FORTRAN 77 Environment Manual (AOS/VS)

093-000288-01

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Ordering No. 093-000288

Printed in the United States of America
Revision 01, January 1984

FORTRAN 77
Environment Manual
(AOS/VS)
093-000288-01

Revision History:	Effective with:
Original Release - April 1983	FORTRAN 77 Rev. 2.10
First Revision - January 1984	

CONTENT UNCHANGED
The content in this revision is unchanged from 093-000288-00. This revision changes only printing and binding details.

Preface

As a programmer fluent in FORTRAN 77 (F77) and familiar with the Advanced Operating System/Virtual Storage (AOS/VS), you will find this environment manual a useful companion to the *FORTRAN 77 Reference Manual* (093-000162). In addition, if you know Data General/Database Management (DG/DBMS) software, this environment manual helps you process DG/DBMS data via F77 statements.

Organization

We have organized this manual as follows.

Chapter 1	Summarizes the software environment in which FORTRAN 77 exists.
Chapter 2	Documents the utility subprograms your FORTRAN 77 programs can access.
Chapter 3	Explains how your FORTRAN 77 programs can directly use AOS/VS (i.e., make system calls) at runtime.
Chapter 4	Presents the general concepts of multitasking. We also detail the individual multitasking subroutines.
Chapter 5	Summarizes debugging. We introduce the SWAT™ program as a valuable aid to debugging.
Chapter 6	Explains subprograms. It shows how to write assembly language subprograms for FORTRAN 77 programs to CALL and how to write FORTRAN 77 subprograms that BASIC, C, COBOL, PASCAL, and PL/I programs can access.
Chapter 7	Gives several hints about writing better FORTRAN 77 programs.
Chapter 8	Introduces the FORTRAN 77 preprocessor interface to DG/DBMS. It also describes the relationship of the preprocessor to DG/DBMS, AOS/VS, and FORTRAN 77.
Chapter 9	Describes DG/DBMS subschemas for FORTRAN 77 programs and how they are generated.
Chapter 10	Briefly defines the function of every Data Manipulation Language (DML) statement and describes their use.
Chapter 11	Describes the exact syntax of every DML statement.
Chapter 12	Explains how to compile and link your FORTRAN 77/DBMS programs by using the preprocessor and runtime routines.
Chapter 13	Contains two sample FORTRAN 77 programs that interface with DG/DBMS.
Chapter 14	Describes DBMS usage issues you must be familiar with. It also lists the restrictions that the F77/DBMS interface imposes.
Chapter 15	Contains a list of DBMS error messages.
Appendix A	Describes FORTRAN 77 heap and stack organization, and changes you can make to it.

Related Documentation

Other manuals you may find useful are as follows.

Manual Title	Manual No.
A Guide to Using the Data General/Database Management System (DG/DBMS)	069-000025
Command Line Interpreter (CLI) (AOS and AOS/VS) User's Manual	093-000122
FORTRAN 5 Programmer's Guide (AOS)	093-000154
Data General/Database Management System (DG/DBMS) Reference Manual	093-000163
Advanced Operating System/Virtual Storage (AOS/VS) Programmer's Manual	093-000241
Advanced Operating System/Virtual Storage (AOS/VS) Macroassembler (MASM) Reference Manual	093-000242
Advanced Operating System/Virtual Storage (AOS/VS) Operator's Guide	093-000244
Advanced Operating System/Virtual Storage (AOS/VS) Link and Library File Editor (LFE) User's Manual	093-000245
SWAT™ Debugger User's Manual	093-000258

In addition, Data General *strongly* recommends that you have the Software Release Notices and Update Notices for FORTRAN 77 and related software. These Notices may contain corrections to this manual and additional information beyond the scope of this manual. For example, the documentation for the subroutine to obtain the system date appeared in Release Notices before this manual was written. And, they may contain suggestions for corrections or adjustments to current software problems.

Reader, Please Note:

We use these conventions for command formats in this manual:

COMMAND required *[optional]* ...

Where	Means
COMMAND	You must enter the command (or its accepted abbreviation) as shown.
required	You must enter some argument (such as a filename). Sometimes, we use: $\left\{ \begin{matrix} \text{required}_1 \\ \text{required}_2 \end{matrix} \right\}$ which means you must enter *one* of the arguments. Don't enter the braces; they only set off the choice.
[optional]	You have the option of entering this argument. Don't enter the brackets; they only set off what's optional.
...	You may repeat the preceding entry or entries. The explanation will tell you exactly what you may repeat.

Additionally, we use certain symbols in special ways:

Symbol	Means
↲	Press the NEW LINE or carriage return (CR) key on your terminal's keyboard.
□	Be sure to put a space here. (We use this only when we must; normally, you can see where to put spaces.)

All numbers are decimal unless we indicate otherwise; e.g., 35_8.

Finally, in examples we use

THIS TYPEFACE TO SHOW YOUR ENTRY↲
THIS TYPEFACE FOR SYSTEM QUERIES AND RESPONSES.

) is the CLI prompt.

Contacting Data General

- If you have comments on this manual, please use the prepaid Remarks Form that appears after the Index. We want to know what you like and dislike about this manual.
- If you need additional manuals, please use the enclosed TIPS order form (USA only) or contact your Data General sales representative.
- If you experience software problems, please notify Data General Systems Engineering.

End of Preface

Contents

Chapter 1 - Introductory Concepts

Chapter 2 - Utility Runtime Routines

Chapter 3 - System Call Interface

Chapter 4 - Multitasking

Chapter 5 - Debugging

Chapter 6 - Subprograms

Chapter 7 - Programming Hints

Chapter 8 - Introduction to DG/DBMS

Chapter 9 - DG/DBMS Subschema Data Definition

Chapter 10 - Data Manipulation Statements for DG/DBMS

Chapter 11 - Data Manipulation Language Syntax for DBMS

Chapter 12 - How to Compile and Link F77/DBMS Programs

Chapter 13 - Sample FORTRAN 77 Application Programs

Chapter 14 - DBMS Usage Considerations

Chapter 15 - DG/DBMS Error Messages

Appendix A - Runtime Memory Configuration

Tables

Table

Illustrations

Figure

Chapter 1
Introductory Concepts

This chapter gives you an overview of the "forest" of FORTRAN 77 and related software. Subsequent chapters explain the "trees" of Data General extensions to ANSI Standard FORTRAN 77 (F77). The *FORTRAN 77 Reference Manual* explains the "trees" of standard-conforming F77 statements and of compilation/linking procedures.

A Software Summary

As an AOS/VS F77 programmer on Data General (DG) hardware, you are familiar with many F77 program statements, instructions to the compiler and Link programs, and other software. Figure 1-1 shows some of this software.

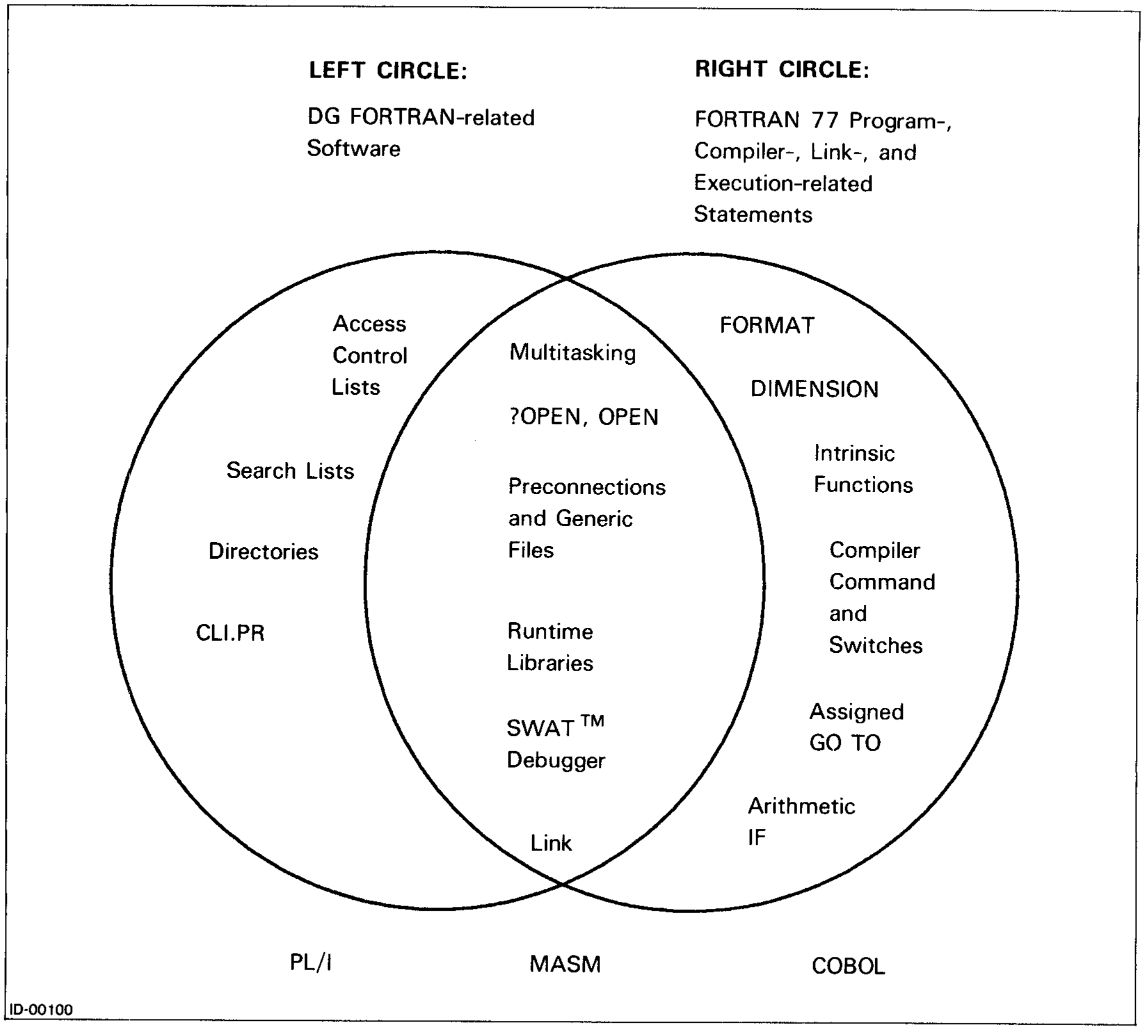


Figure 1-1. Selected Data General Software

This diagram somewhat arbitrarily classifies much of the Data General software that you are (or may want to become) familiar with. In the diagram:

- The *FORTRAN 77 Reference Manual* explains all of the right-hand part of the right circle and some of the overlapping area.
- This environment manual explains none of the right-hand part of the right circle and most of the overlapping area. It extends the reference manual's description of the important Link program.
- Neither manual gives many details about the left-hand part of the left circle. It's sufficient to say that incorrect access control lists, search lists, directories, and generic file assignments have caused many programmers much grief over the years. Be sure yours are correct.
- A program written in one language can CALL a subprogram written in another language. For example, COBOL appears outside both of the diagram's circles. Chapter 6 contains an example of a COBOL program that CALLs a FORTRAN 77 subroutine to perform some number crunching.

The Significance of AOS/VS

Your F77 programs run under AOS/VS. This is a very important statement, because among other things, AOS/VS:

- Handles all file placement and organization.
- Handles all file access commands from your program.
- Allows multitasked processes.

For example, consider the F77 statement

```
READ (2) RECORD
```

When the resulting compiler-generated and Linked machine language instructions execute at runtime, they request AOS/VS (which is also executing in primary storage) to perform an I/O operation. More specifically, these machine language instructions set up and make a ?READ system call. *It is the instructions in this system call* that direct the unformatted transfer of data from the file connected to unit 2 to the variable or array whose name is RECORD. Thus, F77 needs AOS/VS to do any useful processing.

A programmer once told the writer of this manual that "A user program is merely an exit from the operating system." He's right. A user program executes only temporarily; AOS/VS always executes. Furthermore, consider the F77 STOP statement. When its resulting instructions in a program file execute, they tell AOS/VS to terminate the current process and return to the father process. That is, at runtime STOP results in a ?RETURN system call to transfer control back to the father process. This process is normally the Command Line Interpreter (CLI).

The Significance of Link and the Runtime Libraries

If you're familiar with Link and its construction of F77 program files from the runtime libraries, then skip this section.

Many introduction-to-data-processing textbooks contain statements equivalent to: "The FORTRAN compiler translates the FORTRAN source program to a machine language object program. The computer then places this object program in primary storage. Its instructions execute to process data as specified in the FORTRAN source program." These statements are *not* entirely true for Data General's (and most other computer manufacturers') implementation of F77.

The FORTRAN 77 compiler (F77.PR) is a large and complicated program that does create an object (.OB) file from a source (.F77) file. The object file is incomplete because it does not contain all the instructions necessary to carry out the directions of the source program. Where do these missing instructions come from? Program LINK.PR obtains them from other .OB files and from library (.LB) files. LINK.PR creates an executable program file (.PR) based on the compiler-created .OB file and these other .OB files.

As an example, consider the following FORTRAN 77 program SAMPLE.F77. We've numbered its statements for ease of reference.

```
 1      PROGRAM SAMPLE
 2      REAL*8 VARIABLE_1
 3      INTEGER*4 ITIME(3), MY_SUM, J
 4      CALL TIME (ITIME)
 5      MY_SUM = 5 + 4
 6      J = IAND(8,MY_SUM)
 7      PRINT *, 'GIVE ME VARIABLE_1 (XXXX.XX) '
 8      READ (11, 20) VARIABLE_1
 9   20 FORMAT (F7.2)
10      STOP '- THAT IS ALL!'
11      END
```

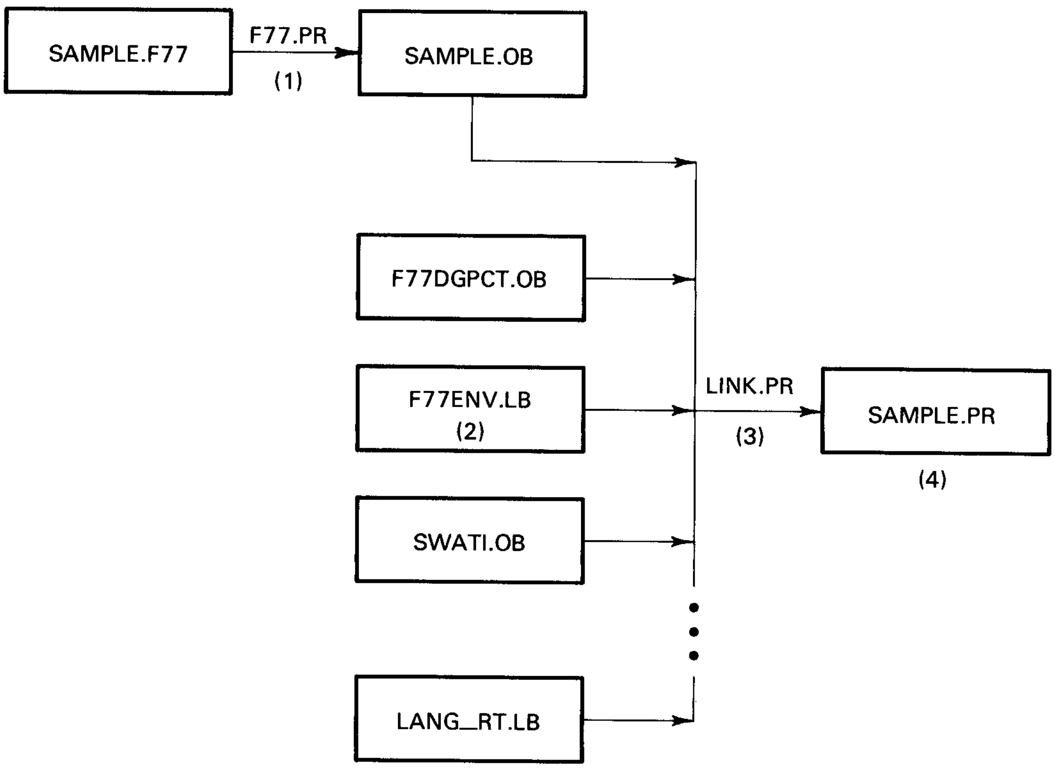
The compilation, link, and execution commands you give to the CLI are:

```
F77 SAMPLE
F77LINK SAMPLE
XEQ SAMPLE
```

Next is a summary of what these three commands do to selected statements in SAMPLE.F77:

- The F77 compiler processes statement 4 by, among other things, creating a note in SAMPLE.OB to LINK.PR. This note tells LINK.PR to insert instructions from TIME.OB into SAMPLE.PR. Then:
    - LINK.PR follows F77LINK.CLI's instructions and searches the runtime libraries to find TIME.OB (in F77ENV.LB).
    - When SAMPLE.PR executes and it reaches the instructions from TIME.OB, they make a ?GTOD system call to obtain the time of day.
    - The respective contents of ITIME(1), ITIME(2), and ITIME(3) are the current hour, minute, and second.
- The F77 compiler reacts to statement 5 by creating self-contained instructions in SAMPLE.OB. These instructions make no reference subroutine; they execute at runtime to perform statement 5 by themselves. We can also say that the compiler generates *in-line code* from statement 5.
- Statement 6 results in the compiler's creation of in-line code for the intrinsic function IAND. The code includes a WAND instruction. At runtime WAND executes to find the logical AND of the 4-byte integer 8 and of the 4-byte integer in the variable MY_SUM.
- Statements 8 and 9 result in several instructions in SAMPLE.OB, and then many more instructions in SAMPLE.PR. At runtime these SAMPLE.PR instructions:
    - Obtain a string of ASCII characters from @INPUT.
    - Check for an illegal character string (such as '027A.38') and report an error if it occurs.
    - Convert the legal character string to a double-precision floating-point number and move it to the 8 bytes that VARIABLE_1 references.

Figure 1-2 also summarizes the three commands that compile, link, and execute program SAMPLE.



