DD EXCP-SOURCE-INSTR     BIN(2) DEF(EXCP-INVOC) POS(33);
    DCL DD EXCP-TARGET-INSTR     BIN(2) DEF(EXCP-INVOC) POS(35);
    DCL DD EXCP-MACHINE-DATA   CHAR(10) DEF(EXCP-INVOC) POS(37);
```

We have made room for 256 bytes of exception data. Compute the offset to the fixed part from the number of bytes available and add it to the space pointer to the exception information to get the basing space pointer to the invocation information:

```
    SUBN        INVOCATION-PART, EXCP-INFO-AVAIL, 46;
    ADDSPP     .EXCP-INVOC, .EXCP-INFO, INVOCATION-PART;
```

The source invocation, `.EXCP-SOURCE-INVOC`, identifies the invocation that caused the exception. The target invocation, `.EXCP-TARGET-INVOC`, identifies the invocation that is the target of the exception, i.e. the last invocation that was given the chance to handle the exception. For machine exceptions, this is the invocation incurring the exception. For user-signaled exceptions you may specify a different target. You also retrieve the number of the MI-instruction that caused the exception in the source invocation and the number of the MI-instruction that is currently being executed in the target invocation.

.

The second operand, *`Exception-handler-type`*, specifies the exception handler type as follows:

X‘00’	retrieve for a branch point exception handler
X‘01’	retrieve for an internal entry point exception handler
X‘02’	retrieve for an external entry point exception handler

You normally begin exception handler processing with retrieving the exception data:

```
ENTRY ERROR INT;
    RETEXCPD   .EXCP-INFO, X'01'; /* RETRIEVE FOR INTERNAL ENTRY */
    BRK "2"; /* TO SHOW INFORMATION */
```

Now we implement the rules for handling the exceptions depending on the exception ID:

```
    CMPBLA(B)   EXCP-INFO-ID, X'0C02'/NEQ(=+2);
    CPYNV(B)    NUMBER, 00000/ZER(=+2);: /* DECIMAL ERROR */
    CPYNV       NUMBER, 99999;:          /* OTHER ERRORS  */

    SUBN        INVOCATION-PART, EXCP-INFO-AVAIL, 46;
    ADDSPP     .EXCP-INVOC, .EXCP-INFO, INVOCATION-PART;
```

Finally we want to return to the invocation with our “repaired” `NUMBER`:

```
    CPYBWP     .EXCP-RTN-INVOC, .EXCP-SOURCE-INVOC;
    RTNEXCP    .EXCP-RETURN;
```

Return from Exception

When an external exception handler invocation or an internal exception handler subinvocation gets control and then has done what it needs to do to process the exception, you need to terminate the handler with the “Return from Exception” (**RTNEXCP**) instruction:

```
RTNEXCP     .Exception-return-template;
```

where the operand specifies a space pointer to a template that in turn will specify the instruction to return to, within a specified invocation. The `RTNEXPD` instruction cannot be executed (and is not needed) in a branch point internal exception handler (you just branch to where you want to go). The template specifies the return address as the invocation pointer `.EXCP-RTN-INVOC`:

```
DCL SPCPTR .EXCP-RETURN INIT(EXCP-RETURN);
DCL DD      EXCP-RETURN CHAR(19) BDRY(16);
    DCL PTR .EXCP-RTN-INVOC     DEF(EXCP-RETURN) POS( 1);
    DCL DD  EXCP-MBZERO CHAR(1) DEF(EXCP-RETURN) POS(17) INIT(X'00');
    DCL DD  EXCP-ACTION CHAR(2) DEF(EXCP-RETURN) POS(18) INIT(X'0100');
```

The reserved variable `EXCP-MBZERO` must be binary zero. The action code, `EXCP-ACTION`, determines what happens next:

Code	Action

X‘0200’	Resume execution with the instruction that follows the RTNEXCP instruction (terminating the internal exception handler subinvocation).
X‘0100’	Resume execution with the instruction following the instruction causing the exception
X‘0000’	Re-execute the instruction that caused the exception

In our example program we want to resume execution of the instruction that follows the instruction causing the exception, so we set the action code to x‘0100’.

The following little routine just inserts the current value of our NUMBER into a message to show at the end of the program. A small thing to note is the **AUTO** attribute of the position variable, N. The default storage attribute is STATIC, meaning that the variable is only allocated storage once and that it keeps its value from invocation to invocation. With the **AUTO** attribute, the variable is re-allocated and re-initialized every time the program is called. In our example, that ensures that we start from the beginning of the message area every time:

```
DCL DD N AUTO BIN(2) INIT(1); /* reset every time PGM is run */
DCL INSPTR .SHOW-DATA;
ENTRY       SHOW-DATA INT;
    CMPNV(B)    N, 1/HI(=+2);
    CPYBREP     MSG-TEXT, " ";:  /* clear to blanks if N = 1 */
    CVTHC       MSG-TEXT(N:6), DATA;      /* contains NUMBER */
    ADDN(S)     N, 10;
    B          .SHOW-DATA;

%INCLUDE SHOWMSG
```

Running the program we get:

```
Type reply (if required), press Enter.
  From  . . . :   LSVALGAARD     12/06/00   13:28:24
  404040    00000F    99999F    99999F    99998F
```

NUMBER starts out as the invalid packed decimal value x‘404040’ (blanks), then is repaired to zeroes, then set to the maximum value after the zero divide and after the attempt to add 1 more to it, and finally ends up being the valid value 9998.

Signaling Exceptions: The MISIGEXC Program

In addition to monitoring for exceptions, you can also *signal* exceptions to occur. You can either *re-signal* the exception from inside an exception handler (in which case the invocation is known) or you can signal a *new* exception (we could use the word *condition* for such new exceptions). When you want to signal a condition, the first problem you have is to get an *invocation pointer* as the target for the exception. You use the “Materialize Invocation Attributes” (**MATINVAT**) instruction to get the invocation pointer.

Materialize Invocation Attributes

The MATINVAT instruction causes either one specific attribute or a list of attributes of the designated invocation to be materialized. We’ll only consider the first option here (as it gets rather complicated otherwise). The syntax is:

```
MATINVAT     .Receiver, Invocation, .Selection-template;
```

Operand 1, *.Receiver*, specifies a space pointer to the area to receive the attribute. Operand 2, *Invocation*, identifies the source invocation whose attribute is to be retrieved. If this operand is the *null* operand, “*”, the invocation issuing the instruction is identified. Operand 3, *.Selection-template*, is a space pointer to a template that selects the attribute to be materialized. Most of the fields in the template have to do with how to store a list of attributes and can be set to zeroes if only one attribute is to be materialized:

```
DCL SPCPTR .MAT-RECEIVER INIT(MAT-RECEIVER);
DCL DD      MAT-RECEIVER CHAR(16) BDRY(16);
    DCL PTR .MAT-INVOC   DEF(MAT-RECEIVER) POS(1);

DCL SPCPTR .MAT-SELECT INIT(MAT-SELECT);
DCL DD      MAT-SELECT CHAR(32);
    DCL DD  MAT-NBR-ATTRS    BIN(4) DEF(MAT-SELECT) POS( 1) INIT( 1);
    DCL DD  MAT-ATTR-FLAGS   BIN(4) DEF(MAT-SELECT) POS( 5) INIT( 0);
    DCL DD  MAT-ATTR-OFFSET  BIN(4) DEF(MAT-SELECT) POS( 9) INIT( 0);
```