Note (1): Macro F77.CLI executes program F77.PR.
Note (2): Object file TIME.OB is part of library file F77ENV.LB.
Note (3): Macro F77LINK.CLI executes program LINK.PR.
Note (4): The CLI command XEQ SAMPLE executes program SAMPLE.PR.

ID-00101

Figure 1-2. The Compilation, Linking, and Execution of a Typical F77 Program

Link doesn't insert all the .OB files listed in Figure 1-2 into SAMPLE.PR. For example, SWATI.OB goes into SAMPLE.PR only if the F77LINK command includes the /DEBUG global switch. The SWAT Debugger requires SWATI.OB. Chapter 5 summarizes the SWAT Debugger. You can print F77LINK.CLI to see the names of all Data-General-created runtime library files.

If you're curious about the .OB files that Link places into a .PR file, use the /B and /L switches to create a load map file. In our case, we replace the CLI command

```
F77LINK SAMPLE
```

with

```
DELETE/2=IGNORE SAMPLE.MAP
F77LINK/B/L=SAMPLE.MAP SAMPLE
TYPE SAMPLE.MAP
```

Normally, you don't have to worry about the details of F77.PR and LINK.PR. You also don't have to know which .OB files are in which .LB files. Just be sure that the F77 and F77LINK commands are correct for each program you write.

One problem arises when you've created a .OB or .LB file whose name matches a Data-General-supplied .OB or .LB file. Link may find and select your .OB or .LB file instead of the correct file intended for the current revision of F77.

To obtain the names of the Data-General-supplied .OB and .LB files that F77LINK uses, simply print F77LINK.CLI. Typically, its pathname is :UTIL:F77:F77LINK.CLI. Then, make sure that none of your filenames matches those in F77LINK.CLI.

The Significance of the Release and Update Notices

It's hard to overemphasize the necessity of having the latest Release and Update Notices for FORTRAN 77 and for related software such as Link. This manual assumes throughout that you have the latest such Notices. Together, they give you the most current information Data General has available on the software you need to write and maintain FORTRAN 77 programs. An F77 Reference or Environment manual is incomplete by itself, just like a solitary Release or Update Notice. Read them all!

End of Chapter

Chapter 2
Utility Runtime Routines

FORTRAN 77 provides many subprograms (both subroutines and external functions) that process data in a variety of ways. This data processing includes program/system runtime interface, which Chapter 3 explains, and multitasking, which Chapter 4 explains. The subprograms also perform various utility functions such as obtaining the date. We document these utility subprograms in this chapter.

NOTE: You don't have to specify any F77 utility subprogram names to the F77LINK macro. F77LINK has Link search all the runtime library files that contain the utility subprograms.

Documentation Categories

The rest of this chapter describes the utility subprograms alphabetically. The explanation of each subprogram includes:

- Its name and function.
- Its format and argument names.
- Descriptions of each argument.
- A sample program that uses the subprogram.

DATE
Obtain the system date.

Format
CALL DATE(date_array)

Argument
date_array — is an INTEGER*4 array into whose first three elements DATE will place the current date from AOS/VS:

First element	— AD year since zero
Second element	— Month, between 1 and 12 inclusive
Third element	— Day, between 1 and 31 inclusive

NOTE: Routine DATE conforms to the ISA S61.1 standard.

Example

```
C       SAMPLE AOS/VS F77 PROGRAM CALL_DATE
        DIMENSION IDATE(3)
C       ...
        CALL DATE (IDATE)
C       PRINT THE DATE IN MONTH/DAY/YEAR FORMAT.
        PRINT *, 'Date is ', IDATE(2), '/', IDATE(3), '/', IDATE(1)-1900
C       ...
        STOP
        END
```

ERRCODE

Report a runtime error based on an error code and an optional severity number.

Format

CALL ERRCODE(code *[,sev]*)

Arguments

code — is an INTEGER*4 expression that contains the code you want ERRCODE to report on. Typically, this might be the value of the IOSTAT= variable from an I/O statement or the result code from the system interface function ISYS. File ERR.F77.IN contains PARAMETER statements for the current values of code that F77 defines for its runtime system. If code is 0, ERRCODE merely returns and writes no output.

NOTE: Be sure your system error message file (usually :ERMES) contains messages from F77 and the AOS/VS Common Language Library. See the current F77 and Language Library Release Notices for instructions to create this file.

sev — is an optional INTEGER*4 expression that contains the severity you assign to the error. If *sev* is

0:	Nonfatal — the task continues execution.
1:	Task fatal — the task terminates in an orderly fashion.
not 0 or 1:	Process fatal — the program terminates in an orderly fashion.
not supplied:	Process fatal — the program terminates in an orderly fashion.

Relation to Error Logging

A CALL to ERRCODE results in output to all units OPENed with ERRORLOG='YES' or, if currently no units are OPEN in this way, to @OUTPUT.

Relation to ERRTEXT

The ERRCODE and ERRTEXT (described next) subroutines have quite similar functions. The most significant difference is that you supply ERRCODE a numeric code argument, whereas you supply ERRTEXT a character text argument. ERRTEXT always writes a diagnostic message, while ERRCODE does so when, and only when, the value of its argument code differs from zero.

ERRCODE (continued)

Example Program

Program TEST_ERRCODE lets us vary the values of the ERRCODE arguments code and *sev*. Its listing is below; an example of its execution follows. If you decide to execute this program, we suggest you select values of code from file ERR.F77.IN at runtime.

```
C       TEST PROGRAM TEST_ERRCODE TO TEST SUBROUTINE ERRCODE.

        INTEGER*4 ERROR_CODE, SEVERITY, Y_OR_N

   10   WRITE (6, 20)
   20   FORMAT (1H0, 'GIVE ME A DECIMAL ERROR CODE AND A SEVERITY', /,
    1           1X,  '    NUMBER SEPARATED BY A COMMA.', /,
    2           1X,  '    THE SEVERITY NUMBER SHOULD BE 0 OR 1.', /,
    3           1X,  'WHAT ARE THESE NUMBERS? ', $)
        READ(5,*) ERROR_CODE, SEVERITY
        PRINT *,' '
        PRINT *,'NOW COMES THE CALL TO ERRCODE(ERROR CODE, SEVERITY NUMBER)'
        PRINT *, '-----'
        CALL ERRCODE (ERROR_CODE, SEVERITY)
        PRINT *, '-----'
   30   PRINT *, ' '

C       THE FOLLOWING STATEMENTS EXECUTE ONLY WHEN   SEVERITY   IS ZERO.
        WRITE (6, 40)
   40   FORMAT (1X, 'DO YOU WANT TO ENTER ANOTHER PAIR OF NUMBERS ',
    1               '(Y OR N) ? _<31>', $) ! <31> BACKSPACES THE CURSOR
        READ (5, 50) Y_OR_N
   50   FORMAT (A1)
        IF ( Y_OR_N .EQ. 'Y   ' ) THEN
                GO TO 10
        ELSEIF ( Y_OR_N .EQ. 'N   ' ) THEN
                PRINT *, 'END OF TESTING OF SUBROUTINE ERRCODE'
                STOP
        ELSE
                PRINT *, '<BEL>YOUR RESPONSE MUST BE   Y   OR   N   .'
                PRINT *, '<BEL>   TRY AGAIN.'
                GO TO 30
        ENDIF

        END
```

```
GIVE ME A DECIMAL ERROR CODE AND A SEVERITY
  NUMBER SEPARATED BY A COMMA.
  THE SEVERITY NUMBER SHOULD BE 0 OR 1.
WHAT ARE THESE NUMBERS?   11264,0 ⤶

NOW COMES THE CALL TO ERRCODE(ERROR CODE, SEVERITY NUMBER)
-----
ERROR       11264.
Invalid unit number

ERROR       11264.

Call Traceback:

from fp=16000005126,   pc=.MAIN+203
from fp=             0,   pc=I.INIT+43

Invalid unit number

-----

DO YOU WANT TO ENTER ANOTHER PAIR OF NUMBERS (Y OR N) ?   Y ⤶

GIVE ME A DECIMAL ERROR CODE AND A SEVERITY
  NUMBER SEPARATED BY A COMMA.
  THE SEVERITY NUMBER SHOULD BE 0 OR 1.
WHAT ARE THESE NUMBERS?   10000,0 ⤶

NOW COMES THE CALL TO ERRCODE(ERROR CODE, SEVERITY NUMBER)
-----
ERROR       10000.
UNKNOWN MESSAGE CODE 00023420

ERROR       10000.

Call Traceback:

from fp=16000005126,   pc=.MAIN+203
from fp=             0,   pc=I.INIT+43

UNKNOWN MESSAGE CODE 00023420

-----
```

```
DO YOU WANT TO ENTER ANOTHER PAIR OF NUMBERS (Y OR N) ?   Y ↓
GIVE ME A DECIMAL ERROR CODE AND A SEVERITY
   NUMBER SEPARATED BY A COMMA.
   THE SEVERITY NUMBER SHOULD BE 0 OR 1.
WHAT ARE THESE NUMBERS?   36,0 ↓
NOW COMES THE CALL TO ERRCODE(ERROR CODE, SEVERITY NUMBER)
-----
ERROR        36.
DEVICE ALREADY IN SYSTEM
ERROR        36.
Call Traceback:
from fp=16000005126,   pc=.MAIN+203
from fp=          0,   pc=I.INIT+43
DEVICE ALREADY IN SYSTEM
-----
DO YOU WANT TO ENTER ANOTHER PAIR OF NUMBERS (Y OR N) ?   N ↓
END OF TESTING OF SUBROUTINE ERRCODE
STOP
```

Please note the following about the execution of TEST_ERRCODE:

- Your frame pointer (fp) and program counter (pc) values probably will differ from those shown.
- The first example shows the outcome if a program had CALLed ERRCODE after an I/O operation returned 11264 as the value of the IOSTAT variable.
- The second example shows what happens if an error code unknown to the system error message file :ERMES is passed to ERRCODE. The F77 Release Notice explains how to construct ERMES so that it contains F77 error codes.
- The third example shows that ERRCODE may respond to more than just nonzero values in ERR.F77.IN. Here, 36 (= 44K) is a valid AOS/VS system error code. ERMES must contain AOS/VS error codes as well as those from F77.
- Program TEST_ERRCODE is compiled without the /LINEID and /PROCID switches. Specifying either or both switches to the compilation macro F77.CLI would have ERRCODE display additional information about the program.

Related Documentation

You may regard subroutine ERRCODE as a natural extension of the software described in the "Runtime Errors" section of the *FORTRAN 77 Reference Manual.*

ERRTEXT

Report a runtime error based on a text string and an optional severity number.

Format

CALL ERRTEXT(text *[,sev]*)

Arguments

text — is a CHARACTER expression that contains the text of the error message that you want ERRTEXT to report.

NOTE: Be sure your system error message file (usually :ERMES) contains messages from F77 and the AOS/VS Common Language Library. See the current F77 and Language Library Release Notices for instructions to create this file.

sev — is an optional INTEGER*4 expression that contains the severity you assign to the error. If *sev* is

- 0: Nonfatal —the task continues execution.
- 1: Task fatal —the task terminates in an orderly fashion.
- not 0 or 1: Process fatal — the program terminates in an orderly fashion.
- not supplied: Process fatal — the program terminates in an orderly fashion.

Relation to Error Logging

A CALL to ERRTEXT results in output to all units OPENed with ERRORLOG='YES' or, if currently no units are OPEN in this way, to @OUTPUT.

Relation to ERRCODE

The ERRTEXT and ERRCODE (described previously) subroutines have quite similar functions. The most significant difference is that you supply ERRTEXT a character text argument, whereas you supply ERRCODE a numeric code argument. ERRCODE writes a diagnostic message when, and only when, the value of its argument code differs from zero, whereas ERRTEXT always writes a diagnostic message.

Example Program

Program TEST_ERRTEXT lets us vary the values of the ERRTEXT arguments text and *sev*. Its listing is below; an example of its execution follows.

```
C       TEST PROGRAM TEST_ERRTEXT TO TEST SUBROUTINE ERRTEXT.

        INTEGER*4 SEVERITY, Y_OR_N
        CHARACTER*70 ERROR_TEXT

   10   WRITE (6, 20)
   20   FORMAT (1H0, 'GIVE ME AN ERROR MESSAGE (UP TO 70 CHARS.)', /,
    1           1X,  '    AND A SEVERITY NUMBER SEPARATED BY A COMMA.', /,
    2           1X,  '    THE SEVERITY NUMBER SHOULD BE 0 OR 1.', /,
    3           1X,  'WHAT ARE THESE ARGUMENTS? ', $)
        READ(5,*) ERROR_TEXT, SEVERITY
        PRINT *,' '
        PRINT *,'NOW COMES THE CALL TO ERRTEXT(ERROR TEXT, SEVERITY NUMBER)'
        PRINT *, '-----'
        CALL ERRTEXT (ERROR_TEXT, SEVERITY)
        PRINT *, '-----'
   30   PRINT *, ' '

C       THE FOLLOWING STATEMENTS EXECUTE ONLY WHEN   SEVERITY   IS ZERO.
        WRITE (6, 40)
   40   FORMAT (1X, 'DO YOU WANT TO ENTER ANOTHER MESSAGE AND NUMBER ',
    1               '(Y OR N) ? _<31>', $) ! <31> BACKSPACES THE CURSOR
        READ (5, 50) Y_OR_N
   50   FORMAT (A1)
        IF ( Y_OR_N .EQ. 'Y    ' ) THEN
                GO TO 10
        ELSEIF ( Y_OR_N .EQ. 'N    ' ) THEN
                PRINT *, 'END OF TESTING OF SUBROUTINE ERRTEXT'
                STOP
        ELSE
                PRINT *, '<BEL>YOUR RESPONSE MUST BE   Y   OR   N   .'
                PRINT *, '<BEL>   TRY AGAIN.'
                GO TO 30
        ENDIF

        END
```

```
GIVE ME AN ERROR MESSAGE (UP TO 70 CHARS.)
   AND A SEVERITY NUMBER SEPARATED BY A COMMA.
   THE SEVERITY NUMBER SHOULD BE 0 OR 1.
WHAT ARE THESE ARGUMENTS?    "SAMPLE ERROR TEXT",0 ⤶

NOW COMES THE CALL TO ERRTEXT(ERROR TEXT, SEVERITY NUMBER)
-----
ERROR         11614.
User defined ERROR text
SAMPLE ERROR TEXT
ERROR         11614.

Call Traceback:

from fp=16000005126,   pc=.MAIN+210
from fp=             0,   pc=I.INIT+43

User defined ERROR text

-----

DO YOU WANT TO ENTER ANOTHER MESSAGE AND NUMBER (Y OR N) ?    Y ⤶

GIVE ME AN ERROR MESSAGE (UP TO 70 CHARS.)
   AND A SEVERITY NUMBER SEPARATED BY A COMMA.
   THE SEVERITY NUMBER SHOULD BE 0 OR 1.
WHAT ARE THESE ARGUMENTS?    "SOME MORE ERROR TEXT",0 ⤶

NOW COMES THE CALL TO ERRTEXT(ERROR TEXT, SEVERITY NUMBER)
-----
ERROR         11614.
User defined ERROR text
SOME MORE ERROR TEXT
ERROR         11614.

Call Traceback:

from fp=16000005126,   pc=.MAIN+210
from fp=             0,   pc=I.INIT+43

User defined ERROR text

-----

DO YOU WANT TO ENTER ANOTHER MESSAGE AND NUMBER (Y OR N) ?    N ⤶

END OF TESTING OF SUBROUTINE ERRTEXT
STOP
```

Please note the following about the execution of TEST_ERRTEXT:

- Your frame pointer (fp) and program counter (pc) values probably will differ from those shown.
- Both examples use list-directed editing because of the

  ```
  READ (5, *) ERROR_TEXT, SEVERITY
  ```

  statement. Thus, quotation marks surround the text given via the console to CHARACTER variable ERROR_TEXT at runtime.
- Both examples show the decimal error code 11614 because this is the error code for user-defined error text.
- Program TEST_ERRTEXT is compiled without the /LINEID and /PROCID switches. Specifying either or both switches to the compilation macro F77.CLI would have ERRTEXT display additional information about the program.

Related Documentation

You may regard subroutine ERRTEXT as a natural extension of the software described in the "Runtime Errors" section of the *FORTRAN 77 Reference Manual.*

EXIT

Terminate the current task.

Subroutine EXIT terminates the calling task. It acts like the F77 STOP statement, but you can't give a number or text string to the subroutine. EXIT returns a null string to the parent process. Thus, for single-task programs, you can use it to halt your program and have it return to the CLI without displaying STOP on the console. In contrast, the F77 STOP statement terminates the process.

Format

CALL EXIT

Arguments

none

Example

```
C       SAMPLE AOS/VS F77 PROGRAM CALL_EXIT
        PRINT *, 'THIS IS THE BEGINNING AND THE END.'
        CALL EXIT
        END
```

Execution of CALL_EXIT.PR results in the following.

) X CALL_EXIT ↵

THIS IS THE BEGINNING AND THE END.
)

RANDOM

Function subprogram to obtain a random number.

Format

RANDOM(ISEED)

Result

The result of a function reference to RANDOM is a REAL*8 number greater than or equal to zero and less than one.

Argument

ISEED is an INTEGER*4 variable or array element. It may *not* be a constant. If ISEED has an initial value

< 0: The initial value of RANDOM(ISEED) depends on the system time of day. Thus, successive references to RANDOM(ISEED) will result in a virtually nonreproducible sequence of random numbers. Don't modify ISEED after assigning it an initial value.

>= 0: The initial value of RANDOM(ISEED) depends on the value of ISEED. To generate a reproducible sequence of random numbers, assign a chosen nonnegative constant to ISEED and then make success references to RANDOM(ISEED). Don't modify ISEED after assigning it an initial value.

RANDOM stores the starting point (seed) for the next number it will generate in the memory location that ISEED refers to. Therefore, ISEED must be a variable and never a constant.

Please note the following.

- Successive references to RANDOM generate a sequence of random numbers with a uniform distribution.
- RANDOM uses Knuth's Linear Congruential Algorithm to create a REAL*8 number based on the value of ISEED. After this creation, RANDOM replaces ISEED with an integer between 0 and 262,143 inclusive. These integers, formed by successive references to RANDOM, are a sequence with a period of 262,144. RANDOM creates a temporary value for ISEED that may exceed 262,143, but the final value of ISEED is MOD(temporary-ISEED,262144).
- Be sure to declare RANDOM as REAL*8 or DOUBLE PRECISION in any program unit that uses this function.

Example Program 1

Figure 2-1 shows program EXAMPLE_RANDOM that uses RANDOM to generate five numbers.

```
      PROGRAM EXAMPLE_RANDOM
      REAL*8 RANDOM, RESULT
      INTEGER*4 ISEED
      ISEED = 0  ! GENERATE A REPRODUCIBLE SEQUENCE OF RANDOM NUMBERS
      DO 10 I = 1, 5
      WRITE (6, 100) I, ISEED
100   FORMAT (1H0, 'BEFORE EXECUTING RANDOM FOR I = ', I1, ', ISEED = ', I7)
      RESULT = RANDOM(ISEED)
      WRITE (6, 110) I, ISEED, RESULT
110   FORMAT (1H , ' AFTER EXECUTING RANDOM FOR I = ', I1,
     1        ', ISEED = ', I7, ' AND RANDOM RETURNS ', F9.6)
 10   CONTINUE
      WRITE (6, 20)
 20   FORMAT (1H0, '*** END OF PROGRAM ***')
      CALL EXIT
      END
```

DG-25213

Figure 2-1. Program EXAMPLE_RANDOM.F77

Figure 2-2 shows the output from program EXAMPLE_RANDOM.

```
BEFORE EXECUTING RANDOM FOR I = 1, ISEED =       0
 AFTER EXECUTING RANDOM FOR I = 1, ISEED =   55397 AND RANDOM RETURNS   .211323

BEFORE EXECUTING RANDOM FOR I = 2, ISEED =   55397
 AFTER EXECUTING RANDOM FOR I = 2, ISEED =  192310 AND RANDOM RETURNS   .733604

BEFORE EXECUTING RANDOM FOR I = 3, ISEED =  192310
 AFTER EXECUTING RANDOM FOR I = 3, ISEED =  182979 AND RANDOM RETURNS   .698009

BEFORE EXECUTING RANDOM FOR I = 4, ISEED =  182979
 AFTER EXECUTING RANDOM FOR I = 4, ISEED =   55324 AND RANDOM RETURNS   .211044

BEFORE EXECUTING RANDOM FOR I = 5, ISEED =   55324
 AFTER EXECUTING RANDOM FOR I = 5, ISEED =  118801 AND RANDOM RETURNS   .453190

*** END OF PROGRAM ***
```

DG-25214

Figure 2-2. The Output from Program EXAMPLE_RANDOM

NOTE: The output from EXAMPLE_RANDOM will always be the same because ISEED has an initial nonnegative value. To generate a virtually nonreproducible sequence of five random numbers, set ISEED to any valid negative integer.

Compare any two successive pairs of lines of output in Figure 2-2. You'll see that RANDOM changes ISEED; the changed value of ISEED becomes input to the next reference to RANDOM. For instance, when I=2, RANDOM uses the ISEED value 55397 to generate .733604; RANDOM changes ISEED to 192310 for input to the next reference to itself.

Example Program 2

Let's look at a program, named ROLL_DICE.F77, that uses RANDOM. This program:

- Simulates the rolling of a pair of fair dice 180 times.
- Counts the number of dots facing up after each roll.
- Computes a number, based on the actual results and the expected results and their differences, after performing all the rolls.
- Uses a standard statistical test, with the computed number, to decide whether or not the differences between the actual and expected results are significant.

Expected Results

We use the following information to calculate the expected results.

Number of Dots Facing up, N	Probability(N) in Each Roll	Expected Value of N in 180 Rolls
2	1/36	1/36 x 180 = 5
3	2/36	2/36 x 180 = 10
4	3/36	3/36 x 180 = 15
5	4/36	4/36 x 180 = 20
6	5/36	5/36 x 180 = 25
7	6/36	6/36 x 180 = 30
8	5/36	5/36 x 180 = 25
9	4/36	4/36 x 180 = 20
10	3/36	3/36 x 180 = 15
11	2/36	2/36 x 180 = 10
12	1/36	1/36 x 180 = 5

Let's look at the second row as an example of all the rows. A pair of dice may land in 6x6=36 different ways on each roll. There are only two ways a total of three dots may appear: the first die shows two dots and the second die one dot, or the first die shows one dot and the second die two dots. The probability of a total of three dots showing is 2/36. Thus, we can *expect* 2/36 of a large number of rolls to have three dots showing. However, we are not *guaranteed* that exactly 2/36 of a large number of rolls will show three dots.

Converting RANDOM(ISEED) to an Integer

Each execution of a statement such as

```
ROLL_RESULT = RANDOM(ISEED)
```

results in a number between 0.0 and 1.0 (including 0.0, excluding 1.0). To simulate the rolling of a die, we must convert each such result to one of the six integers between 1 and 6, inclusive. Let's name this INTEGER*2 variable DOTS. Figure 2-3 shows the necessary conversion between the values of ROLL_RESULT and the corresponding ones of DOTS.

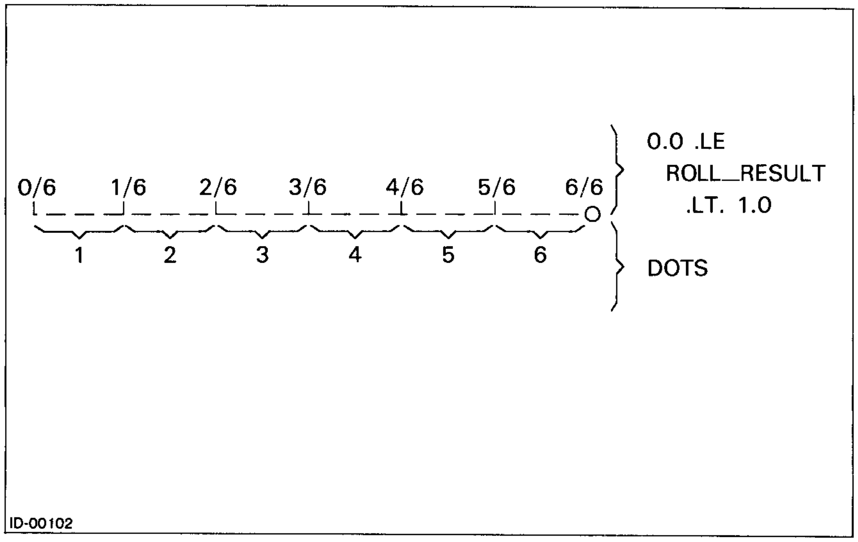


Figure 2-3. A Correspondence Between Selected Real Numbers and Integers

We have divided the real number line between 0.0 and 1.0 into six equal segments, with each segment corresponding to one of the six integers 1, 2, 3, 4, 5, and 6. Now we look for a formula that will take a number between 0.0 and 1.0 — which lies on one of the segments — and compute the proper integer. The formula, as an F77 assignment statement with the variables specified in the previous paragraph, is

```
DOTS = INT( 6.0 * ROLL_RESULT ) + 1
```

For example, suppose that ROLL_RESULT is 0.42. 0.42 is between 2/6 and 3/6. Replacing ROLL_RESULT by 0.42 and evaluating this expression should, according to Figure 2-3, assign 3 to DOTS. Does it?

```
    ?
  3 = INT( 6.0 * 0.42 )  +  1
    ?
  3 = INT(    2.52    )  +  1
    ?
  3 =         2          +  1
    ?
  3 = 3
```

Yes.

Of course, the program will have to execute two such assignment statements to simulate each roll of the pair of dice.

The Decision Rule

Finally, we use the chi-square test from statistics to see if the actual results differ "too much" from the expected results. The formula is

$$\text{chi-square} = \sum_{n=2}^{12} \frac{(\text{dots}_n - \text{expected}_n)^2}{\text{expected}_n}$$

where the Greek letter Sigma represents "the sum of." Another way of expressing the formula for calculating chi-square is

$$\text{chi-square} = \text{sum of } \frac{(\text{actual result} - \text{expected result})^2}{\text{expected result}}$$

If this sum is less than 18.3, we can conclude that RANDOM has generated an acceptable sequence of random numbers between 0.0 and 1.0. Otherwise, we might cast some suspicion on RANDOM and investigate further or else assume the large difference has occurred by chance alone.

A note about statistics:

For those of you with knowledge about statistics:

$P(X^2 >= 18.3$, 10 degrees of freedom$) = 0.05$

And, the expected number of dots showing is five or more for all possible outcomes.

Program ROLL_DICE

Program ROLL_DICE.F77 is shown in Figure 2-4.

```
C       AOS/VS PROGRAM ROLL_DICE TO SIMULATE THE ROLLING OF A
C          PAIR OF FAIR DICE AND TO TEST THE VALIDITY OF THE RESULTS.

        REAL*8 RANDOM           ! RANDOM NUMBER GENERATOR FUNCTION SUBPROGRAM
        REAL*8 ROLL_RESULT      ! RECEIVE OUTPUT FROM   RANDOM   ON
C                                   EACH ROLL OF THE DICE
        REAL*4 CHI_SQUARE /0.0/ ! TO BE COMPUTED

        REAL*4 MAXIMUM_CHI_SQUARE
        INTEGER*2 NUM_ROLLS     ! NUMBER OF ROLLS OF THE DICE
        PARAMETER (MAXIMUM_CHI_SQUARE = 18.3,
     +             NUM_ROLLS = 180)

        INTEGER*2 DOTS_UP_1      ! DOTS SHOWING ON THE FIRST DIE
        INTEGER*2 DOTS_UP_2      ! DOTS SHOWING ON THE SECOND DIE
        INTEGER*2 DOTS_UP       ! DOTS SHOWING ON BOTH DICE AFTER EACH ROLL
        INTEGER*4 ISEED  / -1 / ! START A NEW SEQUENCE OF RANDOM NUMBERS
        INTEGER*2 ACTUAL_RESULTS(2:12)    / 11*0 /
        INTEGER*2 EXPECTED_RESULTS(2:12)  /  5, 10, 15, 20, 25, 30,
     1                                      25, 20, 15, 10,  5       /

        WRITE (6, 20) NUM_ROLLS
   20   FORMAT (1H , '<TAB>RESULTS OF ROLLING A PAIR OF DICE ',I3,' TIMES', /)

        DO 30 I = 1, NUM_ROLLS

C               ROLL A PAIR OF DICE ...
                ROLL_RESULT = RANDOM(ISEED)
                DOTS_UP_1 = 6*ROLL_RESULT + 1  ! 1ST DIE
                ROLL_RESULT = RANDOM(ISEED)
                DOTS_UP_2 = 6*ROLL_RESULT + 1  ! 2ND DIE
                DOTS_UP = DOTS_UP_1 + DOTS_UP_2                      ! BOTH DICE

C               ... AND TALLY THE RESULT.  FOR EXAMPLE, IF   DOTS_UP   IS 5,
C                       THEN   ACTUAL_RESULTS(5)   IS INCREASED BY 1.
                ACTUAL_RESULTS(DOTS_UP) = ACTUAL_RESULTS(DOTS_UP) + 1
   30   CONTINUE
C       DISPLAY THE RESULTS
        WRITE (6, 40)
   40   FORMAT (1H , '<TAB>DOTS        ACTUAL     EXPECTED', /,
     1          1H , '<TAB>SHOWING     COUNT      COUNT   ', /)
        DO 60 I = 2, 12
                WRITE (6, 50) I, ACTUAL_RESULTS(I), EXPECTED_RESULTS(I)
   50           FORMAT (1H , '<TAB>', 2X, I2, 9X, I3, 9X, I3)
   60   CONTINUE

C       CALCULATE CHI-SQUARE
        DO 70 I = 2, 12
        CHI_SQUARE = CHI_SQUARE +
     1     FLOAT(  ( ACTUAL_RESULTS(I) - EXPECTED_RESULTS(I) )**2  )  /
C          --------------------------------------------------------
     2                   FLOAT( EXPECTED_RESULTS(I) )
   70   CONTINUE
```

DG-25215

Figure 2-4. Program ROLL_DICE.F77 (continues)

```
      WRITE (6, 80) MAXIMUM_CHI_SQUARE, CHI_SQUARE
80    FORMAT (1H0, '<TAB>MAXIMUM ALLOWABLE VALUE OF CHI-SQUARE: ', F5.2, /,
  1           1H , '<TAB>ACTUAL VALUE OF CHI-SQUARE:             ', F5.2, /)
      IF ( CHI_SQUARE .LE. MAXIMUM_CHI_SQUARE ) THEN
              PRINT *, '<TAB>CONCLUSION: RANDOM PASSES THIS TEST'
      ELSE
              PRINT *, '<TAB>CONCLUSION: RANDOM FAILS THIS TEST'
      ENDIF
      PRINT *, '<NL><TAB>END OF SIMULATION'
      CALL EXIT
      END
```

DG-25215

Figure 2-4. Program ROLL_DICE.F77 (concluded)

ROLL_DICE Output

Figure 2-5 shows typical output from program ROLL_DICE.

```
RESULTS OF ROLLING A PAIR OF DICE 180 TIMES

DOTS        ACTUAL      EXPECTED
SHOWING     COUNT       COUNT

   2           5           5
   3          10          10
   4          15          15
   5          27          20
   6          26          25
   7          27          30
   8          25          25
   9          24          20
  10           9          15
  11           8          10
  12           4           5

MAXIMUM ALLOWABLE VALUE OF CHI-SQUARE: 18.30
ACTUAL VALUE OF CHI-SQUARE:             6.59

CONCLUSION: RANDOM PASSES THIS TEST

END OF SIMULATION
```

DG-25216

Figure 2-5. Typical Output from Program ROLL_DICE

TIME

Obtain the system time of day.

Format

CALL TIME(time_array)

Argument

time_array — is an INTEGER*4 array into whose first three elements TIME will place the absolute time (based on a 24-hour clock) from AOS/VS:

First element	— Hours, between 0 and 23 inclusive
Second element	— Minutes, between 0 and 59 inclusive
Third element	— Seconds, between 0 and 59 inclusive

NOTE: Routine TIME conforms to the ISA S61.1 standard.

Example

```
C       SAMPLE AOS/VS F77 PROGRAM CALL_TIME
        DIMENSION ITIME(3)
C       ...
        CALL TIME (ITIME)
C       PRINT THE TIME IN HOUR:MINUTE:SECONDS FORMAT.
        PRINT 100, ITIME
  100   FORMAT (' Time is ', I2, ':', I2.2, ':', I2.2)
C       ...
        STOP
        END
```

End of Chapter

Chapter 3
System Call Interface

This chapter almost exclusively explains the system call interface subprogram ISYS. ISYS is an external function that lets your F77 programs have full access to AOS/VS. This chapter also explains the external function subprogram IO_CHAN that returns an AOS/VS channel number.

Basically, you supply arguments to ISYS that represent a system call's name and accumulator values. You obtain these names and values from the *AOS/VS Programmer's Manual* and from your program's requirements. At runtime, F77 attempts an AOS/VS system call in response to each occurrence of ISYS. It returns a value of 0 if the call executed successfully, or else a nonzero value, if it did not. The nonzero value identifies the exceptional condition that occurred.

Structure

The structure of function ISYS is

ISYS (call_name, AC0, AC1, AC2)

where:

call_name	is an INTEGER*4 expression that contains the value of an AOS/VS system call code. This code comes from a statement in SYSID.32.SR that assigns the value to a system call symbol. SYSID.32.SR is normally in :UTIL.
AC0 AC1 AC2	are INTEGER*4 variables or array elements that contain the values you want the corresponding accumulators to have when the system call occurs. After the system call completes, these variables or array elements are defined with the corresponding accumulator values.

Frequently, your program will implement ISYS by means of statements whose general structure is

```
        IER = ISYS (CALL_CODE, AC0, AC1, AC2)
        IF ( IER .NE. 0 ) THEN
C               PLACE ERROR HANDLING ROUTINE HERE
        ENDIF
```

or

```
        IF ( ISYS (CALL_CODE, AC0, AC1, AC2) .NE. 0 ) THEN
C               PLACE ERROR HANDLING ROUTINE HERE
        ENDIF
```

NOTE: In a few cases, the "system calls" that the *AOS/VS Programmer's Manual* documents are actually calls to the User Runtime Library (URT). The ISYS function cannot work in these cases. ?TRCON is an example; to obtain a complete list, give the CLI command

X LFE/L=@CONSOLE T :UTIL:URT.LB

Implementing ISYS: an Initial Approach

Be sure you're familiar with the BYTEADDR and WORDADDR intrinsic functions. They can supply arguments for ISYS. The explanation of BYTEADDR and WORDADDR first appeared as the table *System Intrinsic Functions* in file F77_DOCUMENTATION that accompanied the Release Notice for Revision 1.30 of AOS/VS FORTRAN 77. If the explanation of BYTEADDR and WORDADDR isn't in your *FORTRAN 77 Reference Manual*, then find it in your current file

F77_DOCUMENTATION

Let's look at an example of the application of the ISYS function. Suppose our username on an AOS/VS system is TOM and we want our F77 program to change the Access Control List (ACL) of a file NEW_STUDENTS from

TOM,OWARE

to

TOM,OWARE JERRY,RE

We begin by reading the explanation of the ?SACL (set a new ACL) system call in the *AOS/VS Programmer's Manual* to learn that we must construct the new ACL as a special text string. From there, we go to the appendixes to obtain the following information from the listings of PARU.32.SR and SYSID.32.SR. We should inspect these files in our system (usually in :UTIL) to get the latest information.