```
    DCL DD  MAT-ATTR-STORED  BIN(4) DEF(MAT-SELECT) POS(13) INIT( 0);

    DCL DD  MAT-ATTR-ID      BIN(4) DEF(MAT-SELECT) POS(17) INIT( 1);
    DCL DD  MAT-RECV-FLAGS   BIN(4) DEF(MAT-SELECT) POS(21) INIT( 0);
    DCL DD  MAT-RECV-OFFSET  BIN(4) DEF(MAT-SELECT) POS(25) INIT( 0);
    DCL DD  MAT-RECV-LENGTH  BIN(4) DEF(MAT-SELECT) POS(29) INIT(16);
```

The important fields are the number of attributes to materialize (NBR-ATTRs = 1), the attribute identifier (ATTR-ID = 1, for the invocation pointer), and the length of the receiver area (RECV-LENGTH = 16 bytes, because we're materializing a pointer, and that's how large and plump pointers are).

The program now starts by materializing the invocation pointer to its own invocation:

```
GET-OWN-INVOCATION:
    MATINVAT   .MAT-RECEIVER, *, .MAT-SELECT;
```

Signaling an Exception

The "Signal Exception" (**SIGEXCP**) instruction signals ("sends") a new exception or re-signals an existing exception to the target invocation. The syntax is:

```
    SIGEXCP      .Signal-info, .Exception-info;
```

The first operand is a space pointer to a template holding the target invocation pointer, .EXCP-TO-INVOC. Bits 0 of the EXCP-**OPTION** determines if we have a new exception (bit 0 = 0) or if we are re-signaling an existing exception (bit 0 = 1). Setting bit 2 to a 1 gives you control over which exception description to search first. We are happy with the default, which is simply the first:

```
DCL SPCPTR .EXCP-SIGNAL INIT(EXCP-SIGNAL);
DCL DD      EXCP-SIGNAL CHAR(20) BDRY(16);
    DCL PTR .EXCP-TO-INVOC        DEF(EXCP-SIGNAL) POS( 1);
    DCL DD  EXCP-OPTION   CHAR(1) DEF(EXCP-SIGNAL) POS(17) INIT(X'00');
    DCL DD  *             CHAR(1) DEF(EXCP-SIGNAL) POS(18) INIT(X'00');
    DCL DD  EXCP-1ST-DESCR BIN(2) DEF(EXCP-SIGNAL) POS(19) INIT(1);
```

The second operand (which is ignored for a re-signal operation, as the data is already known) has the same basic format as the template used by the RTNEXCPD instruction:

```
DCL SPCPTR .EXCP-INFO INIT(EXCP-INFO);
DCL DD      EXCP-INFO CHAR(64) BDRY(16);
    DCL DD  EXCP-BYTES-PROV   BIN(4) DEF(EXCP-INFO) POS( 1) INIT(64);
    DCL DD  EXCP-BYTES-AVAIL  BIN(4) DEF(EXCP-INFO) POS( 5);
    DCL DD  EXCP-ID           BIN(2) DEF(EXCP-INFO) POS( 9);
    DCL DD  EXCP-CMP-SIZE     BIN(2) DEF(EXCP-INFO) POS(11);
    DCL DD  EXCP-CMP-VAL    CHAR(32) DEF(EXCP-INFO) POS(13);
    DCL DD  *                 BIN(4) DEF(EXCP-INFO) POS(45);
    DCL DD  EXCP-DATA       CHAR(16) DEF(EXCP-INFO) POS(49);
```

Since we are not interested in the invocation data, we set the length provided for the information area, EXCP-BYTES-PROV, to only include the 16 bytes of exception specific data that our sample code will use.

The plan is now to signal two exceptions. The first one to be caught by our own exception monitor (or condition handler if you prefer), and the second one without a monitor so that the exception will be intercepted by the default exception handler that every process has. First the monitor for our very own condition "LSV7777" (I just made this up):

```
DCL EXCM MY-CONDITION EXCID(H'7777') BP(GOT-CONDITION) IMD CV("LSV");
```

We now signal the condition:

```
SIGNAL-CONDITION:
    CPYBWP     .EXCP-TO-INVOC, .MAT-INVOC;
    CPYBLA      EXCP-ID,        X'7777';
    CPYNV       EXCP-CMP-SIZE,  3;
    CPYBLA      EXCP-CMP-VAL,   "LSV";
    CPYBLAP     EXCP-DATA,   "Exception Data", " ";
    SIGEXCP    .EXCP-SIGNAL, .EXCP-DATA;
```

The condition handler is shown a little further on. After handling the condition, control returns to here, where we issue the data value error (MCH1223) exception:

```
SIGNAL-ERROR:
```

```
    CPYBLA      EXCP-ID,        X'0C17';    /* 1223 = Data value error */
    CPYNV       EXCP-CMP-SIZE,  4;          /* machine-generated compare length */
    CPYBLA      EXCP-CMP-VAL,   X'00000000'; /* MCH */
    SIGEXCP    .EXCP-SIGNAL, .EXCP-DATA;
```

Finally we return from the program:

```
    RTX         *;
```

The condition handler is a branch point handler, so we must retrieve exception data as appropriate for a branch point:

```
GOT-CONDITION:
    RETEXCPD   .EXCP-INFO, X'00'; /* RETRIEVE FOR BRANCH POINT */
    CPYBLAP     MSG-TEXT, EXCP-DATA(1:16), " ";
    CALLI       SHOW-MESSAGE, *, .SHOW-MESSAGE;
    B           SIGNAL-ERROR;

%INCLUDE SHOWMSG
```

We simply show the data returned as a message:

```
Type reply (if required), press Enter.
  From  . . . :   LSVALGAARD     12/06/00   21:52:15
  Exception Data
```

When the program finishes, the job log contains:

```
Data value error.
Processing command failure, refer to job log for details
```

Preventing Messages in the Joblog

When you run the **MIDECEXC** program the program catches all three exceptions and does its repair work as desired. There is, however, an undesirable (in most cases) side effect: For every exception caught, an entry is made in the job log for your job, *e.g.*:

```
3 > call midecexc
    Decimal data error.
    Attempt made to divide by zero for fixed point operation.
    Receiver value too small to hold result.
```

There could be thousands of such exceptions and although some indications of the occurrence of these exceptions would be desirable, it is better that the program itself issues a short summary at the end, rather than the system clogging up the job log with thousands of messages. So, the question is: Can we prevent an exception from generating a job log entry?

Since the *real* compare value generated by the machine is x'**00000000**' and not "**MCH**", we must monitor with *that* compare value, i.e.:

```
DCL EXCM DEC-ERROR EXCID(H'0C02', H'0C0A', H'0C0B')
                                  INT(ERROR) IMD CV(X'00000000');
```

instead of (as we did):