Symbol	**Decimal Value**	**Meaning**
?FACO	16	Owner Access
?FACW	8	Write Access
?FACA	4	Append Access
?FACR	2	Read Access
?FACE	1	Execute Access
?.SACL	76	?SACL System Call (114K = 76)

The decimal equivalent of ACL "OWARE" is 16+8+4+2+1 = 31 and the decimal equivalent of ACL "RE" is 2+1 = 3. The respective octal equivalents are 37K and 3K.

The new ACL as an assembly language text string is

'TOM<0><?FACO+?FACW+?FACA+?FACR+?FACE> →
→ JERRY<0><?FACR+?FACE><0>'

We know, from our previous table and arithmetic, that the respective values of

<?FACO+?FACW+?FACA+?FACR+?FACE> and <?FACR+?FACE>

are 37K and 3K. Now, we can easily create the string to which AC1 must contain a byte pointer. The string is

'TOM<0><37>JERRY<0><3><0>'

Sample Program

The F77 statements resulting from our exploration of ?SACL appear in program TEST_SACL.

```
PROGRAM TEST_SACL
INTEGER*4 ISYS
INTEGER*4 BPTR_AC0, BPTR_AC1 ! BYTE POINTERS TO AC0, AC1
BPTR_AC0 = BYTEADDR('NEW_STUDENTS<0>')
BPTR_AC1 = BYTEADDR('TOM<0><37>JERRY<0><3><0>')
IER = ISYS (114K, BPTR_AC0, BPTR_AC1, IAC2) ! DO IT!
PRINT *, 'RESULT CODE FROM ISYS TO ?SACL IS ', IER
STOP
END
```

NOTE: We appended a null to 'NEW_STUDENTS' because ?SACL requires a null delimiter for a string whose byte pointer is in AC0. The second string has a trailing null because of this system call's requirement for AC1, and thus we don't add another one.

This program does the same thing as the CLI command

ACL NEW_STUDENTS TOM,OWARE JERRY,RE

Program Testing

We may test this program after we have compiled and linked it. Again, our username is TOM and the program name is TEST_SACL. The following console dialog shows the results of the test.

) DELETE/2=IGNORE NEW_STUDENTS ↵
) CREATE NEW_STUDENTS ↵
) ACL/V NEW_STUDENTS ↵
NEW_STUDENTS TOM,OWARE
) X TEST_SACL ↵
RESULT CODE FROM ISYS TO ?SACL IS 0
STOP
) ACL/V NEW_STUDENTS ↵
NEW_STUDENTS TOM,OWARE JERRY,RE
)

Summary

The sample program TEST_SACL shows how we can bring together the

- Documentation of operating system calls.
- Operating system's definition files (SYSID.32.SR and PARU.32.SR).
- BYTEADDR and WORDADDR intrinsic functions.
- ISYS external function.

to create a FORTRAN 77 program that hooks into AOS/VS via system calls at runtime.

However, this nonparametric method has its drawbacks! Program TEST_SACL is *hard-wired.* That is, it contains the current numerical values of symbols such as ?FACO. These values can change with future revisions of the operating system, and the unchanged program (with its constant values such as 37K = <37>) might then give incorrect results. Furthermore, there is no guarantee that symbols such as ?FACO will always have the same value in the AOS/VS and AOS parameter files (PARU.32.SR and PARU.SR, respectively).

How can we overcome the limitations of hard-wiring the values of system parameters in our F77 programs? For the answer, read the next section.

Implementing ISYS: a Final Approach

Data General has developed a program (F77BUILD_SYM) that builds a symbol file (QSYM.F77.IN) from your system's PARU.32 and SYSID.32 files. The command to execute the program is

X F77BUILD_SYM *[filename]*

where *filename* is the name of an optional file whose contents are symbols from the PARU.32 and SYSID.32 files. Then, your program can %INCLUDE QSYM.F77.IN and access operating system values as symbols instead of as hard-wired constants.

Files Related to Program F77BUILD_SYM

Symbol file QSYM.F77.IN contains FORTRAN 77 PARAMETER statements and values for, by default, each symbol defined in the parameter and system call definition files. For example, the statements

```
.DUSR    ?FAOB = 11.              ; OWNER ACCESS
.DUSR    ?FACO = 1B(?FAOB)        ; OWNER ACCESS
```

are in PARU.32.SR. Program F77BUILD_SYM by default transforms the second statement from its equivalent in listing file PARU.32.LS into

```
INTEGER*2 ISYS_FACO
PARAMETER (ISYS_FACO = 16)       ! ?FACO  = 20K
```

in QSYM.F77.IN. You can place the statement

```
%INCLUDE "QSYM.F77.IN"
```

in your F77 source program, and then work with symbols such as ISYS_FACO instead of with hard-wired constants such as 16 or 20K.

NOTE: The words "by default" appear twice in the previous paragraph. If, when executing F77BUILD_SYM, the CLI command does not include a filename, then the default case occurs and F77BUILD_SYM transforms *all* PARU.32 and SYSID.32 .DUSR symbols into INTEGER and PARAMETER statements in QSYM.F77.IN. If this CLI command includes a filename, then F77BUILD_SYM transforms only *specific* PARU.32 and SYSID.32 .DUSR symbols.

Figure 3-1 expands this explanation of program F77BUILD_SYM and its input files. The figure also contains a partial listing of a program (SHOW_SYMBOLS) that uses the system symbol ?FACO.

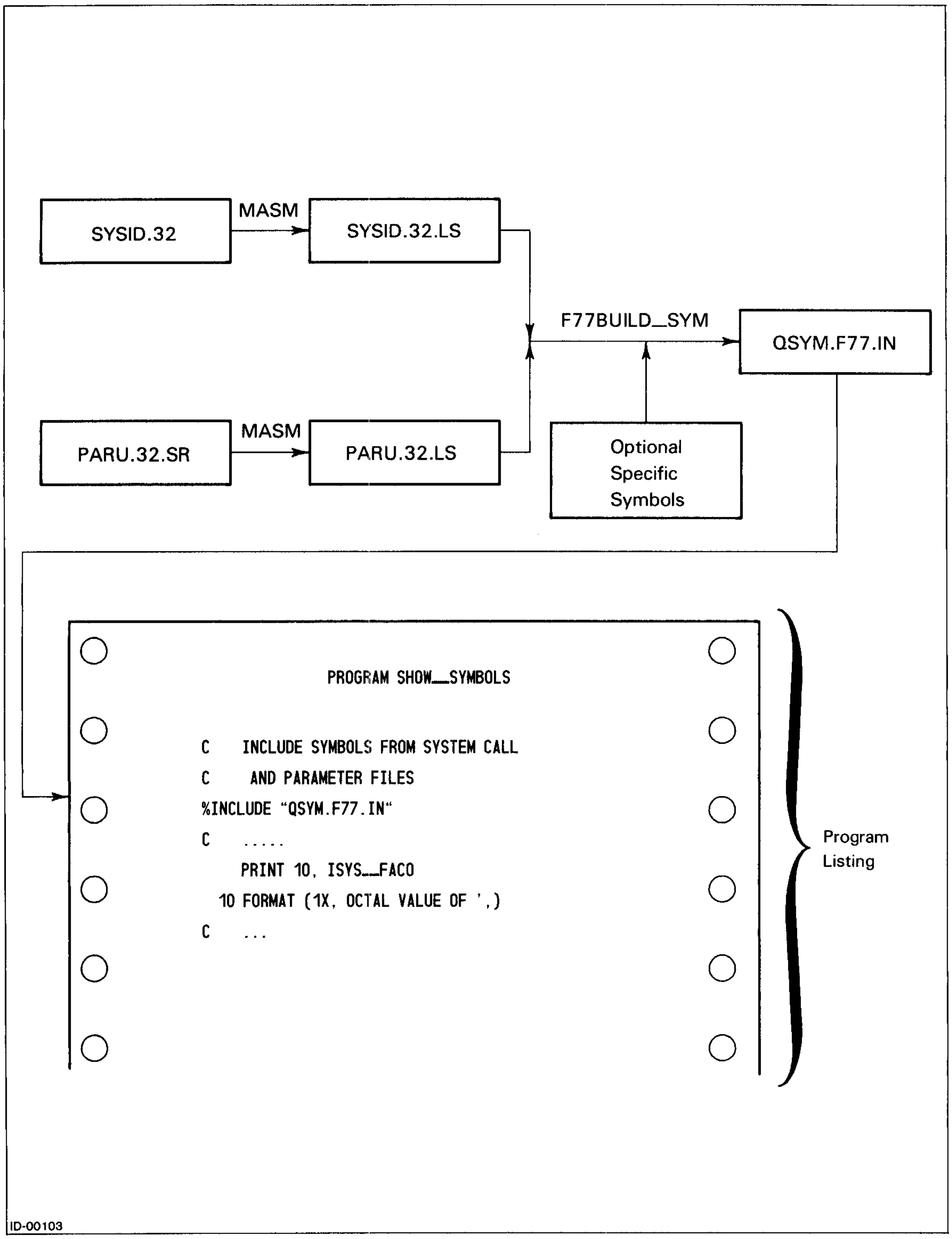


Figure 3-1. The Construction and Use of Parameter File QSYM.F77.IN

Symbol Construction Rules

F77BUILD_SYM follows these rules in sequence as it converts each PARU.32 and SYSID.32 .DUSR statement to a pair of INTEGER/PARAMETER statements in QSYM.F77.IN.

1. If the .DUSR statement defines a symbol of the form ?.<root>, then construct a symbol of the form ISYS_<root>.
   Example: ?.RETURN → ISYS_RETURN
2. If the .DUSR statement defines a symbol of the form ?<root>, then construct a symbol of the form ISYS_<root>.
   Example: ?RTDS → ISYS_RTDS
3. If the .DUSR statement defines a symbol of the form <root>, then construct a symbol of the form ISYS_<root>.
   Example: ERFTL → ISYS_ERFTL
4. If, after the ISYS_<root> symbol is formed according to one of these previous rules, <root> contains any periods, then change them to underscores.
   Example: ISYS_SYM.BOL → ISYS_SYM_BOL

Sometimes F77BUILD_SYM creates ISYS_<root> slightly differently from what you expect. For example, "?TRUNCATE" in SYSID.SR results in "ISYS_TRC" in QSYM.F77.IN. F77BUILD_SYM places ISYS_<root> symbols in QSYM.F77.IN in the same order as it reads SYSID.32.LS — sequentially.

Once it derives the ISYS_<root> symbol, F77BUILD_SYM constructs either an

```
INTEGER*4 ISYS_<root>
```

or else an

```
INTEGER*2 ISYS_<root>
```

statement. It follows three rules to select INTEGER*4 or INTEGER*2:

- If the symbol comes from SYSID.32, then the data type is INTEGER*4.
- If the symbol comes from PARU.32 and fits into 16 bits, then the data type is INTEGER*2.
- If the symbol comes from PARU.32 and requires 32 bits, then the data type is INTEGER*4.

NOTE: We explain the optional input file to F77BUILD_SYM (labeled "Optional Specific Symbols" in Figure 3-1) later in this chapter in the "Reducing QSYM.F77.IN" section. This is the same file whose name appears in a CLI command of the form

X F77BUILD_SYM *[filename]*

Operating Instructions for F77BUILD_SYM

Be sure you have access to SYSID.32.SR, PARU.32.SR, and F77BUILD_SYM.PR. The first two are usually in :UTIL and the third comes with the FORTRAN 77 software. Ask your system manager for their location.

The primary output file is QSYM.F77.IN. Most likely, you'll want to make it available to all F77 programmers on your system. You can do this by constructing it in :UTIL or in a directory devoted to F77 and accessible to all F77 programmers. Or, you may create QSYM.F77.IN in any directory, and then move it to a publicly available directory (after setting its ACL).

The CLI commands to execute F77BUILD_SYM and create QSYM.F77.IN are:

```
DELETE/2=IGNORE SYSID.32.LS
DELETE/2=IGNORE PARU.32.LS
X MASM/NOPS/L=SYSID.32.LS SYSID.32.SR
X MASM/NOPS/L=PARU.32.LS PARU.32.SR
DELETE/2=IGNORE QSYM.F77.IN
X F77BUILD_SYM
```

You should now place the statement

```
%INCLUDE "QSYM.F77.IN"
```

in an AOS/VS F77 program that references function subprogram ISYS. Then, all the .DUSR symbols and their values in files SYSID.32.SR and PARU.32.SR are available to the program.

Reducing QSYM.F77.IN

File QSYM.F77.IN, although comprehensive and usable by any F77 program that needs to interface with the operating system, is quite large. The following shows the approximate number of symbols and statements in various files.

	SYSID Symbols	PARU Symbols	QSYM.F77.IN Statements
AOS/VS	390	1700	4180
AOS	220	1610	3660

You can shorten the length of your programs' listing files considerably by including the statements

```
%LIST (OFF)
%INCLUDE "QSYM.F77.IN"
%LIST (ON)
```

Even so, this inclusion increases compilation time and usage of symbol table space during your program's compilation. Your program probably needs only a small fraction of these 2000+ symbols.

One way to reduce the size of QSYM.F77.IN is to select only the SYSID and PARU symbols that you need in your F77 programs. Place the selected symbols in a file, and then give the file's name to F77BUILD_SYM.PR. This file appears in Figure 3-1 with the label "Optional Specific Symbols."

Example

Recall program TEST_SACL that contains hard-wired PARU.32 and SYSID.32 values. We now work strictly with *symbols* instead of their *values* as we create program NEW_TEST_SACL. It performs the same function of setting the ACL of file NEW_STUDENTS to TOM,OWARE JERRY,RE.

The following CLI dialog creates a new file QSYM.F77.IN with only the six symbols necessary for ?SACL. We assume that SYSID.32.LS and PARU.32.LS remain from a prior assembly of SYSID.32.SR and of PARU.32.SR. This assembly must have occurred according to the description in the "Operating Instructions for F77BUILD_SYM" section.

```
) DELETE/2=IGNORE SACL_SYMBOLS QSYM.F77.IN SACL_SYMBOLS.F77.IN ⏎
) CREATE/I SACL_SYMBOLS ⏎
))?FACO ⏎
))?FACW ⏎
))?FACA ⏎
))?FACR ⏎
))?FACE ⏎
))?.SACL ⏎
))) ⏎
) X F77BUILD_SYM SACL_SYMBOLS ⏎
) RENAME QSYM.F77.IN SACL_SYMBOLS.F77.IN ⏎
)
```

We renamed QSYM.F77.IN to more accurately summarize its limited contents.

NOTE: In the "Operating Instructions for F77BUILD_SYM" section, we gave the CLI command

```
X F77BUILD_SYM
```

for program F77BUILD_SYM. This command results in F77BUILD_SYM's not reading the optional file (shown in Figure 3-1) and in a large output file QSYM.F77.IN.

Here, we give the following CLI command instead.

```
X F77BUILD_SYM SACL_SYMBOLS
```

This command results in F77BUILD_SYM's reading of file SACL_SYMBOLS and in a small output file QSYM.F77.IN.

Now, let's look at part of the listing (.LS) file from the compilation of program NEW_TEST_SACL.F77. See Figure 3-2.

```
Source file: NEW_TEST_SACL.F77
Compiled on 15-Jun-82 at 14:14:06 by AOS/VS F77 Rev 02.00.00.00
Options: F77/L=NEW_TEST_SACL.LS

C       AOS/VS PROGRAM NEW_TEST_SACL
        INTEGER*4 ISYS, VALUE_OWARE, VALUE_RE
        CHARACTER*20 AC1 ! FOR THE NEW ACL
        INTEGER*4 BPTR_AC0, BPTR_AC1 ! BYTE POINTERS TO AC0, AC1
%INCLUDE "SACL_SYMBOLS.F77.IN"
****  F77 INCLUDE file for system parameters ****

****   INTEGER*4 parameters for SYSID  ****


        INTEGER*4  ISYS_SACL
        PARAMETER (ISYS_SACL = 76)      ! ?.SACL = 114K

****    Parameters for PARU   ****


        INTEGER*2  ISYS_FACA
        PARAMETER (ISYS_FACA = 4)       ! ?FACA = 4K

        INTEGER*2  ISYS_FACE
        PARAMETER (ISYS_FACE = 1)       ! ?FACE = 1K

        INTEGER*2  ISYS_FACO
        PARAMETER (ISYS_FACO = 16)      ! ?FACO = 20K

        INTEGER*2  ISYS_FACR
        PARAMETER (ISYS_FACR = 2)       ! ?FACR = 2K

        INTEGER*2  ISYS_FACW
        PARAMETER (ISYS_FACW = 8)       ! ?FACW = 10K


****  END of F77 INCLUDE file for system parameters ****
C       CONSTRUCT THE VALUE OF ?FACO+?FACW+?FACA+?FACR+?FACE .
        VALUE_OWARE = ISYS_FACO + ISYS_FACW + ISYS_FACA +
     1                ISYS_FACR + ISYS_FACE
C       CONSTRUCT THE VALUE OF ?FACR+?FACE .
        VALUE_RE    = ISYS_FACR + ISYS_FACE
C       CONSTRUCT THE NEW ACL IN CHARACTER VARIABLE AC1.  NOTE THE
C          USE OF THE CHAR INTRINSIC FUNCTION TO CONVERT AN INTEGER
C          NUMBER TO ITS ASCII CHARACTER EQUIVALENT.  FOR EXAMPLE,
C          VALUE_RE IS CURRENTLY (AOS/VS REVISION 1.50) 3 AND
C          CHAR(VALUE_RE) IS '<3>'.
        AC1 = 'TOM<0>' // CHAR(VALUE_OWARE) // 'JERRY<0>' //
     1                    CHAR(VALUE_RE)    // '<0>'
        BPTR_AC0 = BYTEADDR("NEW_STUDENTS<0>")
        BPTR_AC1 = BYTEADDR(AC1)
        IER = ISYS (ISYS_SACL, BPTR_AC0, BPTR_AC1, IAC2) ! DO IT!
        PRINT *, 'RESULT CODE FROM ISYS TO ?SACL IS ', IER
        END
```

DG-25217

Figure 3-2. Program NEW_TEST_SACL

Error Messages

The following error messages from F77BUILD_SYM may appear on @OUTPUT:

- *Can't open <filename>*

  This refers to one of the input files. Either you haven't created the necessary .LS files or the optional special symbols file, or for some reason the file isn't accessible.

- *Unreferenced symbol: <symbol>*

  You've supplied an optional special symbols file. However, <symbol> in that file wasn't found in either .LS file. BIG_MAC is an example of an unreferenced symbol.

- *Invalid symbol: <symbol>*

  You've supplied an optional special symbols file. However, <symbol> in that file does not have one of the following formats:

  - ?.<name>
  - .<name>
  - <name>

  where <name> begins with a letter. $LPT is an example of an invalid symbol.

Updating your Operating System

We suggest that you do the following for each revision or update of your operating system:

- Reassemble the new SYSID.32 and PARU.32 .SR files.
- Rerun F77BUILD_SYM.
- Recompile and relink all programs that %INCLUDE statements from QSYM.F77.IN.

It isn't always necessary to do these things, but doing them may prevent some strange F77 program behavior because of changes to the operating system.

ISYS and Sample Program LIST_DIRECTORY

Program NEW_TEST_SACL is an elaborate way of invoking the ?SACL system call. It is, of course, easier to give the CLI command ACL to invoke ?SACL. However, sometimes we want to invoke a system call that has no direct counterpart as a CLI command. ?GNFN (Get the Next FileName) is an example.

Program Unit Listings

Program LIST_DIRECTORY is an instance of a program that uses ISYS to invoke ?GNFN. At runtime, LIST_DIRECTORY accepts a directory name and a template. It attempts to list the filenames of all the files that are in the directory and that match the template. LIST_DIRECTORY appears in Figure 3-3. Figures 3-4 and 3-5 contain listings of its respective subroutine subprograms ADD_NULL and CHECK.

NOTE: We have executed program F77BUILD_SYM to create an all-inclusive AOS/VS symbol file QSYM.F77.IN. Both LIST_DIRECTORY and CHECK %INCLUDE this file, even though the statement

```
%INCLUDE 'QSYM.F77.IN'
```

does not appear in Figures 3-3 and 3-5. This statement is, of course, part of source program files LIST_DIRECTORY.F77 (at line 31) and CHECK.F77 (at line 10).

```
Source file: LIST_DIRECTORY.F77
Compiled on 14-Jun-82 at 14:15:31 by AOS/VS F77 Rev 02.00.00.00
Options: F77/L=LIST_DIRECTORY.LS

                PROGRAM LIST_DIRECTORY

                INTEGER*4 AC0,AC1,AC2         ! Accumulators
                INTEGER*4 ISYS                ! System interface function subprogram
                INTEGER*4 RESULT_CODE         ! Result of calling ISYS

                CHARACTER*132 FILENAME        ! Received by GNFN
                CHARACTER*132 DIRECTORY       ! Supplied to OPEN
                CHARACTER*132 TEMPLATE        ! Supplied to GNFN

                INTEGER*2 OPEN_PACKET(0:23) / 24*0 /  ! Parameter packet for ?OPEN
                   INTEGER*2 CHANNEL                  ! Offset ?ICH
                   INTEGER*2 ISTI                     ! Offset ?ISTI
                   INTEGER*2 ISTO                     ! Offset ?ISTO
                   INTEGER*2 IMRS                     ! Offset ?IMRS
                   INTEGER*4 IBAD                     ! Offset ?IBAD/?IBAL
                   INTEGER*4 IFNP                     ! Offset ?IFNP/?IFNL
                   INTEGER*4 IDEL                     ! Offset ?IDEL/?IDLL

                   EQUIVALENCE (OPEN_PACKET(0), CHANNEL)
                   EQUIVALENCE (OPEN_PACKET(1), ISTI)
                   EQUIVALENCE (OPEN_PACKET(2), ISTO)
                   EQUIVALENCE (OPEN_PACKET(3), IMRS)
                   EQUIVALENCE (OPEN_PACKET(4), IBAD)
                   EQUIVALENCE (OPEN_PACKET(12),IFNP)
                   EQUIVALENCE (OPEN_PACKET(14),IDEL)

                INTEGER*4 GNFN_PACKET(0:2)            ! Parameter Packet for ?GNFN

        %LIST(OFF)
        %LIST(ON)

          100   PRINT *,"Directory? "
                READ (*,10,END=1000) DIRECTORY    ! Accept a directory name.
           10   FORMAT(A)
        C       @INPUT end-of-file is CTRL-D.

                CALL ADD_NULL(DIRECTORY)         ! Change the first (if any)
        C                                            space ('<040>') in the
        C                                            directory name to a null.

        C       Prepare the parameter packet for ?OPEN.
                ISTI = 0                         ! Default ?OPEN options
                ISTO = 0                         ! Default file type
                IMRS = -1                        ! Default block size
                IBAD = -1                        ! Default byte pointer to buffer
                IFNP = BYTEADDR(DIRECTORY)       ! Byte pointer to directory name
                IDEL = -1                        ! Default delimiters
```

DG-25218

Figure 3-3. Program LIST_DIRECTORY (continues)

```

          AC2 = WORDADDR(OPEN_PACKET)

C         Execute the ?OPEN system call to the accepted directory.

          RESULT_CODE = ISYS(ISYS_OPEN, AC0, AC1, AC2)

C         If ?OPEN has executed successfully, then report nothing and
C            continue.  Otherwise, report the error on @OUTPUT and STOP
C            the program.
          CALL CHECK(RESULT_CODE,"On OPEN of directory " // DIRECTORY)


          PRINT *,"Template?  "
          READ (*,20,END=1000) TEMPLATE        ! Typical response is  + .
     20   FORMAT(A)

          CALL ADD_NULL(TEMPLATE)              ! Change the first (if any)
C                                                   space in TEMPLATE to
C                                                   a null.

          GNFN_PACKET(0) = 0                   ! Offset ?NFKY/?NFRS
          GNFN_PACKET(1) = BYTEADDR(FILENAME)  ! Offset ?NFNM/?NFNL
          GNFN_PACKET(2) = BYTEADDR(TEMPLATE)  ! Offset ?NFTP/?NFTL
          AC1 = CHANNEL                        ! Channel number from ?OPEN
          AC2 = WORDADDR(GNFN_PACKET)

C         Call ?GNFN to get the next filename from the current directory.
    200   RESULT_CODE = ISYS(ISYS_GNFN, AC0, AC1, AC2)

          IF ( RESULT_CODE .EQ. 0 ) THEN  ! Ignore the first (if any) null
C                                            in FILENAME and then print
C                                            the filename.
                  NULL_POS = INDEX(FILENAME,"<NUL>")
                          IF ( NULL_POS .EQ. 0 ) NULL_POS = LEN(FILENAME)-1
                  PRINT *, FILENAME(1:NULL_POS-1)
                  GOTO 200                ! Get the next filename.

          ELSE IF ( AC0 .EQ. ISYS_EREOF ) THEN
                  PRINT *
                  PRINT *,"-- End of Directory --"
                  PRINT *
                  AC2 = WORDADDR(OPEN_PACKET)

C                 Close the current directory and move to its superior.
                  RESULT_CODE = ISYS(ISYS_CLOSE, AC0, AC1, AC2)
                  CALL CHECK(RESULT_CODE, 'While closing the directory')

                  GOTO 100                ! Get the next directory name
```

DG-25218

Figure 3-3. Program LIST_DIRECTORY (continued)

```
               ELSE    ! A ?GNFN error, different from end-of-file, has occurred.
                       CALL CHECK(ACO, 'During a ?GNFN Call')

               ENDIF

        1000   PRINT *
               PRINT *,'<7>*** End of program LIST_DIRECTORY ***<NL>'
               END
```

DG-25218

Figure 3-3. Program LIST_DIRECTORY (concluded)

```
Source file: ADD_NULL.F77
Compiled on 14-Jun-82 at 14:17:17 by AOS/VS F77 Rev 02.00.00.00
Options: F77/L=ADD_NULL.LS

           SUBROUTINE ADD_NULL(TEXT)
     C
     C     Change the first space in TEXT to a null.
     C

           CHARACTER*(*) TEXT
           INTEGER SPACE_POS

           SPACE_POS = INDEX(TEXT,'<040>')
           IF ( SPACE_POS .NE. 0 ) TEXT(SPACE_POS:SPACE_POS) = '<NUL>'
           RETURN
           END
```

DG-25219

Figure 3-4. Subroutine Subprogram ADD_NULL

```
Source file: CHECK.F77
Compiled on 14-Jun-82 at 14:17:29 by AOS/VS F77 Rev 02.00.00.00
Options: F77/L=CHECK.LS

              SUBROUTINE CHECK(ECODE,TEXT)

              INTEGER*4 ECODE       ! Error code returned from ISYS
              CHARACTER*(*) TEXT    ! Error text from main program to
      C                                  accompany ECODE

              INTEGER*4 AC2

      %LIST(OFF)
      %LIST(ON)

              IF ( ECODE.EQ.0 ) RETURN  ! ISYS executed without an error.

      C       ISYS executed with an error, so report it.
              AC2 = ISYS_RFCF + ISYS_RFEC + ISYS_RFER
              AC2 = AC2 + MIN(LEN(TEXT),255)

      C       Execute ?RETURN and report the error from ISYS.
              IER = ISYS(ISYS_RETURN, ECODE, BYTEADDR(TEXT), AC2)
              STOP '- Impossible-to-occur error occurred during ?RETURN'
              END
```

DG-25220

Figure 3-5. Subroutine Subprogram CHECK

Sample Execution of Program LIST_DIRECTORY

Figure 3-6 shows the dialog that occurred during an execution of LIST_DIRECTORY. In the working directory, subdirectory FOO_DIR existed with at least one file; nondirectory file FOO also existed. Note the resulting error message when ?GNFN attempted to read file FOO.

```
) XEQ LIST_DIRECTORY ↵
Directory?  FOO_DIR ↵
Template?   + ↵
FOO1_FILE
FOO2_FILE
FOO3_FILE
--End of Directory--
Directory?  FOO ↵
Template?   + ↵
*ERROR*
NOT A DIRECTORY
During a ?GNFN Call
ERROR: FROM PROGRAM
X,LIST_DIRECTORY
)
```

DG-25167

Figure 3-6. @CONSOLE Dialog During Execution of LIST_DIRECTORY

ISYS and Subroutine CLI

You may be one of many programmers using the SED text editor to create source files. If so, you're probably familiar with the convenient DO command that lets you create a short-lived CLI process to execute one or more CLI commands. One such application of the DO command is

```
DO DELETE/V/2=IGNORE LINES_3_15 ; DUPLICATE LINES 3 TO 15 ONTO LINES_3_15
```

A natural question to ask now, regardless of whether or not you're familiar with SED, is: "If ISYS lets me execute any AOS/VS system call, thus including ?PROC, can I create a subroutine that does the following:

- Receives a string of CLI commands.
- Creates a son process (via ?PROC) that executes :CLI.PR.
- Gives the string to :CLI.PR for processing.
- Reports on the success or failure of the process' creation."

Happily, the answer is "yes." Continue reading for details about the subroutine.

Program Unit Listings

Figure 3-7 contains a listing of a subroutine subprogram, CLI, that performs these four consecutive functions. Figure 3-8 contains a listing of a program, TEST_CLI, to test the subroutine.

NOTE: We have executed program F77BUILD_SYM to create an all-inclusive AOS/VS symbol file QSYM.F77.IN. Subprogram CLI %INCLUDEs this file, even though the statement

```
%INCLUDE 'QSYM.F77.IN'
```

does not appear in Figure 3-7. This statement is, of course, part of source program file CLI.F77 (at line 31).

```
Source file: CLI.F77
Compiled on 14-Jun-82 at 10:39:02 by AOS/VS F77 Rev 02.00.00.00
Options: F77/L=CLI.LS

              SUBROUTINE CLI(TEXT, RESULT_CODE)

C             This subroutine receives a string of CLI commands from the main
C                program.  The subroutine then creates a CLI son process and
C                gives it the string of commands to execute.

              INTEGER*4 ADDRESS_OF_PROGRAM_NAME    ! Program name of the son
C                                                        process is CLI.PR.
              INTEGER*4 ADDRESS_OF_STRING          ! The string is the string
C                                                        of CLI commands.
              INTEGER*4 ADDRESS_OF_MESSAGE_HEADER  ! Packet for ?ISEND header
              INTEGER*4 AC0, AC1, AC2              ! Accumulators
              INTEGER*4 ISYS                       ! System interface function
              INTEGER*4 RESULT_CODE                !    Number it returns to
C                                                       this subroutine and
C                                                       then to the main program.

              INTEGER*2 PROC_PACKET(0:31) / 32*-1/ ! Packet for ?PROC call
                 EQUIVALENCE ( ADDRESS_OF_PROGRAM_NAME, PROC_PACKET(2) )
                 EQUIVALENCE ( ADDRESS_OF_MESSAGE_HEADER, PROC_PACKET(4) )

              INTEGER*2 ISEND_HEADER(0:7) / 8*0 /  ! Packet for ?ISEND header
C                                                       for interprocess
C                                                       communication (IPC).
                 EQUIVALENCE ( ADDRESS_OF_STRING, ISEND_HEADER(6) )

              CHARACTER*(*) TEXT                   ! String of CLI commands
              CHARACTER*(256) TEMPORARY_TEXT

      %LIST(OFF)
      %LIST(ON)

              TEMPORARY_TEXT = TEXT                ! Move the CLI commands to
C                                                       a fixed-length buffer.

C             Prepare ?ISEND header packet.
              ISEND_HEADER(5) = 128                ! Maximum length of the IPC
C                                                       message in words
              ADDRESS_OF_STRING = WORDADDR(TEMPORARY_TEXT)

C             Prepare ?PROC packet.
              PROC_PACKET(0) = ISYS_PFEX   ! Set ?PFEX bit so that CLI.PR will
C                                             execute with its father blocked.
              ADDRESS_OF_PROGRAM_NAME = BYTEADDR(':CLI.PR<0>')
              ADDRESS_OF_MESSAGE_HEADER = WORDADDR(ISEND_HEADER)
              AC2 = WORDADDR(PROC_PACKET)
```

DG-25221

Figure 3-7. Subroutine Subprogram CLI (continues)

```
C       Do it!
        RESULT_CODE = ISYS(ISYS_PROC, AC0, AC1, AC2)
C       The main program receives the value of RESULT_CODE.

        RETURN
        END
```

DG-25221

Figure 3-7. Subroutine Subprogram CLI (concluded)

```
Source file: TEST_CLI.F77
Compiled on 14-Jun-82 at 10:44:52 by AOS/VS F77 Rev 02.00.00.00
Options: F77/L=TEST_CLI.LS

              PROGRAM TEST_CLI         ! to test subroutine CLI
              CHARACTER*80 CLI_STRING  ! string of CLI commands
              INTEGER*4 IER            ! error variable returned from
    C                                       subroutine CLI and from its
    C                                       reference to function ISYS

              WRITE (6, 20)
         20   FORMAT (1H0, 'GIVE ME A CLI COMMAND STRING: ', $)
              READ (5, 30, END=60) CLI_STRING
         30   FORMAT (A)
    C         @INPUT end-of-file is CTRL-D.

              WRITE (6, 40)
         40   FORMAT (1H , 'HERE WE GO ...', /, /)
              CALL CLI (CLI_STRING, IER)
              WRITE (6, 50)
         50   FORMAT (1H , 'JUST RETURNED FROM SUBROUTINE CLI')
              IF ( IER .NE. 0 ) THEN
                   PRINT *
                   PRINT *, 'ERROR ', IER, ' OCCURRED DURING ',
           1                 'REFERENCE TO ISYS'
                   PRINT *, '   WHEN SUBROUTINE CLI EXECUTED.'
              ENDIF

         60   WRITE (6, 70)
         70   FORMAT (1H0, '*** END OF PROGRAM ***', /)

              STOP
              END
```

DG-25222

Figure 3-8. Program TEST_CLI

Sample Execution of Program TEST_CLI

Figure 3-9 shows the dialog that occurred during an execution of TEST_CLI. In the working directory, subdirectory FOO_DIR existed with at least one file; nondirectory file FOO also existed. Note the resulting error message

ERROR: NON-DIRECTORY ARGUMENT IN PATHNAME, FILE FOO
DIR,FOO

when user Marll tried to make FOO the working directory. The son process CLI.PR reported this two-line error message. The ?PROC call from subroutine CLI.OB that created this son process executed without error. So, TEST_CLI received 0 in argument IER and did *not* execute its statements in lines 19-22.

```
) XEQ TEST_CLI ↵
GIVE ME A CLI COMMAND STRING:    TIME; DATE; DIRECTORY; WHO ↵
HERE WE GO ...
15:30:49
14-JUN-82
:UDD:F77:MARLL
PID:   38 F77          038          :CLI.PR
JUST RETURNED FROM SUBROUTINE CLI
*** END OF PROGRAM ***
STOP
) XEQ TEST_CLI ↵
GIVE ME A CLI COMMAND STRING:    DIR FOO; FILESTATUS + ↵
HERE WE GO ...
ERROR: NON-DIRECTORY ARGUMENT IN PATHNAME, FILE FOO
DIR,FOO
JUST RETURNED FROM SUBROUTINE CLI
*** END OF PROGRAM ***
STOP
)
```

DG-25165

Figure 3-9. @CONSOLE Dialog During Execution of TEST_CLI

A Variation of Program TEST_CLI

Program TEST_CLI accepts a CLI command string at runtime from @INPUT. You may also write programs that contain a "hard-wired" CLI command string in a CHARACTER variable. For example, let's modify lines 6 through 11, inclusive, of program TEST_CLI (in Figure 3-8) to create program TEST1_CLI. Figure 3-10 contains TEST1_CLI, and Figure 3-11 shows the results of its execution.

```
Source file: TEST1_CLI.F77
Compiled on 15-Jun-82 at 10:30:02 by AOS/VS F77 Rev 02.00.00.00
Options: F77/L=TEST1_CLI.LS

              PROGRAM TEST1_CLI          ! to test subroutine CLI
              CHARACTER*80 CLI_STRING    ! string of CLI commands
              INTEGER*4 IER              ! error variable returned from
      C                                     subroutine CLI and from its
      C                                     reference to function ISYS



              CLI_STRING = 'TIME; DATE; WHO; RUNTIME'



              WRITE (6, 40)
         40   FORMAT (1H , 'HERE WE GO ...', /, /)
              CALL CLI (CLI_STRING, IER)
              WRITE (6, 50)
         50   FORMAT (1H , 'JUST RETURNED FROM SUBROUTINE CLI')
              IF ( IER .NE. 0 ) THEN
                   PRINT *
                   PRINT *, 'ERROR ', IER, ' OCCURRED DURING ',
           1                'REFERENCE TO ISYS'
                   PRINT *, '   WHEN SUBROUTINE CLI EXECUTED.'
              ENDIF

         60   WRITE (6, 70)
         70   FORMAT (1H0, '*** END OF PROGRAM ***', /)

              STOP
              END
```

DG-25223

Figure 3-10. Program TEST1_CLI

```
) XEQ TEST1_CLI ⤶
HERE WE GO ...
10:31:22
15-JUN-82
PID:   35 F77          035          :CLI.PR
ELAPSED   0:00:01, CPU   0:00:00.046, I/O BLOCKS   0, PAGE SECS 2
JUST RETURNED FROM SUBROUTINE CLI
*** END OF PROGRAM ***
STOP
)
```

DG-25166

Figure 3-11. @CONSOLE Dialog During Execution of TEST1_CLI

You could modify program TEST1_CLI to pass a character constant to subroutine CLI by making lines 2 and 9 blank and by changing line 15 to

```
CALL CLI ('TIME; DATE; WHO; RUNTIME', IER)
```

The runtime results would be identical to those of the original TEST1_CLI (in Figure 3-10).