```
DCL EXCM DEC-ERROR EXCID(H'0C02', H'0C0A', H'0C0B')
                                  INT(ERROR) IMD CV("MCH");
```

If we do that, it turns out that no job log entry is generated. If we use CV("**MCH**"), the exception is really caught first by a lower layer monitoring for x'**00000000**'. That lower layer issues its joblog message and resignals a new exception (with compare value "**MCH**") for us to catch.

Editing of Numeric Variables

The Importance of Editing

It has been said that 90% of all computer power is spent on sorting (either directly or indirectly through maintenance of keyed data), and that 90% of the remaining power is spent on editing of numeric data for presentation. One rationale for the use of "zoned" and "packed" data formats was mainly to cut down on the computational cost of editing (dividing by 10 several times for each number is expensive - some early computers did not even have a `DIVIDE` instruction). Another rationale was simply that many early computers mainly worked in decimal anyway (even addresses were decimal).

Edit Codes and Edit Words

Traditionally, editing numeric values on the AS/400 was specified through the use of *Edit Codes* and *Edit Words*.

> "An edit code is a standard description of how a number should be formatted. There are many standard edit codes defined by the system. Users can define several edit codes the way they want with the use of the Create Edit Description (`CRTEDTD`) command. An edit word is a user-defined description of how a number should be formatted. An edit word is usually used when one of the standard edit codes or user-defined edit codes is not sufficient for a particular situation".

The above quote from one of the RPG manuals already hints that editing using edit codes and edit words is less than satisfactory. In fact, it often seems that these techniques are part of the problem rather than part of the solution. Luckily, the machine has a very powerful **`EDIT`** MI-instruction that makes the use of the non-intuitive edit codes and edit words superfluous.

COBOL Pictures

In contrast to RPG, the COBOL language has always had a very simple, visual, and intuitive way of working with editing specifications: the COBOL *Picture*. With only a slight simplification one can state that the **`EDIT`** instruction's purpose is to support the COBOL *edited picture* specification. A numeric variable in COBOL can be either in internal numeric or in *edited numeric* format. Let's look at a few examples. The `IN-DATA` picture is internal numeric, while all the `OUT-DATA` pictures are edited numeric:

```
02  IN-DATA                 PIC S9(7)V99.
02  OUT-DATA1               PIC -9,999,999.99.
02  OUT-DATA2               PIC --,---,--9.99.
02  OUT-DATA3               PIC 9,999,999.99-.
02  OUT-DATA4               PIC Z,ZZZ,ZZ9.99-.

MOVE -1234567.89 TO IN-DATA
MOVE IN-DATA TO OUT-DATA1, OUT-DATA2,  OUT-DATA3, OUT-DATA4
DISPLAY         OUT-DATA1, OUT-DATA2,  OUT-DATA3, OUT-DATA4        results in:

-1,234,567.89 -1,234,567.89  1,234,567.89- 1,234,567.89-

MOVE -1.23       TO IN-DATA
MOVE IN-DATA TO OUT-DATA1, OUT-DATA2,  OUT-DATA3, OUT-DATA4
DISPLAY         OUT-DATA1, OUT-DATA2,  OUT-DATA3, OUT-DATA4        results in:

-0,000,001.23         -1.23  0,000,001.23-         1.23-
```

Very little explanation is really needed. The pictures speak for themselves. The only subtlety is that two of the edited pictures (with all the 9s) specify a **fixed** format, while the other two specify a **floating** format. In the floating format (nothing to do with *floating-point* format), the leading blanks and the sign "float" to the right until they meet the first significant digit. Note also the thousand separator commas that are inserted at either fixed places (for fixed formats) or as needed among significant digits (for floating formats). In the following section we'll see how to instruct the **`EDIT`** instruction to do its magic.

The EDIT MI-Instruction

The general format of the **EDIT** instruction is:

```
EDIT        Character-Receiver, Numeric-Source, Edit-Mask
```

The *Edit-Mask* controls the editing of the *Numeric-Source* into the *Character-Receiver*. The first step of the editing process is to automatically translate the source number to a packed decimal format large enough to hold the numeric value.

The Edit Mask

The COBOL move-statement: `MOVE IN-DATA TO OUT-DATA1` generates the following code:

```
DCL CON PICTURE1 CHAR(23) INIT(X'AF40AE60AEAAB36BAEAAAAAAB36BAEAAAAAAB34BAEAAAA';
EDIT OUT-DATA1, IN-DATA, PICTURE1;
```

Although the edit mask (`PICTURE1`) looks somewhat forbidding it is really just a simple encoding of the COBOL picture:

Edit Mask Part	Explanation for **-9,999,999.99**
AF **40** AE **60** AE	Positive: **space**; Negative: **minus sign**
AA	9 digit position (significant or not)
B3 **6B** AE	**,** separator comma
AA AA AA	999 digit positions
B3 **6B** AE	**,** separator comma
AA AA AA	999 digit positions
B3 **4B** AE	**.** decimal point
AA AA	99 decimals

The edit mask contains both *control* characters and *data* characters. The character x'AE' is used as an "End-of-String" character (EOS), but only for terminating the various streams of control characters that are interspersed between the data characters. Any character less than x'40' can also be used as the EOS, but it is simpler to stick to the standard x'AE'. The x'AF' control character specifies which character to use for the sign (first for positive or zero values and after the EOS for negative values) for fixed format pictures. The `EDIT` instruction works from left to right through the source and the edit mask. The x'AA' control character specifies that the corresponding source digit is to be copied to the receiver, even if we are still among the leading zeroes *(i.e.* have no numeric "significance" yet). The x'B3' control character specifies that the character that follows is to be inserted at the current position in the receiver.

Here is the next example:

```
DCL CON PICTURE2 CHAR(25) INIT(X'B14040AE60AEB2B2B06BAEB2B2B2B06BAEB2B2AAB34BAEAAAA');
EDIT OUT-DATA2, IN-DATA, PICTURE2;
```

Edit Mask Part	Explanation for **--,---,--9.99**
B1 **40 40** AE **60** AE	Fill: **space**; Positive: **space**; Negative: **minus sign**
B2 B2	-- either a source digit or the *fill* character to be used (two of them)
B0 **6B** AE	**,** separator comma
B2 B2 B2	--- source digits (if significant) otherwise fill character
B0 **6B** AE	**,**
B2 B2 AA	--9 last of the three always source digit
B3 **4B** AE	**.** decimal point
AA AA	99 source decimals

The x'B1' control character specifies first which character to use as a fill character before the sign for floating formats; then follow the characters to use for the sign (first for positive or zero values and after the EOS for negative values). Note that the sign characters could actually be a string, *e.g.* `CR` for credit or `DB` for debit. The x'B0' control character specifies to insert either the character given (if significance has been

reached) or the fill character. The x'B2' control character specifies to use either a source digit (if significance has been reached) or the fill character.

Next example:

```
DCL CON PICTURE3 CHAR(23) INIT(X'AAB36BAEAAAAAAB36BAEAAAAAAB34BAEAAAAAF40AE60AE');
EDIT OUT-DATA3, IN-DATA, PICTURE3;
```

Edit Mask Part	Explanation for **9,999,999.99-**
AA	9 source digit
B3 **6B** AE	, separator comma
AA AA AA	999 source digits
B3 **6B** AE	, separator comma
AA AA AA	999 source digits
B3 **4B** AE	. decimal point
AA AA	99 source decimals
AF **40** AE **60** AE	Positive: **space**; Negative: **minus sign**