The ISYS Function and Multitasking

Very briefly — Don't use the ISYS function to do multitasking!

Chapter 4 documents the subroutines that your F77 programs should CALL when they issue multitasking instructions. These subroutines interact correctly with the FORTRAN 77 runtime routines and databases.

IO_CHAN Function

This external function returns the channel number that the operating system assigned to the F77 I/O unit number supplied as the function's argument. If this unit number is invalid IO_CHAN returns a value of -1.

Structure

The structure of function subprogram IO_CHAN is

IO_CHAN(unit)

where:

IO_CHAN	is a symbol whose data type you specify via an INTEGER*4 statement.
unit	is an INTEGER*4 expression that specifies an F77 I/O unit number.

Example

```
C       AOS/VS PROGRAM DEMO_IO_CHAN
C       ...
        INTEGER*4 IO_CHAN
C       ...
        OPEN (3, FILE='TIME_CARDS', RECFM='DS',
     +          STATUS='OLD')
        IOCHAN_3 = IO_CHAN(3)
        IF ( IOCHAN_3 .EQ. -1 ) THEN
                PRINT *, 'IO_CHAN RECEIVED AN ',
     +                  'INVALID UNIT NUMBER'
        ELSE
                PRINT 10, IOCHAN_3
   10           FORMAT (1X, 'OPERATING SYSTEM CHANNEL NUMBER',
     +                 ' ASSIGNED TO UNIT 3 IS', O6, 'K')
        ENDIF
C       ...
        STOP
        END
```

Reference

The number that the IO_CHAN function returns is the ?ICH offset of a parameter packet for the ?OPEN system call. In the previous example, the F77 runtime routines prepared a parameter packet and used it to make the ?OPEN call in response to the

```
OPEN (3, ...)
```

F77 source program statement. This ?OPEN call set ?ICH; the subsequent reference to IO_CHAN(3) then retrieved the value of ?ICH.

End of Chapter

Chapter 4
Multitasking

AOS/VS supports multitasking — a useful programming technique. Just as timesharing allows several concurrent processes to exist within one computer, multitasking allows several concurrent instruction paths to exist within one process.

This chapter gradually introduces multitasking in the following sections:

- What is a Task?
- What is Multitasking? — A Nonsoftware Example
- What is Multitasking?
- Task States, Transitions, and Subroutines
- Re-entrant Code
- Multitasking Subroutines
- Sample Programs

If you're familiar with multitasking (such as implemented in Data General's FORTRAN IV or FORTRAN 5) and only want to know the details of FORTRAN 77's subroutines that "hook into" AOS/VS multitasking routines, then skip to Figure 4-7, and then to the section entitled "Multitasking Subroutines."

What is a Task?

A task is an asynchronous path of execution through a program.

Let's examine the key words in this definition:

- "Path" implies a beginning and an end. Thus, each task has an initial instruction and a final instruction during its existence.
- "Path" means the sequence of instructions that execute at runtime. An instruction may execute more than once during the task's existence. For example, such an instruction may originate from the body of a DO loop.
- "Asynchronous" means each instruction executes by itself during a specific time period. Instructions vary in the amount of time they require. For example, a binary addition requires much less time than the division of two REAL*8 numbers. And, the instructions from one program unit may execute interleaved with those from other program units. "Asynchronous" comes from three Greek words that mean "*not* in the *same time* [as something else]."

Single-task Programs

Any FORTRAN 77 program you've written according to the rules in the *FORTRAN 77 Reference Manual* is a single-task program. That is, at runtime the CPU executes exactly one flow of instructions from your program. An instruction usually has to wait only for its predecessor's completion before CPU execution. (An exception occurs when there is overlap in floating-point instruction execution.)

Single-tasking: a Nonsoftware Example

Consider the physical situation of a one-way, one-lane road that leads to a narrow and short tunnel. Assume that drivers have cooperated so their cars form one line of traffic. Thus, each driver merely has to wait until the cars ahead go through the one-lane tunnel. That is, once a car is in line, there is no competition from parallel lines of cars for the one available lane. Thus, the tunnel only handles traffic arriving from one lane. Furthermore, the vehicles go through the tunnel one at a time — not in a continuous flow. Figure 4-1 portrays this situation.

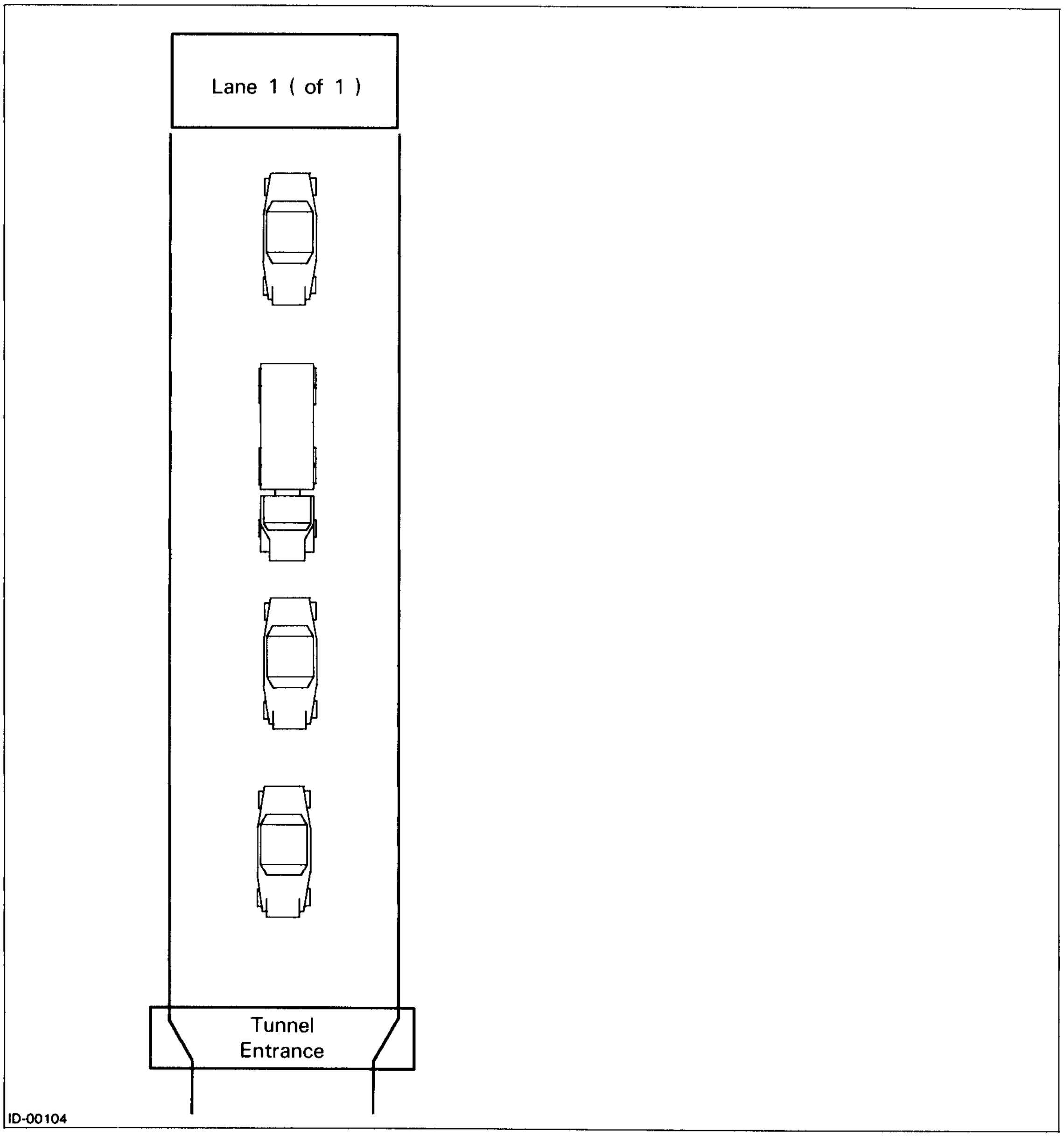


Figure 4-1. A One-Lane Tunnel with One Approach Lane (Single-Tasking)

In the figure, the third vehicle awaiting passage through the tunnel is a semitrailer truck. Note that the truck cannot pass through the tunnel as quickly as the cars. This also means that the cars behind the truck have a longer wait than the cars in front of the truck.

Below we list certain correspondences between a single-tasked program and the lane/tunnel situation in Figure 4-1:

- Each instruction executes asynchronously, awaiting the completion of its predecessor. (Each vehicle goes through the tunnel after its predecessor completes the trip.)
- The program has a beginning and an end, even though the sequence of instructions may change (depending on the data read) from one execution to the next. (In a given time period, there is an initial vehicle and a final vehicle.)
- Some instructions, particularly those resulting in commands to AOS/VS to perform an I/O operation, require much more time to complete than others. (Some vehicles, particularly loaded trucks, require much more time to go through the tunnel than others.)
- If certain instructions — particularly I/O commands — could execute without tying up the CPU, then many other instructions could execute along with the certain instructions. (If a separate and parallel truck lane existed in the tunnel, then many autos and motorcycles could pass through along with one truck.)

The last point raises an important question: is there some way to construct a program file so that *many fast instructions* may execute in the same time period that *one slow instruction* executes? In other words: adding a truck lane to the tunnel greatly increases the traffic flow; is there a parallel software construction? Happily, the answer is "yes"; it's called *multitasking.*

What is Multitasking? — a Nonsoftware Example

To lead up to the software construction, let's create a hardware system that greatly increases the number of vehicles that can go through the tunnel in a given time period. To do this, we:

- Widen the tunnel so that a car and a truck (but only one of each) can be passing through the tunnel simultaneously.
- Assume that there are four competing lanes of traffic leading into the tunnel.
- Set up a controller who regulates the gates at the end of each lane to control the overall throughput.
- Assume that many cars can go through the tunnel while one truck is passing through.

See Figure 4-2.

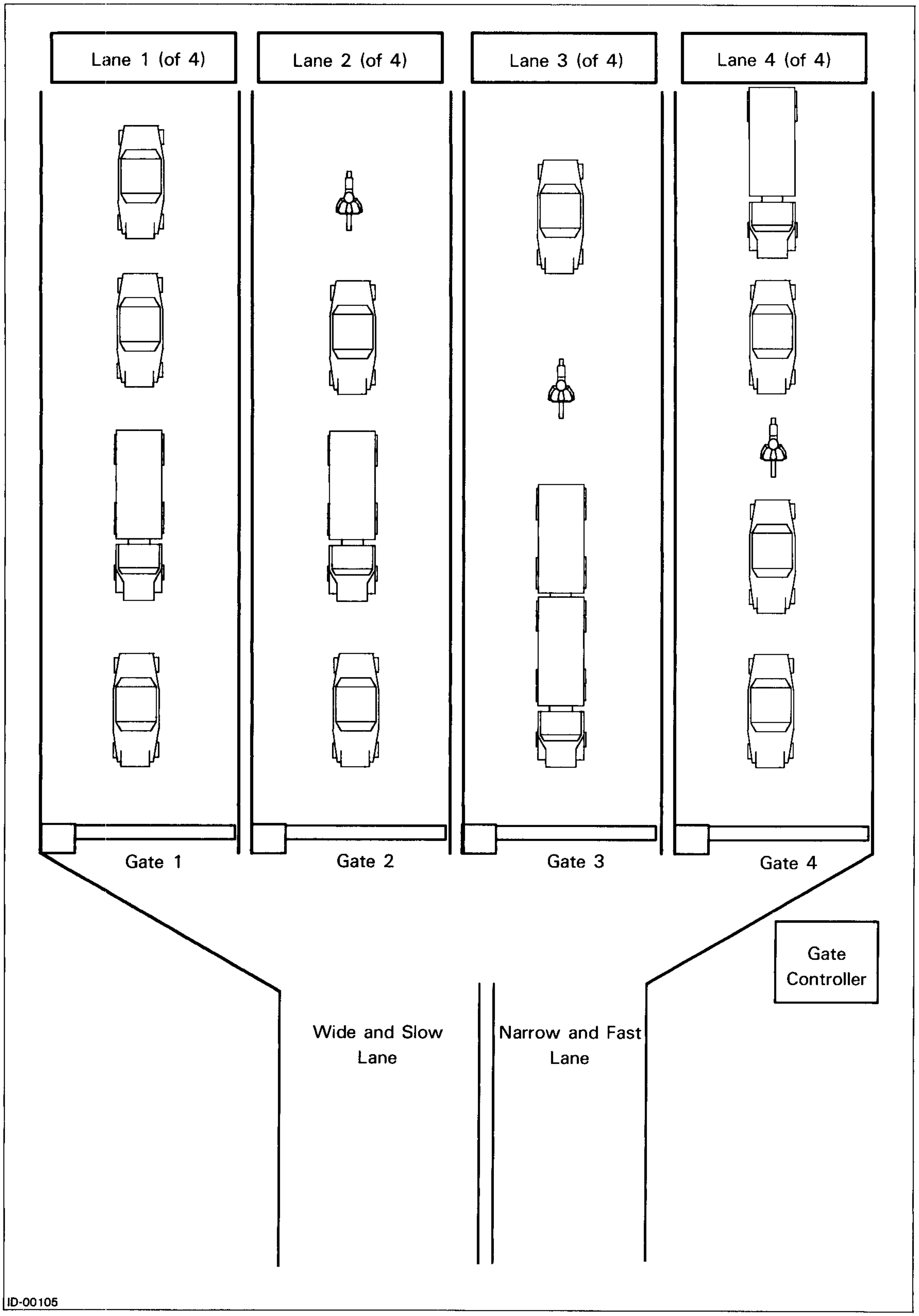


Figure 4-2. A Two-Lane Tunnel with Four Approach Lanes (Multitasking)

Note that setting up the controller to regulate the gates is most important. We assume that:

- Each traffic lane has a unique number to identify it.
- Each lane has a priority number. For example, one lane might be reserved for emergency vehicles. If the lead vehicles in two or more lanes are both ready to go, then the vehicle in the lane with the higher priority will go first.
- Each lane can communicate with any other lane and with the controller.
- Each lane can attempt to control itself and other lanes by:
  - Closing gates permanently.
  - Closing gates temporarily.
  - Changing priorities of lanes.
  - Making lanes ready if their gates are closed.
- The controller can overrule any command by any lane.

In summary, *a set of multiple tasks (lanes of vehicles) competes for limited resources (two routes through the tunnel) according to certain rules (the lanes' requests and the controller's decisions).*

These assumptions may not entirely represent real-life situations, especially in terms of communication and control amongst the lanes and the controller. However, this traffic situation and the assumptions listed above provide a convenient bridge to understanding software multitasking.

For another real-life example of a multitasking situation, consider an expert chess player who plays several games simultaneously. He concentrates on one board at a time, yet is aware of the other boards and must service them periodically.

At this point, we mostly leave behind our lane/tunnel situation and explain multitasking in terms of software.

What is Multitasking?

In software multitasking, we create a source program and subroutines, which we compile and link into a program file. At runtime the program file has several paths of instructions awaiting CPU execution, just as the tunnel has several lanes of traffic to accept. In either case a very important part is, of course, the rules for lane selection (i.e., which gate is open) and path selection (i.e., which instruction executes next). Figure 4-3 shows the structure of a program file with a main program and three tasks.

Figure 4-3 shows that multitasking consists of multiple, concurrent flows through a program, where the various flows (tasks) compete for CPU control. In multitasking, a single program deals easily and efficiently with two or more tasks at one time. Although there is only one CPU, and in reality only one instruction executes at a time, it appears as though several instructions from different tasks are executing simultaneously. This is because tasks take turns executing. For example, when one suspends execution (because of awaiting completion of an I/O instruction or some other reason), another task gains control of the CPU. All of this happens automatically within the operating system. Thus, you have no need to keep track of the various tasks and to appropriately switch control among them. F77 runtime routines and AOS/VS take care of such bookkeeping activities. As many as 32 tasks may be active simultaneously.