```
DCL DD PICTURE4 CHAR(27) INIT(X'B140AEAEB2B06BAEB2B2B2B06BAEB2B2AAB34BAEAAAAAF40AE60AE');
EDIT OUT-DATA4, IN-DATA, PICTURE4;
```

Edit Mask Part	Explanation for **Z,ZZZ,ZZ9.99-**
B1 **40** AE AE	Fill: **space**; note that there are no sign characters, only EOSs
B2	Z either a source digit or the *fill* character to be used
B0 **6B** AE	, separator comma
B2 B2 B2	ZZZ source digits (if significant) otherwise fill character
B0 **6B** AE	,
B2 B2 AA	ZZ9 last of the three always source digit
B3 **4B** AE	. decimal point
AA AA	99 source decimals
AF **40** AE **60** AE	Positive: **space**; Negative: **minus sign**

The starting x'B1' control sequence does not contain any sign character strings (as the picture specifies a trailing sign), but we still need two EOSs. Let's finish with two more examples:

```
02  OUT-DATA5          PIC 9B999B999.99+.       Result = 1 234 567.89+   0 000 001.23+
```

```
DCL CON PICTURE5 CHAR(23) INIT(X'AAB340AEAAAAAAB340AEAAAAAAB34BAEAAAAAF4EAE60AE');
EDIT OUT-DATA5, IN-DATA, PICTURE5;
```

Edit Mask Part	Explanation for **9B999B999.99-**
AA	9 source digit
B3 **40** AE	ь separator blank
AA AA AA	999 source digits
B3 **40** AE	ь separator blank
AA AA AA	999 source digits
B3 **4B** AE	. decimal point
AA AA	99 source decimals
AF **4E** AE **60** AE	Positive: **plus sign**; Negative: **minus sign**

```
02  OUT-DATA6          PIC ********9.99-.       Result = **1234567.89-    ********1.23-
```

```
DCL CON PICTURE6 CHAR(23) INIT(X'B15CAEAEB2B2B2B2B2B2B2B2B2AAB34BAEAAAAAF40AE60AE');
EDIT OUT-DATA6, IN-DATA, PICTURE6;
```

Edit Mask Part	Explanation for **********9.99-**
B1 **5C** AE AE	Fill: **asterisk**; note that there are no sign characters, only EOSs
B2 B2 B2 B2 … AA	***…9 fill with *, end with one source digit
AA AA AA	999 source digits

B3 **4B** AE	**.** decimal point
AA AA	99 source decimals
AF **4E** AE **60** AE	Positive: **plus sign**; Negative: **minus sign**

Parameter Length Conformance

An *edit digit count* (x'0C04') exception is signaled if:

- The end of the source is reached and there are more control characters that correspond to digits in the edit mask.
- The end of the edit mask is reached and there are more digit positions in the source.

A *length conformance* (x'0C08') exception is signaled if:

- The end of the edit mask is reached and there are more control character positions in the receiver.
- The end of the receiver is reached and more positions remain in the edit mask.
- The number of B2s following a B1 cannot accommodate the longer of the two floating strings.

The moral here is simple: be careful with the lengths. The receiver cannot be too large either. Everything must just fit.

Left-Justifying the Number

Numbers are usually right-justified in a field and the `EDIT` instruction is designed to produce right-justified results, i.e. values that "butt" up against the right-hand side of the field. Occasionally, one has the need to produce results that are *left-justified*. The following little MI-program, **`MIEDTNBR`**, shows how first to use the `EDIT` instruction to edit a numeric value and then how to use the `VERIFY` instruction to help left-justify the result:

```
DCL DD RECEIVER  CHAR(13);
DCL DD NUMBER    PKD(9,2) INIT(P'-1.23');
DCL DD PICTURE   CHAR(25) /* COBOL PICTURE --,---,--9.99 */
          INIT(X'B14040AE60AEB2B2B06BAEB2B2B2B06BAEB2B2AAB34BAEAAAA');
               /*           - -   ,   - - -   ,   - - 9   .   9 9 */

DCL SPCPTR .PARM1 PARM;
DCL DD PARM-NBR PKD(15,5) BAS(.PARM1);

DCL OL PARMS(.PARM1) PARM EXT MIN(1);

ENTRY * (PARMS) EXT;
    CPYNV        NUMBER, PARM-NBR;
    EDIT         RECEIVER, NUMBER, PICTURE;

    CPYBLAP      MSG-TEXT, "[", " ";
    CPYBLAP      MSG-TEXT(2:14), RECEIVER, "]";

DCL DD WHERE  BIN(2);
DCL DD LENGTH BIN(2);

    VERIFY(B)    WHERE,  RECEIVER, " "/ZER(=+3);
    SUBN         LENGTH, 14, WHERE; /* 14 = 13 + 1 */
    CPYBOLAP     RECEIVER, RECEIVER(WHERE:LENGTH), " ";:

    CPYBLA       MSG-TEXT(18:1), "[";
    CPYBLAP      MSG-TEXT(19:14), RECEIVER, "]";
    CALLI        SHOW-MESSAGE, *, .SHOW-MESSAGE;

    RTX          *;

%INCLUDE SHOWMSG
```

Here is a sample run:

```
==> CALL MIEDTNBR PARM(-1.23)
From . . . LSVALGAARD    02/12/01  21:54:43
[        -1.23]  [-1.23        ]
```

MI-Instructions Quick Reference

MI-Instruction Quick Reference

The following Quick Reference Guide covers all MI-instructions for which information for the AS/400 is available. It is not meant as a substitute for the MI Functional Reference, but does provide a useful (and mercifully short) overview of the various instructions, their possible operands and branch conditions. I use it all the time in lieu of the MIFR. I hope the format is reasonably self-explanatory.

Computational and Branching Instructions

OPCODE	Name	Operands and Conditions
ADDLC *IBS* *4 ext*	*Add Logical Character*	*Sum CHARVF* *Addend 1 CHARF* *Addend 2 CHARF* *All same length <= 256* *ZNTC NTZNTC ZC NTZC*
ADDN *IBSR* *4 ext*	*Add Numeric*	*Sum NUMV* *Addend 1 NUM* *Addend 2 NUM* *POS NEG ZER NAN*
AND *IBS* *2 ext*	*And Bytes*	*Result CHARV* *Source 1 CHAR* *Source 2 CHAR* *Longer of 2 Sources, then place in result, pad X'00' or truncate* *ZER NZER*
B	*Branch*	*Target BP, INSPTR, IDL element*
CIPHER	*Cipher*	*Result SPCPTR* *Control CHAR(32)V* *Source SPCPTR*
CIPHERKY	*Cipher Key*	*Result CHAR(8)V* *Control CHAR(64)V* *Source CHAR(8)*
CLRBTS	*Clear Bit in String*	*Result CHARV, NUMV* *Bit number BIN*
CMPBLA *IB req* *3 ext*	*Compare Bytes Left-Adjusted*	*Compare 1 CHAR, NUM* *Compare 2 CHAR, NUM* *Shorter of 2 compare operands* *Stops on 1st unequal byte* *HI LO EQ*
CMPBLAP *IB req* *3 ext*	*Compare Bytes Left-Adjusted with Pad*	*Compare 1 CHAR, NUM* *Compare 2 CHAR, NUM* *Pad CHAR(1), NUM(1)* *Shorter of 2 compare operands* *Stops on 1st unequal byte* *HI LO EQ*
CMPBRA *IB req* *3 ext*	*Compare Bytes Right-Adjusted*	*Compare 1 CHAR, NUM* *Compare 2 CHAR, NUM* *Shorter of 2 compare operands* *Stops on 1st unequal byte* *HI LO EQ*
CMPBRAP *IB req* *3 ext*	*Compare Bytes Right-Adjusted with Pad*	*Compare 1 CHAR, NUM* *Compare 2 CHAR, NUM* *Pad CHAR(1), NUM(1)* *Shorter of 2 compare operands* *Stops on 1st unequal byte* *HI LO EQ*
CMPNV *IB req* *4 ext*	*Compare Numeric Values*	*Compare 1 NUM* *Compare 2 NUM* *HI LO EQ UNOR*

OPCODE	Name	Operands and Conditions
CMPSW IB req 1 ext	Compare and Swap	Compare 1 CHAR(1,2,4,8)V Compare 2 CHAR(1,2,4,8)V Swap Operand CHAR(1,2,4,8) EQ
CPRDATA	Compress Data	Template SPCPTR to CHAR(64)
CAI	Compute Array Index	Index BIN(2)V Subscript A BIN(2) Subscript B BIN(2) Dimension BIN(2)C, IMM(2)
CMF1 IB 4 ext	Compute Mathematical Function using 1 Input	Result FLTV Control CHAR(2) Source FLT POS NEG ZER NAN
CMF2 IB 4 ext	Compute Mathematical Function using 2 Inputs	Result FLTV Control CHAR(2) Source 1 FLT Source 2 FLT POS NEG ZER NAN
CAT	Concatenate	Result CHARV Source 1 CHAR Source 2 CHAR Length of Result, pad X'40' or truncate
CVTBC IB 3 ext	Convert BSC to Characters	Result CHARV Control CHAR(3)VF Source CHAR CR SE TR
CVTCB IB 2 ext	Convert Characters to BSC	Result CHARV Control CHAR(3)VF Source CHAR SE RO
CVTCH	Convert Characters to HEX	Result CHARV Source CHAR [0-9A-F] Length of Result, pad X'00'
CVTCM IB 2 ext	Convert Characters to Multi-leaving Remote Job Entry Format	Result CHARV Control CHAR(13)VF Source CHAR SE RO
CVTCN	Convert Characters to Numeric	Result NUMV, DTAPTR-NUM Source CHAR, DTAPTR-CHAR Attributes CHAR(7), DTAPTR-CHAR(?)
CVTCS IB 2 ext	Convert Characters to SNA Format	Result CHARV Control CHAR(15)V Source CHAR SE RO
CVTDFFP	Convert Decimal Form to Floating-Point	Result FLOATV Dec.exponent PKD, ZND Dec.mantissa PKD, ZND
CVTEFN	Convert External Form to Numeric Value	Result NUMV, DTAPTR-NUM Source CHAR, DTAPTR-CHAR Mask CHAR(3),DTAPTR-CHAR(3),*
CVTFPDF R	Convert Floating-Point to Decimal Form	Dec.exponent PKDV, ZNDV Dec.mantissa PKDV, ZNDV Source FLOAT
CVTHC	Convert HEX to Characters	Result CHARV Source CHAR Length of Result, pad X'F0'
CVTMC IB 2 ext	Convert Multi-leaving Remote Job Entry Format to Characters	Result CHARV Control CHAR(6)VF Source CHAR SE RO
CVTNC	Convert Numeric to Characters	Result CHARV, DTAPTR-CHARV Source NUM, DTAPTR-NUM Attributes CHAR(7), DTAPTR-CHAR(7)
CVTSC IB 3 ext	Convert SNA Format to Characters	Result CHARV Control CHAR(14)V Source CHAR SE RO EC

OPCODE	Name	Operands and Conditions
CPYBTA	*Copy Bits Arithmetic*	*Receiver CHARV, NUMV* *Source CHAR, NUM* *Offset IMM* *Length IMM between 1 and 32* *Length of Receiver not > than 4 bytes*
CPYBTL	*Copy Bits Logical*	*Receiver CHARV, NUMV* *Source CHAR, NUM* *Offset IMM* *Length IMM between 1 and 32* *Length of Receiver not > than 4 bytes*
CPYBTLLS	*Copy Bits with Left Logical Shift*	*Receiver CHARV, NUMV* *Source CHAR, NUM, IMM(1)* *Shift CHAR(2)F, IMM(2)* *Length of Receiver, pad B'0'*
CPYBTRAS	*Copy Bits with Right Arithmetic*	*Receiver CHARV, NUMV* *Source CHAR, NUM, IMM(1)* *Shift CHAR(2)F, IMM(2)* *Length of Receiver, pad sign*
CPYBTRLS	*Copy Bits with Right Logical Shift*	*Receiver CHARV, NUMV* *Source CHAR, NUM, IMM(1)* *Shift CHAR(2)F, IMM(2)* *Length of Receiver, pad B'0'*
CPYBLA	*Copy Bytes Left-Adjusted*	*Receiver CHARV, NUMV,* *DTAPTR-CHARV,-NUMV* *Source CHAR, NUM,* *DTAPTR-CHAR,-NUM* *Shorter of two operands*
CPYBLAP	*Copy Bytes Left-Adjusted with Pad*	*Receiver CHARV, NUMV,* *DTAPTR-CHARV,-NUMV* *Source CHAR, NUM,* *DTAPTR-CHAR,-NUM* *Pad CHAR(1), NUM(1)* *Length of Receiver, pad*