Even though you have no need to switch control among tasks, you *can* exercise a fine control over the tasks that the system selects for execution and the time at which it selects them. When you define a task and specify the instructions it will execute at the source program level, you also assign the task a priority number relative to other tasks. However, you can change these task priorities at runtime. This change allows you to control which tasks receive CPU control, and when. A *task scheduler*, which is part of AOS/VS, allocates CPU control to the highest priority task that is ready either to perform or to continue to perform its function.

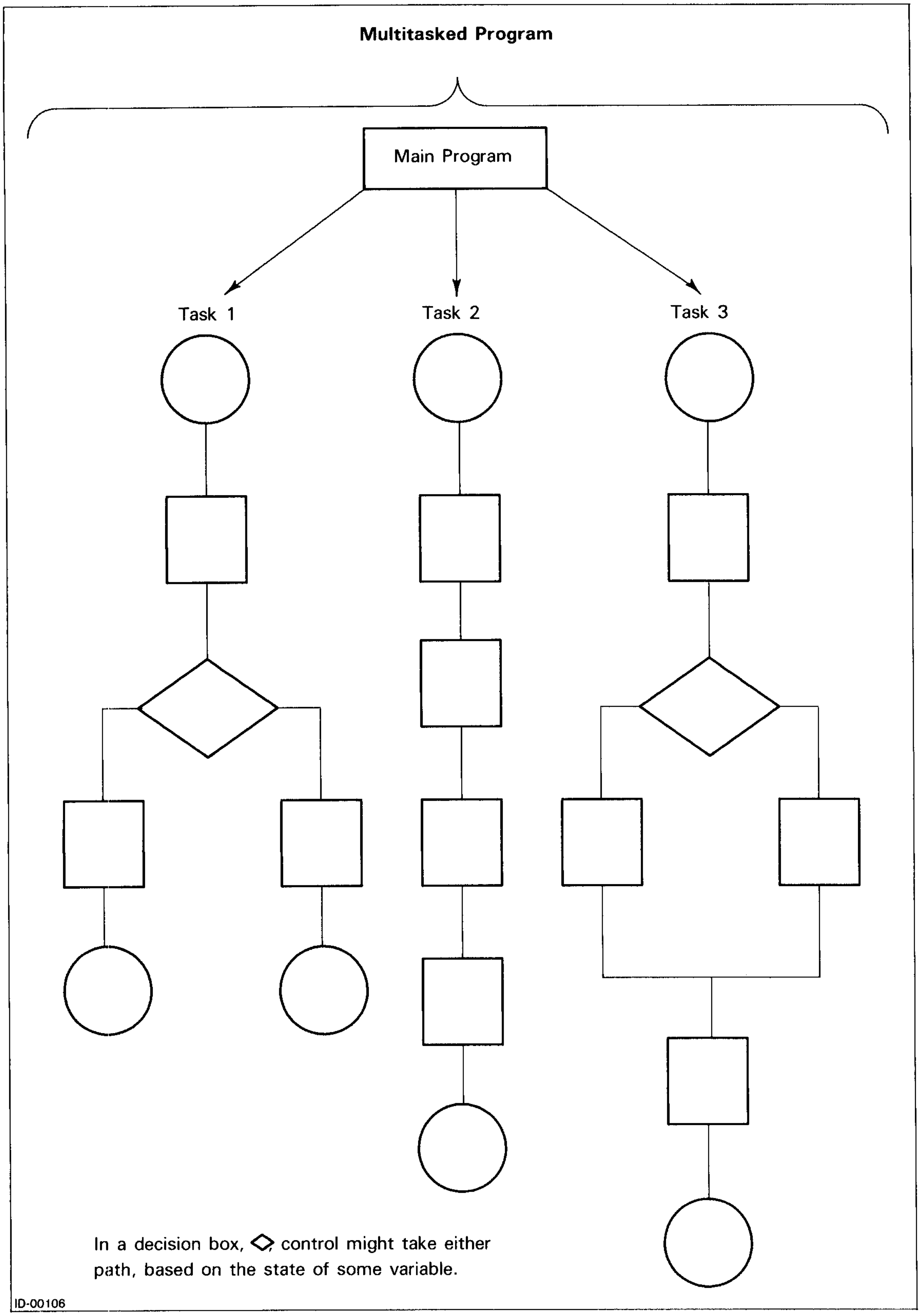


Figure 4-3. A Multitasking Program File

Although each task in a multitask environment can execute independently, a certain amount of interaction between the tasks is often required. F77's multitasking subroutines allow convenient intertask communication, providing for synchronization. For example, a task may suspend its own execution at a certain point, awaiting a signal from another task.

Remember, you do not create tasks; you, the computer, and Link create instructions in the program file. The one or more runtime execution paths through these instructions create a multitasking environment.

Multitasking Program Organization

We construct a multitasked program based on a main program unit and one or more subroutines. As an example, Figure 4-4 shows both the organization of a single-task program with two subroutines and its execution. Then, for comparison, Figure 4-5 shows both the organization of a multitask program with two tasks and its execution.

Figures 4-4 and 4-5 illustrate the following general principles of multitasking:

- The instructions in MAIN5.PR, after the CALL TQSTASK statements, execute among the MAIN.OB, TASK1.OB, and TASK2.OB sections according to whatever task has won the competition for the CPU. In contrast, the instructions in MAIN4.PR execute in predictable sections according to CALL and RETURN statements.
- Program MAIN5 does not, and cannot, contain a STOP statement. Its execution stops the entire process — including the execution of TASK1 and TASK2. Program MAIN5 could kill itself via a CALL KILL statement with no effect on TASK1 and TASK2.
- TASK1 and TASK2 will finish when they execute a RETURN statement, regardless of whether MAIN5 has executed its CALL EXIT statement. (Execution of CALL EXIT and END statements, along with the RETURN statement, result in a task's finishing.) Furthermore, TASK1 and TASK2 could be killed by themselves, by the other tasks, or by MAIN5; thus, the rectangles in Figure 4-5 representing their execution are open-ended.
- The main program unit is a task. Thus, the Link command in Figure 4-5 is

  ```
  F77LINK/TASKS=3
  ```

  instead of

  ```
  F77LINK/TASKS=2
  ```
- Some tasks may execute for the life of a program.

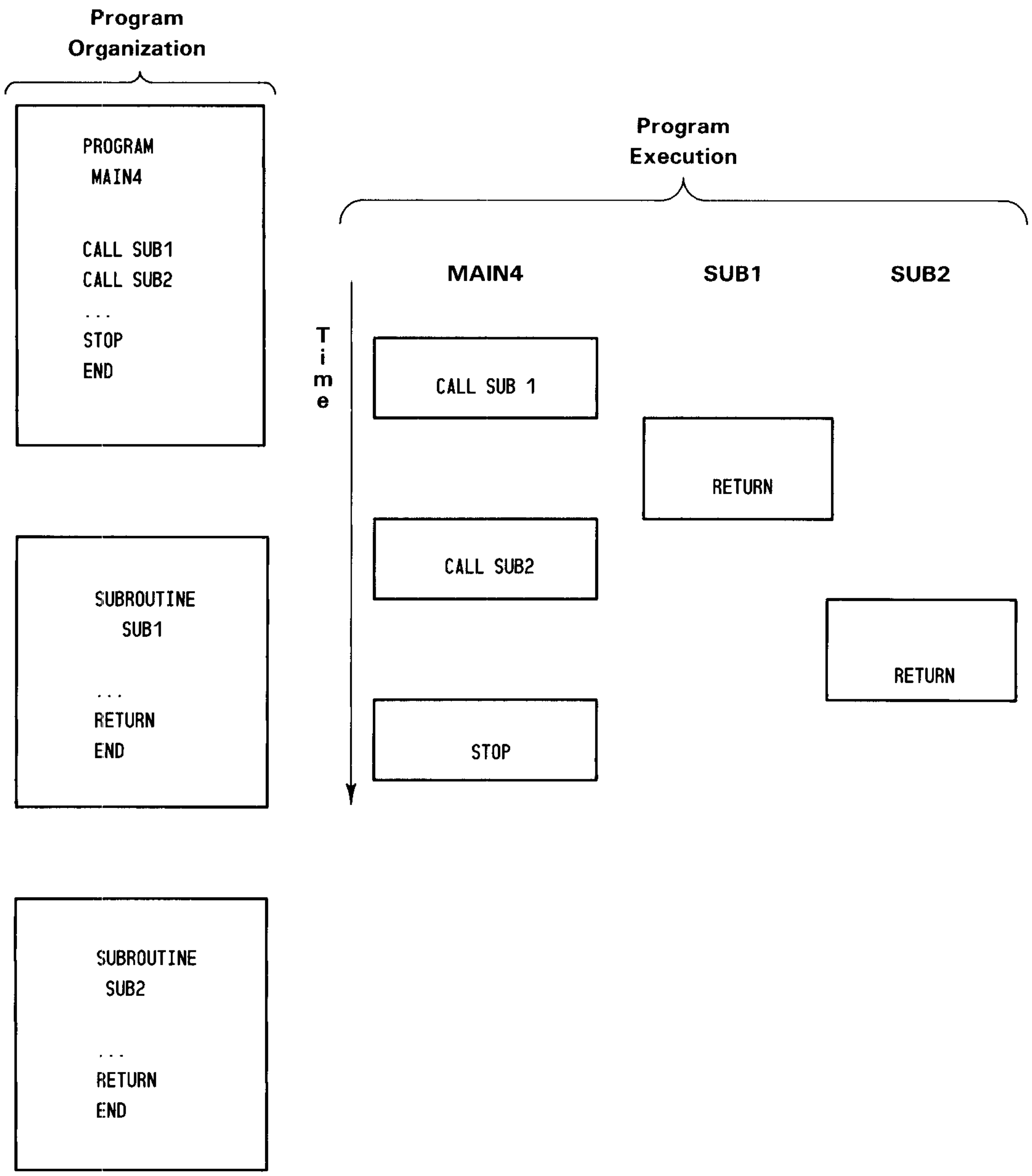
We explain subroutine TQSTASK, which MAIN5 calls, later. It's enough to say here that TQSTASK initiates the execution of a task.

Task States, Transitions, and Subroutines

This section explains the states a task has, the transitions from one state to another, and the F77-callable subroutines that cause the transitions.

Task States

It's obvious by now that competing tasks gain control and lose control of the CPU during their lifetimes. We can be more specific about the states of a task during its lifetime. Figure 4-6 shows these states.

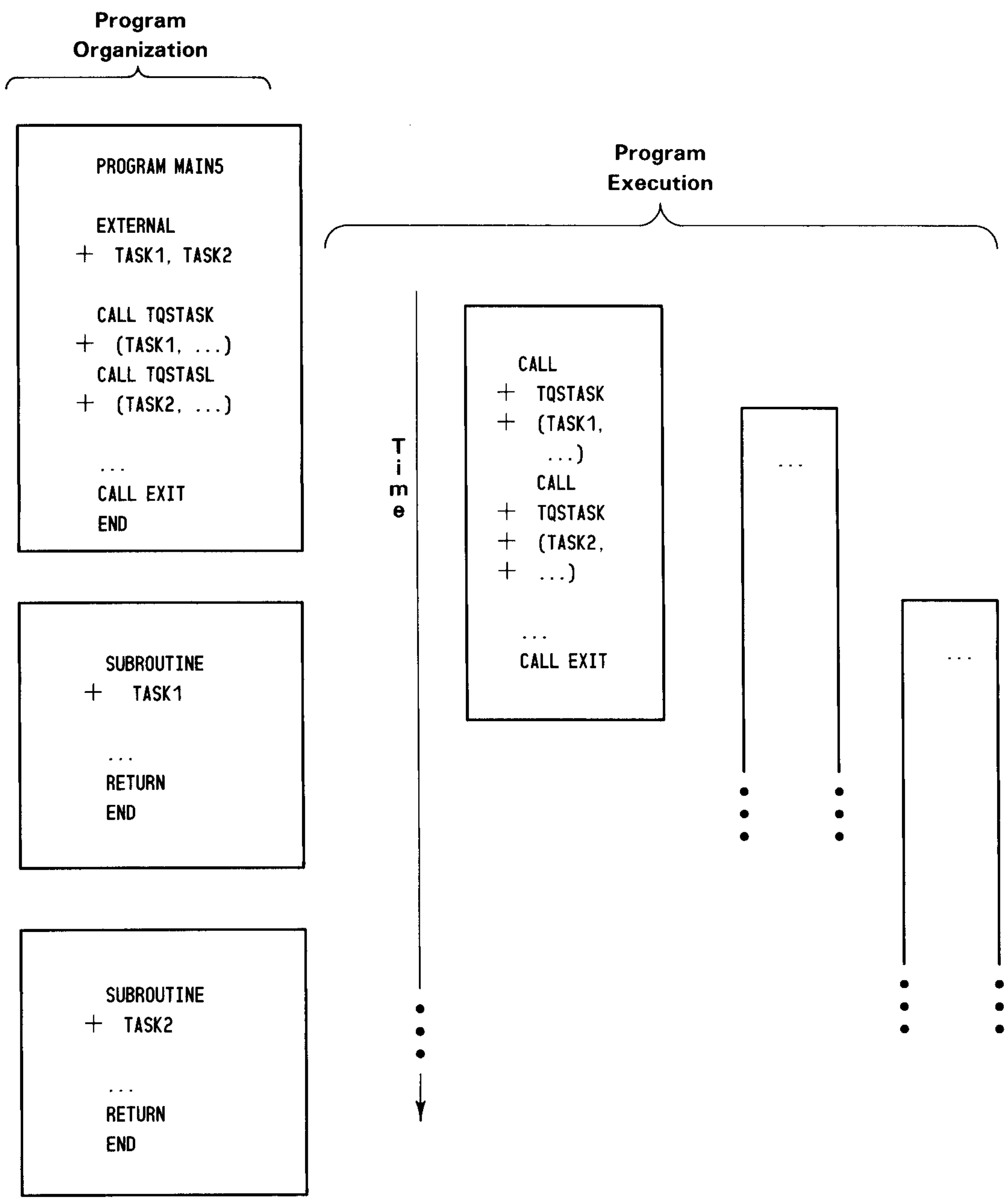


NOTE: The compilation and Link commands are:

F77 (MAIN4 SUB1 SUB2)
F77LINK MAIN4 SUB1 SUB2

ID-00107

Figure 4-4. The Organization and Execution of a Single-Task Program



NOTE: The compilation and Link commands are:

F77 (MAIN5 TASK1 TASK2)
F77 LINK/TASKS = 3 MAIN5 TASK1 TASK2

ID-00108

Figure 4-5. The Organization and Execution of a Multitask Program

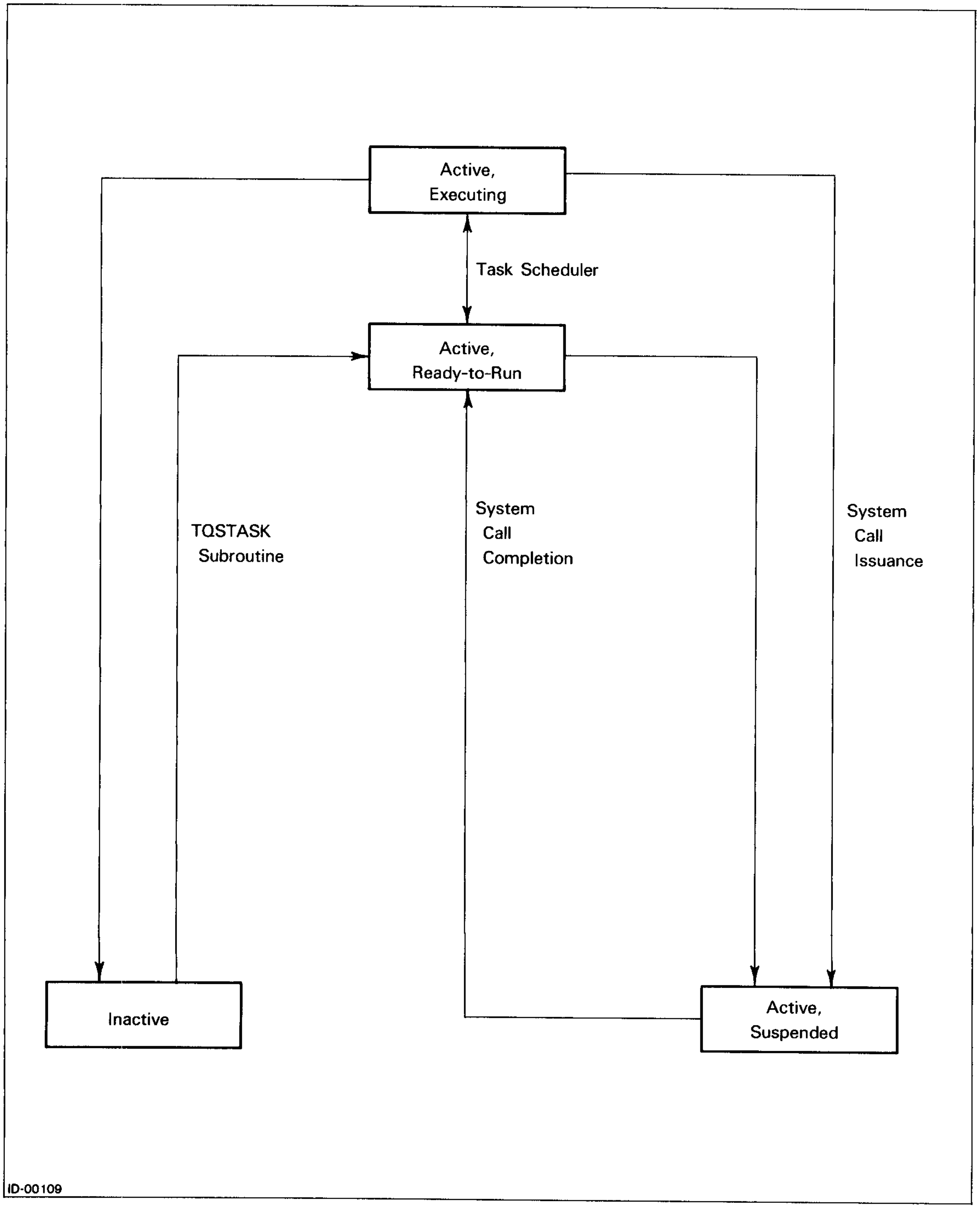


Figure 4-6. Task States

The runtime states a task can have (in increasing ability to gain control of the CPU) are:

- Inactive — Dormant. The task does not have control of the CPU. The task is dead and never even attempts to gain control of the CPU. (This is similar to a stopped lane of traffic in Figure 4-2 whose gate is locked.)
- Active — Suspended. The task does not have control of the CPU. It is unable to gain control for one or more reasons. A common reason is that a time-consuming system call must complete before the task is again eligible to execute. (This is similar to a stopped lane of traffic in Figure 4-2 whose gate is closed while the lane awaits the passage through the tunnel of its currently moving vehicle — a slow-moving truck.)
- Active — Ready-to-run. The task does not have control of the CPU. However, it is willing and able to gain control; it is merely waiting its turn. (This is similar to a stopped lane of traffic in Figure 4-2 whose gate is open, but whose vehicles are blocked by those moving from another lane.)
- Active — Executing. The task has control of the CPU. (This is similar to a moving lane of traffic in Figure 4-2 whose gate is, of course, open.)