CPYBOLA	*Copy Bytes with Overlap Left-Adjusted*	*Receiver CHARV, NUMV,* *Source CHAR, NUM,* *Shorter of two operands*
CPYBOLAP	*Copy Bytes with Overlap Left-Adjusted with Pad*	*Receiver CHARV, NUMV,* *Source CHAR, NUM,* *Pad CHAR(1), NUM(1)* *Length of Receiver, pad*
CPYBREP	*Copy Bytes Repeatedly*	*Receiver CHARVF, NUMV,* *Source CHARF, NUM,* *Length of Receiver*
CPYBRA	*Copy Bytes Right-Adjusted*	*Receiver CHARV, NUMV,* *DTAPTR-CHARV,-NUMV* *Source CHAR, NUM,* *DTAPTR-CHAR,-NUM* *Shorter of two operands*
CPYBRAP	*Copy Bytes Right-Adjusted with Pad*	*Receiver CHARV, NUMV,* *DTAPTR-CHARV,-NUMV* *Source CHAR, NUM,* *DTAPTR-CHAR,-NUM* *Pad CHAR(1), NUM(1)* *Length of Receiver, pad*
CPYBBTA	*Copy Bytes to Bits Arithmetic*	*Receiver CHARV, NUMV,* *Offset IMM* *Length IMM from 1 to 32* *Source CHAR, NUM,* *Length of Source <= 4 bytes*
CPYBBTL	*Copy Bytes to Bits Logical*	*Receiver CHARV, NUMV,* *Offset IMM* *Length IMM from 1 to 32* *Source CHAR, NUM,* *Length of Source <= 4 bytes*
CPYECLAP	*Copy Extended Characters Left-Adjusted with Pad*	*Receiver DTAPTR-CHARV* *Source DTAPTR-CHAR* *Pad CHAR(3), **

OPCODE	Name	Operands and Conditions
CPYHEXNN	Copy Hex Digit Numeric to Numeric	Receiver CHARV, NUMV, Source CHARF, NUM,
CPYHEXNZ	Copy Hex Digit Numeric to Zone	Receiver CHARVF, NUMV Source CHARF, NUM
CPYHEXZN	Copy Hex Digit Zone to Numeric	Receiver CHARVF, NUMV Source CHARF, NUM
CPYHEXZZ	Copy Hex Digit Zone to Zone	Receiver CHARVF, NUMV Source CHARF, NUM
CPYNV IBR 4 ext	Copy Numeric Value	Receiver NUMV, DTAPTR-NUMV Source NUM, DTAPTR-NUM POS NEG ZER NAN
DCPDATA	Decompress Data	Template SPCPTR to CHAR(64)
DIV IBSR 4 ext	Divide	Quotient NUMV Dividend NUM Divisor NUM POS NEG ZER NAN
DIVREM IBSR 3 ext	Divide with Remainder	Quotient NUMV Dividend NUM Divisor NUM Remainder NUMV POS NEG ZER
EDIT	Edit	Result CHARV, DTAPTR-CHARV Source NUM, DTAPTR-NUM Mask NUM, DTAPTR-NUM
EXCHBY	Exchange Bytes	Source 1 NUMV, CHARVF Source 2 NUMV, CHARVF Same length
XOR IBS 2 ext	Exclusive Or Bytes	Result CHARV Source 1 CHAR Source 2 CHAR Longer of 2 Sources, then place in result, pad X'00' or truncate ZER NZER
ECSCAN IB req 3 ext	Extended Character Scan	Result BINV, BINA Base CHAR Compare CHAR Mode CHAR(1) POS ZER EC
EXTREXP IB 4 ext	Extract Exponent	Result BINV Source FLOAT NOR DEN INF NAN
EXTRMAG IB 3 ext	Extract Magnitude	Result NUMV Source NUM POS ZER NAN
MULT IBSR 4 ext	Multiply	Product NUMV Multiplicand NUM Multiplier NUM POS NEG ZER NAN
NEG IBS 4 ext	Negate Number	Result NUMV Source NUM POS NEG ZER NAN
NOT IBS 2 ext	Not Bytes	Result CHARV Source CHAR Length of Result, pad X'00' ZER NZER
OR IBS 2 ext	Or Bytes	Result CHARV Source 1 CHAR Source 2 CHAR Longer of 2 Sources, then place in result, pad X'00' or truncate ZER NZER
REM IBS 3 ext	Remainder	Remainder NUMV Dividend NUM Divisor NUM POS NEG ZER

OPCODE	Name	Operands and Conditions
SCALE IBS 4 ext	Scale	Result NUMV Source NUM Scale Factor BIN(2) POS NEG ZER NAN
SCAN IB 3 ext	Scan Characters	Result BINV, BINA Base CHAR (<= 32k) Compare CHAR POS ZER NCMP
SCANWC IB 4 ext	Scan with Control	Base Locator SPCPTR Control CHAR(8)V Options CHAR(4) Escape Addr BP, INSPTR, IDL elem, * HI LO EQ NF
SEARCH IB 2 ext	Search	Result BINV, BINA Array CHARA, NUMA Find CHAR, NUM Location BIN POS ZER
SETBTS	Set Bit in String	Result CHARV, NUMV Offset BIN
SETIP	Set Instruction Pointer	Result INSPTR Target BP
SSCA IB	Store and Set Computational Attributes	Old Attrs CHAR(5)V New Attrs CHAR(5), * Control CHAR(5), *
SUBLC IBS 3 ext	Subtract Logical Character	Difference CHARVF Minuend CHARF Subtrahend CHARF All same length <= 256 ZC NTZC NTZNTC
SUBN IBSR 4 ext	Subtract Numeric	Difference NUMV Minuend NUM Subtrahend NUM POS NEG ZER NAN
TSTRPLC IBS	Test and Replace Characters	Result CHARV Replacement CHAR
TSTBTS IB 2 ext	Test Bit in String	Source CHARV, NUMV Offset BIN ZER ONE
TSTBUM IB req 3 ext	Test Bits Under Mask	Source CHAR, NUM Mask CHAR, NUM Only 1st byte ZER ONES MXD
XLATE	Translate	Result CHARV Source CHAR Position CHAR, * Replacement CHAR
XLATEMB	Translate Multiple Bytes	Template SPCPTR to CHAR(128)
XLATEWT	Translate with Table	Result CHARV Source CHAR Table CHAR(256) Shorter of Result and Source
XLATWTDS	Translate with Table and DBCS Skip	Target CHARV Length BIN(4) Table CHAR(256)
TRIML	Trim Length	Length NUMV Source CHAR Trim Char CHAR(1)
VERIFY IB 2 ext	Verify	Invalid pos. BINV, BINA Source CHAR Class CHAR ZER POS

Pointer/Resolution Instructions

OPCODE	Name	Operands and Conditions
CMPPTRA IB req 2 ext	Compare Pointers for Object Addressability	Compare 1 PTR Compare 2 PTR EQ NEQ
CMPPSPAD IB req 4 ext	Compare Pointers for Space Addressability	Compare 1 SPCPTR, DTAPTR Compare 2 SPCPTR, DTAPTR, NUMV, NUMA, CHARV, CHARA HI LO EQ UNEQ
CMPPTRE IB req 2 ext	Compare Pointers for Equality	Compare 1 PTR Compare 2 PTR EQ NEQ
CMPPTRT IB req 2 ext	Compare Pointer Types	Compare PTR Type CHAR(1), * EQ NEQ
CPYBWP	Copy Bytes with Pointer	Receiver CHARV, PTR Source CHAR, PTR, * Shorter of Receiver and Source
MATPTR	Materialize Pointer	Receiver SPCPTR Pointer PTR
MATPTRIF	Materialize Pointer Information	Receiver SPCPTR Pointer PTR Mask CHAR(4)
MATPTRL	Materialize Pointer Locations	Receiver SPCPTR Source SPCPTR Length BIN
MATCTX	Materialize Context	Receiver SPCPTR Context SYSPTR, * Options CHARF
RSLVDP	Resolve Data Pointer	Pointer DTAPTR Object CHAR(32)F, * Program SYSPTR, *
RSLVSP	Resolve System Pointer	Pointer SYSPTR Object CHAR(34)F, * Context SYSPTR, * Authority CHAR(2)F, *
ADDSPP	Add Space Pointer	Result SPCPTR Source SPCPTR Increment BIN
CMPSPAD IB req 4 ext	Compare Space Addressability	Compare 1 NUMV, NUMA, CHARV, CHARA, PTR, PTRA Compare 2 NUMV, NUMA, CHARV, CHARA, PTRDOA HI LO EQ UNEQ
SETDP	Set Data Pointer	Result DTAPTR Source NUMV, NUMA, CHARV, CHARA
SETDPADR	Set Data Pointer Addressability	Result DTAPTR Source NUMV, NUMA, CHARV, CHARA
SETDPAT	Set Data Pointer Attributes	Result DTAPTR Attributes CHAR(7)F
SETSPP	Set Space Pointer	Result SPCPTR Source NUMV, NUMA, CHARV, CHARA, PTRDO
SETSPPD	Set Space Pointer with Displacement	Result SPCPTR Source NUMV, NUMA, CHARV, CHARA, PTRDO Displacement BIN
SETSPPFP	Set Space Pointer from Pointer	Result SPCPTR Source DTAPTR, SYSPTR, SPCPTR
SETSPPO	Set Space Pointer Offset	Result SPCPTR Source BIN
SETSPFP	Set System Pointer from Pointer	Result SYSPTR Source DTAPTR, SYSPTR, SPCPTR, INSPTR
STSPPO	Store Space Pointer Offset	Result BINV Source SPCPTR

OPCODE	Name	Operands and Conditions
SUBSPP	Subtract Space Pointer Offset	Result SPCPTR Source SPCPTR Decrement BIN
SUBSPPFO	Subtract Space Pointers For Offset	Offset Diff BIN(4)V Pointer SPCPTR Pointer SPCPTR

Space Management Instructions

OPCODE	Name	Operands and Conditions
CRTS	Create Space	Space SYSPTR Template SPCPTR
DESS	Destroy Space	Space SYSPTR
MATS	Materialize Space Attributes	Result SPCPTR Space SYSPTR
MODS	Modify Space Attributes	Space SYSPTR Template BIN, CHAR(28)

Independent Index Instructions

OPCODE	Name	Operands and Conditions
CRTINX	Create Independent Index	Index SYSPTR Template SPCPTR
DESINX	Destroy Independent Index	Index SYSPTR
FNDINXEN	Find Independent Index Entry	Result SPCPTR Index SYSPTR Options SPCPTR Argument SPCPTR
INSINXEN	Insert Independent Index Entry	Index SYSPTR Argument SPCPTR Options SPCPTR
RMVINXEN	Remove Independent Index Entry	Receiver SPCPTR, * Index SYSPTR Options SPCPTR Argument SPCPTR
MATINXAT	Materialize Independent Index Attributes	Receiver SPCPTR Index SYSPTR
MODINX	Modify Independent Index	Index SYSPTR Modifs CHAR(4)

Authorization Instructions

OPCODE	Name	Operands and Conditions
MATAU	Materialize Authority	Result SPCPTR Object SYSPTR User Profile SYSPTR, *
MATAUOBJ	Materialize Authorized Objects	Result SPCPTR Object SYSPTR Options CHAR(1)F, CHAR(*)
MATAL	Materialize Authority List	Receiver SPCPTR List SYSPTR Options SPCPTR
MATAUU	Materialize Authorized Users	Result SPCPTR Object SYSPTR Options CHAR(1)F
MATUP	Materialize User Profile	Result SPCPTR User Profile SYSPTR
MATUPID	Materialize User Profile Pointers from ID	Result SPCPTR Template SPCPTR
MODINVAU	Modify Inv. Auth. Attrs	Template CHAR(1)
TESTAU IB 2 ext	Test Authority	Result CHAR(2)V, * Object SYSPTR, PTRDO User Profile CHAR(2)F AUTH NAUTH

OPCODE	Name	Operands and Conditions
TESTEAU IB 2 ext	Test Authority Extended	Receiver CHAR(8)V, * Authority CHAR(8) Invocation BIN(2), * AUTH NAUTH
TESTULA IB 2 ext	Test User List Authority	Receiver SPCPTR, * System Obj. SYSPTR Authority SPCPTR AUTH NAUTH

Program and Invocation Instructions

OPCODE	Name	Operands and Conditions
MATBPGM	Materialize Bound Program	Result SPCPTR Program SYSPTR
MATPG	Materialize Program	Result SPCPTR Program SYSPTR
ACTBPGM	Activate Program	Template SPCPTR Program SYSPTR
ACTPG	Activate Program	Program SPCPTR, SYSPTR, PTRDO Program SYSPTR
CALLX	Call External	Program SYSPTR, PTRDO Arguments OL, * Returns IDL, *
CALLI	Call Internal	Entry Point ENTRY Arguments OL, * Return INSPTR
CLRIEXIT	Clear Invocation Exit	
DEACTPG	De-Activate Program	Program SYSPTR, *
PEND	Program End	
FNDRINVN	Find Relative Invocation Number	Inv Number BIN(4)V Range CHAR(48)F, * Criterion SPCPTR
MATACTAT	Materialize Activation Attributes	Receiver SPCPTR Act. Mark UBIN(4) Selection CHAR(1)
MATAGPAT	Materialize Activation Group Attributes	Receiver SPCPTR Act. Mark UBIN(4) Selection CHAR(1)
MATINV	Materialize Invocation	Receiver SPCPTR Selection SPCPTR
MATINVAT	Materialize Invocation Attributes	Receiver SPCPTR Invocation CHAR(48)F, * Selection SPCPTR
MATINVE	Materialize Invocation Entry	Receiver SPCPTR Selection CHAR(8)F, * Options CHAR(1)F, *
MATINVS	Materialize Invocation Stack	Receiver SPCPTR Process SYSPTR, *
MODASA	Modify Automatic Storage Allocation	Storage PTRDO, * Size BIN
NOOP	No Operation	
NOOPS	No Operation and Skip	Skip Count UIMM
OVRPGATR	Override Program Attributes	Attr ID UIMM Modifier UIMM
RINZSTAT	Reinitialize Static Storage	Activation SPCPTR to Template
RTX	Return from External	Return BIN(2), *
SETALLEN	Set Argument List Length	Arguments OL Length BIN
SETIEXIT	Set Invocation Exit	Program SYSPTR Arguments OL, *
STPLLEN	Store Parameter List	Length BINV
XCTL	Transfer Control	Program SYSPTR, SPCPTR Arguments OL, *
MATPRATR	Materialize Process Attributes	Result SPCPTR Process SYSPTR, *

OPCODE	Name	Operands and Conditions	
		Options	*CHAR(1)*
MATPRAGP	*Materialize Process Activation Group*	*Receiver*	*SPCPTR*
WAITTIME	*Wait on Time*	*Wait*	*CHAR(16)*
YIELD	*Yield Timeslice*		

Exception Management Instructions

OPCODE	Name	Operands and Conditions	
MATEXCPD	*Materialize Exception Description*	*Result* *Description* *Options*	*SPCPTR* *EXCM* *CHAR(1)*
MODEXCPD	*Modify Exception Description*	*Description* *Modifs* *Options*	*EXCM* *SPCPTR, CHAR(2)C* *CHAR(1)*
RETEXCPD	*Retrieve Exception Data*	*Result* *Options*	*SPCPTR* *CHAR(1)F*
RTNEXCP	*Return from Exception*	*Invocation*	*SPCPTR*
SNSEXCPD	*Sense Exception Description*	*Receiver* *Invocation* *Exception*	*SPCPTR* *SPCPTR to Template* *SPCPTR to Template*
SIGEXCP *IB* *2 ext*	*Signal Exception*	*Signal* *Exception* *IGN DEF*	*SPCPTR* *SPCPTR*
TESTEXCP *IB* *1 ext*	*Test Exception*	*Receiver* *Description* *SIG*	*SPCPTR* *EXCM*

Queue Management Instructions

OPCODE	Name	Operands and Conditions	
CRTQ	*Create Queue*	*Queue* *Template*	*SYSPTR* *SPCPTR*
DEQ *IB* *2 ext*	*Dequeue*	*Prefix* *Message* *Queue* *DQ NDQ*	*CHAR* *SPCPTR* *SYSPTR*
DESQ	*Destroy Queue*	*Queue*	*SYSPTR*
ENQ	*Enqueue*	*Queue* *Prefix* *Message*	*SYSPTR* *CHAR* *SPCPTR*
MATPRMSG	*Materialize Process Message*	*Receiver* *Message* *Source* *Selection*	*SPCPTR* *SPCPTR* *SPCPTR* *SPCPTR*
MATQAT	*Materialize Queue Attributes*	*Result* *Queue*	*SPCPTR* *SYSPTR*
MATQMSG	*Materialize Queued Messages*	*Result* *Queue* *Selection*	*SPCPTR* *SYSPTR* *CHAR(16)*

Object Lock Instructions

OPCODE	Name	Operands and Conditions	
LOCK	Lock Objects	Objects	SPCPTR
LOCKOL	Lock Object Location	Template	SPCPTR
LOCKSL	Lock Space Location	Location Lock Type	PTRDO CHAR(1), *
MATAOL	Materialize Allocated Object Locks	Receiver Object	SPCPTR SYSPTR, PTRDO
MATDRECL	Materialize Data Space Record Locks	Receiver Selection	SPCPTR SPCPTR
MATOBJLK	Materialize Object Locks	Receiver Object	SPCPTR SYSPTR, PTRDO
MATPRLK	Materialize Process Locks	Receiver Process	SPCPTR SYSPTR, *
MATPRECL	Materialize Process Record Locks	Receiver Selection	SPCPTR SPCPTR
MATSELLK	Materialize Selected Locks	Receiver Object	SPCPTR SYSPTR, PTRDO
XFRLOCK	Transfer Object Lock	Process Template	SYSPTR SPCPTR
UNLOCKOL	Unlock Object Location	Template	SPCPTR
UNLOCKSL	Unlock Space Location	Space Lock Type	PTRDO CHAR(1), *

Context Management Instructions

OPCODE	Name	Operands and Conditions	
CRTMTX	Create Pointer-Based Mutex	Mutex Template Result	SPCPTR SPCPTR BIN(4)V
DESMTX	Destroy Pointer-Based Mutex	Mutex Options Result	SPCPTR SPCPTR BIN(4)V
LOCKMTX	Lock Pointer-Based Mutex	Mutex Template Result	SPCPTR SPCPTR BIN(4)V
UNLKMTX	Unlock Pointer-Based Mutex	Mutex Result	SPCPTR BIN(4)V

Heap Management Instructions

OPCODE	Name	Operands and Conditions	
ALCHSS	Allocate Heap Space Storage	Heap Space Heap ID Size	SPCPTR BIN(4)V BIN(4)
CRTHS	Create Heap Space	Heap ID Template	BIN(4)V SPCPTR
DESHS	Destroy Heap Space	Heap ID	BIN(4)V
FREHSS	Free Heap Space Storage	Heap Space	SPCPTR
FREHSSMK	Free Heap Space Storage from Mark	Mark ID	PTRDO
MATHSAT	Materialize Heap Space Attributes	Receiver Heap ID Selection	SPCPTR SPCPTR to Template CHAR(1)
REALCHSS	Reallocate Heap Space Storage	Heap Space Size	SPCPTR BIN(4)
SETHSSMK	Set Heap Space Storage Mark	Mark ID Heap ID	PTRDO BIN(4)

Resource Management Instructions

OPCODE	Name	Operands and Conditions
CRMD	Compute Resource Management Data	Result SPCPTR Source SPCPTR Control CHAR(8)
ENSOBJ	Ensure Object	Object SYSPTR
MATAGAT	Materialize Access Group Attributes	Receiver SPCPTR Access Group SYSPTR
MATRMD	Materialize Resource Management Data	Receiver SPCPTR Control CHAR(8)
SETACST	Set Access State	Template SPCPTR
MATDMPS	Materialize Dump Space	Receiver SPCPTR Dump Space SYSPTR
MATJPAT	Materialize Journal Port Attributes	Receiver SPCPTR Journal Port SYSPTR, PTRDO
MATJPAT	Materialize Journal Space Attributes	Receiver SPCPTR Journal SPC SYSPTR
MATINAT	Materialize Instruction Attributes	Receiver SPCPTR Selection CHAR(16)
MATSOBJ	Materialize System Object	Receiver SPCPTR Object SYSPTR
MATMATR	Materialize Machine Attributes	Receiver SPCPTR Selection CHAR(2), SPCPTR
MATMDATA	Materialize Machine Data	Receiver CHARV Options CHAR(2), UBIN(2), UIMM
TESTTOBJ IB 2 ext	Test Temporary Object	Object SYSPTR EQ NEQ

MI Support Functions Instructions

OPCODE	Name	Operands and Conditions
GENUUID	Generate Universal Unique Identifier	Result SPCPTR
DIAG	Diagnose	Function BIN Argument SPCPTR

Date/Time/Timestamp Instructions

OPCODE	Name	Operands and Conditions
CDD	Compute Date Duration	Duration PKDV Date to CHAR Date from CHAR Template SPCPTR
CTSD	Compute Time Duration	Duration PKDV Time to CHAR Time from CHAR Template SPCPTR
CTSD	Compute Timestamp Duration	Duration PKDV Timestamp to CHAR Timestamp fr CHAR Template SPCPTR
CVTD	Convert Date	Result date CHARV, PKDV, ZNDV Source date CHAR, PKD, BIN Template SPCPTR
CVTT	Convert Time	Result time CHARV Source time CHAR Template SPCPTR
CVTTS	Convert Timestamp	Result TS CHARV Source TS CHAR Template SPCPTR
DECD	Decrement Date	Result date CHARV Source date CHAR Duration PVD Template SPCPTR

OPCODE	Name	Operands and Conditions
DECT	*Decrement Time*	*Result time CHARV* *Source time CHAR* *Duration PVD* *Template SPCPTR*
DECTS	*Decrement Timestamp*	*Result TS CHARV* *Source TS CHAR* *Duration PVD* *Template SPCPTR*
INCD	*Increment Date*	*Result date CHARV* *Source date CHAR* *Duration PVD* *Template SPCPTR*
INCT	*Increment Time*	*Result time CHARV* *Source time CHAR* *Duration PVD* *Template SPCPTR*
INCTS	*Increment Timestamp*	*Result TS CHARV* *Source TS CHAR* *Duration PVD* *Template SPCPTR*