The task scheduler is the piece of system software that selects a task for execution from among those that are ready. Naturally, you can affect the task scheduler's selection rules. One way to do this is to assign a priority to each task.

Task Transitions

A task could change its runtime state because of one of the following situations:

- The task scheduler's actions, such as suspending a task because it had been executing for a certain amount of time.
- An event, such as a planned I/O transfer or an unplanned interrupt from a device (e.g., an alarm unit).
- Instructions and requests tasks issue to the scheduler, to each other, and to themselves. For example, a task may kill itself.

The rest of this chapter deals almost exclusively with the last situation. Thus, next we'll learn how to issue these instructions and requests.

Task Subroutines

This chapter later documents 25 subroutines. But first, in this section we introduce a subset of 13 subroutines that affect task transitions. We will also modify Figure 4-6 to contain these 13 subroutines.

The subroutines may seem to have strange names. However, the core of each subroutine is one or more system calls or calls to routines in the user runtime library, URT.LB. Each F77 multitasking subroutine takes its name from a system call name or a URT.LB routine name. For example, an assembly language programmer might terminate a task via a ?KILL system call. If we remove the "?", replace it by the letter "Q" (for "question mark"), and add the letter "T" (for "task"), we obtain TQKILL. An examination of assembly language module TQKILL.SR would reveal at least one ?KILL statement.

Recall Figure 4-2 and the five-item bulleted list of standards for controller regulation. We rewrite the list to describe a multitasking program.

- Each task should have a unique positive number to identify it. When you initiate one or more tasks via a call to subroutine TQSTASK or to subroutine TQQTASK, you also specify their ID numbers. Other multitasking subroutines use the ID number to specify a particular task. If you assign no ID number (i.e., 0) to one or more tasks, or the same ID number to two or more tasks, a runtime error occurs. By default, the main program has a task ID of 1.
- Each task has a number to specify its priority. When you initiate one or more tasks via a call to subroutine TQSTASK or to subroutine TQQTASK, you also specify their priority numbers. The highest priority task has priority number 0; the lowest priority task has priority number 255. You may assign the same priority number to two or more tasks. By default, the main program has a priority of 0. Furthermore:
  - If two or more tasks are ready to run, then the task scheduler selects the one with the highest priority (i.e., lowest priority number).
  - If two or more tasks with the same priority number are ready to run, then the task scheduler selects the next one in round-robin fashion. That is, the task that executed the longest time ago among two or more tasks with equal priority executes next (first in, first out).
- Each task can communicate with any other task, including the main program. The two intertask communication calls affecting the task scheduler are TQREC (wait to receive a message) and TQXMTW (transmit a message and await its reception). Calls to TQRECNW (receive a message without waiting) and to TQXMT (transmit a message without waiting for its reception) also affect scheduling; they may cause a suspended task to become active.
- Each task controls itself and others by:
  - *Killing.* Subroutine TQIDKIL kills (makes inactive) a task with a specified ID number. Subroutine TQKILL kills the calling task.
  - *Suspension.* Subroutine TQPRSUS suspends all tasks with a specified priority. Subroutine TQIDSUS suspends a task with a specified ID number. Subroutine TQSUS suspends the calling task. TQXMTW and TQREC might also suspend the calling task.
  - *Changing priorities.* Subroutine TQIDPRI changes the priority of a task with a specified ID number. Subroutine TQPRI changes the priority of the calling task.
  - *Making tasks ready.* Subroutine TQPRRDY makes ready (changes the state from suspended to ready-to-run) all tasks with a specified priority. Subroutine TQIDRDY makes ready a task with a specified ID number.
- Any task can control and communicate with any other task. (This is in contrast to the controller/gate relationship shown in Figure 4-2). Recall that the main program is itself a task whose default ID is 1 and whose default priority is 0. However, any task may use the above subroutines to control and communicate with the main program.

We change Figure 4-6 to contain the information in this modified list. The result is in Figure 4-7.

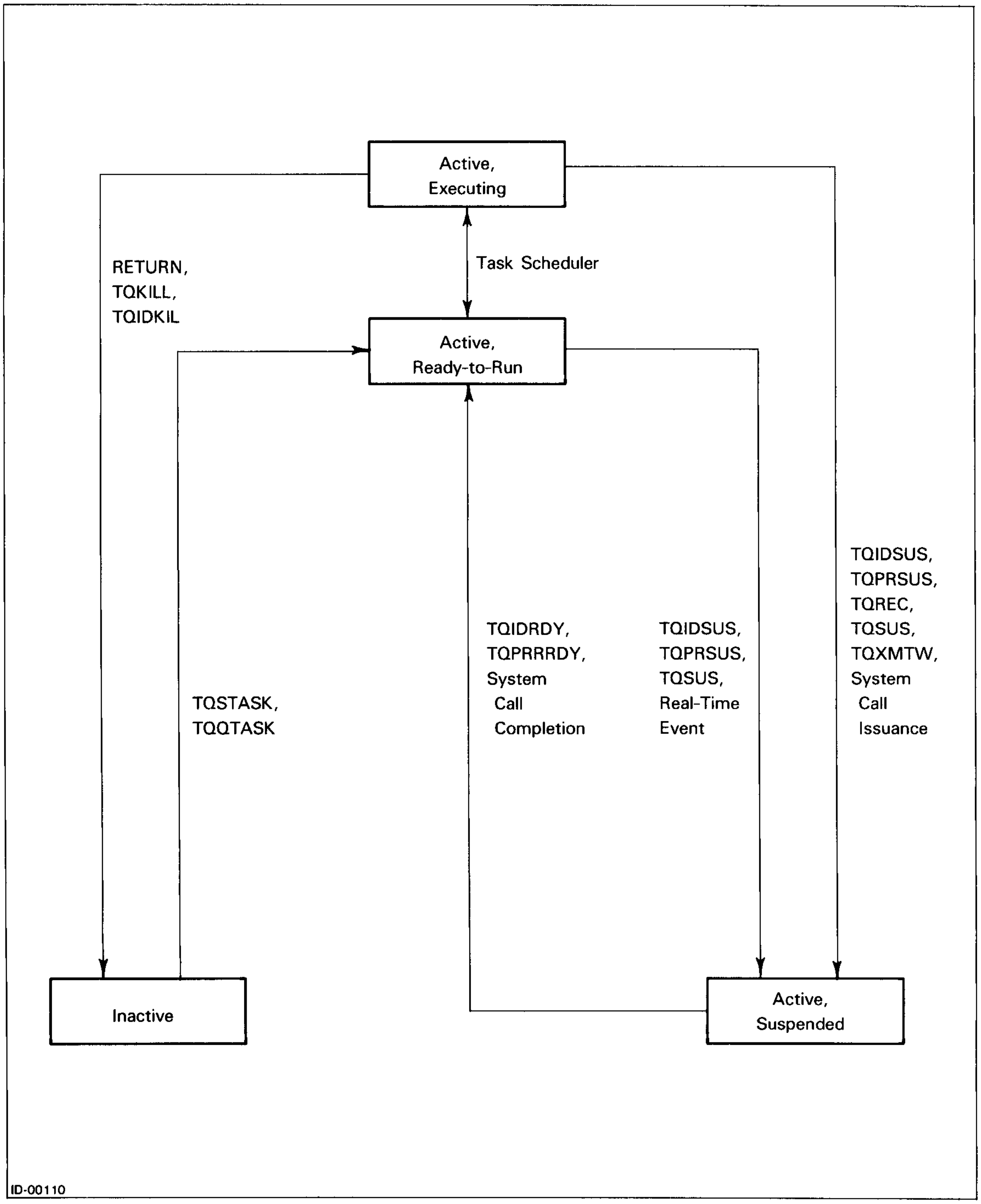


Figure 4-7. Task States and Transitions

NOTE: TQIDPRI and TQPRI do not appear in Figure 4-7. They do not immediately change the state of a task, but will affect the task scheduler's future actions with the task.

Sample Program

Figure 4-5 contains the outline of a multitasking program with its program units named MAIN5, TASK1, and TASK2. We now add to the outline and create the three program units. At runtime:

- MAIN5 initiates TASK1.
- MAIN5 initiates TASK2.
- MAIN5 kills itself.
- TASK1 opens a fresh output file TASK1.OUT.
- TASK1 sends the message 377K to TASK2 and awaits its reception.
- TASK2 opens a fresh output file TASK2.OUT.
- TASK2 awaits a message.
- After the receipt of the message has synchronized the two tasks, they remain active for about 1-1/2 seconds. At the end of this time, TASK1 KILLs TASK2 and the process terminates upon execution of TASK1's RETURN statement.

We have already summarized the multitasking subroutines appearing in the program units. The subroutines are (in chronological order of execution): TQSTASK, TQXMTW, TQREC, and TQIDKIL. Comments appear in the programs to explain the subroutines' arguments. Figure 4-8 contains MAIN5.F77. Figure 4-9 contains TASK1.F77. Figure 4-10 contains TASK2.F77.

NOTE: We assign 11 as the ID number of TASK1 instead of 1. Why? By default, MAIN5 is itself a task whose ID number is 1 (and whose priority is 0).

```
      PROGRAM MAIN5           ! TO CONTROL TASKS TASK1 AND TASK2

      COMMON /COLD/ MAILBOX   ! FOR TASK1 --> TASK2 COMMUNICATION
      EXTERNAL TASK1, TASK2   ! NECESSARY !

      PRINT *
      PRINT *, 'PRIORITY OF TASK1? '
      READ *, IPR1
      PRINT *, 'PRIORITY OF TASK2? '
      READ *, IPR2
      PRINT *, 'MAIN PROGRAM   MAIN5   EXECUTES NOW'
      PRINT *
      MAILBOX = 0             ! SHARED MAILBOX MUST CONTAIN INITIAL 0
C                                FOR TASK1 --> TASK2 COMMUNICATION

C     INITIATE TASK1 WITH AN ID NUMBER OF 11, PRIORITY <IPR1>, AND
C        DEFAULT (SYSTEM-SELECTED) STACK SIZE OF 0.
      CALL TQSTASK (TASK1, 11, IPR1, 0, IER)
      IF ( IER .NE. 0 )
     1        WRITE (*, *, ERR = 97, IOSTAT=IOS)
     2        'ERROR ', IER, ' OCCURRED INITIATING TASK1'

C     INITIATE TASK2 WITH AN ID NUMBER OF 12, PRIORITY <IPR2>, AND
C        DEFAULT (SYSTEM-SELECTED) STACK SIZE OF 0.
      CALL TQSTASK (TASK2, 12, IPR2, 0, IER)
      IF ( IER .NE. 0 )
     1        WRITE (*, *, ERR = 98, IOSTAT=IOS)
     2        'ERROR ', IER, ' OCCURRED INITIATING TASK2'

      CALL EXIT               ! I'M DONE!

   97 PRINT *, 'AT 97, IOS IS ', IOS
      STOP 97

   98 PRINT *, 'AT 98, IOS IS ', IOS
      STOP 98

      END
```
DG-25224

Figure 4-8. A Listing of Program MAIN5.F77

```
Source file: TASK1.F77
Compiled on 6-Dec-82 at 12:08:54 by AOS/VS F77 Rev 02.10.00.00
Options: F77/L=TASK1.LS

        SUBROUTINE TASK1

        COMMON /COLD/ MAILBOX

%INCLUDE 'TASK1_SYMBOLS.F77.IN'   ! FOR ?DELAY SYSTEM CALL
****  F77 INCLUDE file for system parameters ****

****   INTEGER*4 Parameters for SYSID  ****


        INTEGER*4  ISYS_WDELAY
        PARAMETER (ISYS_WDELAY = 179)   ! ?.WDELAY = 263K

****    Parameters for PARU   ****



****  END of F77 INCLUDE file for system parameters ****

        OPEN (1, FILE='TASK1.OUT', STATUS='FRESH',
     1          RECFM='DATASENSITIVE', CARRIAGECONTROL='LIST')

        WRITE (1, 100)
  100   FORMAT ('IN FILE TASK1.OUT: TASK1 HAS BEGUN<NL>')

C       SEND THE "MESSAGE" 377K TO ALL TASKS WHO ARE WAITING FOR ONE TO
C          ARRIVE IN A SHARED MEMORY LOCATION ("COMMON MAILBOX"), AND
C          WAIT UNTIL THE MESSAGE ARRIVES.

        CALL TQXMTW(MAILBOX, 377K, -1, IER)
        IF ( IER .NE. 0 )
     1          WRITE (1, 110) IER
  110           FORMAT ('ERROR ', O12, ' OCCURRED DURING TQXMTW<NL>')

C       DELAY (SUSPEND) THIS TASK FOR 1.5 SECONDS.

        IAC0 = 1500       ! SPECIFY A DELAY OF 1500 MILLISECONDS
        IAC1 = 0
        IAC2 = 0
        IER = ISYS(ISYS_WDELAY, IAC0, IAC1, IAC2)
        IF ( IER .NE. 0 ) THEN
                PRINT *, 'ERROR ', IER, ' OCCURRED IN TASK1 DURING ',
     1                   'A ?WDELAY SYSTEM CALL'
                STOP '-- PROGRAM ENDS NOW'
        ENDIF

C       1 1/2 SECONDS HAVE ELAPSED; NOW KILL TASK2.
        WRITE (1, 120)
  120   FORMAT ('PAST TQXMTW; NOW I KILL TASK2<NL>')
```

DG-25225

Figure 4-9. A Listing of Subroutine TASK1.F77 (continues)

```
          CALL TQIDKIL (12, IER)
          IF ( IER .NE. 0 )
     1           WRITE (1, 130) IER
130              FORMAT ('ERROR ', O8, ' OCCURRED TQIDKILing TASK2<NL>')

          WRITE (1, 140)
140       FORMAT ('NOW I RETURN TO MAIN PROGRAM   MAIN5')

          RETURN
          END
```

DG-25225

Figure 4-9. A Listing of Subroutine TASK1.F77 (concluded)

```
      SUBROUTINE TASK2

      COMMON /COLD/ MAILBOX

      INTEGER*4 ITIME(3)

      OPEN (2, FILE='TASK2.OUT', STATUS='FRESH',
     1          RECFM='DATASENSITIVE', CARRIAGECONTROL='LIST')

      WRITE (2, 100)
100   FORMAT ('IN FILE TASK2.OUT: TASK2 HAS BEGUN<NL>')

C     AWAIT A COMMUNICATION BY MONITORING VARIABLE <MAILBOX>.
C        WHEN <MAILBOX> IS NONZERO, ITS CONTENTS MOVE INTO <MESSAGE>.
      CALL TQREC(MAILBOX, MESSAGE, IER)
      IF ( IER .NE. 0 )
     1        WRITE (2, 110) IER
110           FORMAT ('ERROR ', O8, ' OCCURRED ON TQREC<NL>')

      WRITE (2, 120) MESSAGE
120   FORMAT ('CONTENTS OF   MESSAGE   ARE ', O8, '<NL>')

      WRITE (2, 130)
130   FORMAT ('NOW FOR UP TO 10000 LINES OF TEXT<NL><NL>')
      DO 150 I = 1, 10000
              WRITE (2, 140) I
140           FORMAT ('IN DO 150 LOOP, I = ', I5)
150   CONTINUE

      RETURN
      END
```

DG-25226

Figure 4-10. A Listing of Subroutine TASK2.F77

After the commands

```
F77 (MAIN5 TASK1 TASK2)
F77LINK/TASKS=3 MAIN5 TASK1 TASK2
```

have created MAIN5.PR, we execute it three times while varying the priority numbers. The results appear next; note how the amount of output from TASK2 varies according to its priority number. Remember: A lower priority number for a task means it is more likely to execute.

```
) X MAIN5; F/AS TASK1.OUT TASK2.OUT ↵
PRIORITY OF TASK1?   4 ↵
PRIORITY OF TASK2?   5 ↵
MAIN PROGRAM   MAIN5   EXECUTES NOW
DIRECTORY   :UDD2:F77:MARLL
TASK1.OUT      TXT   29-JUN-82   13:51:48     143
TASK2.OUT      TXT   29-JUN-82   13:51:50      36
) X MAIN5; F/AS TASK1.OUT TASK2.OUT ↵
PRIORITY OF TASK1?   5 ↵
PRIORITY OF TASK2?   5 ↵
MAIN PROGRAM   MAIN5   EXECUTES NOW
DIRECTORY :UDD2:F77:MARLL
TASK1.OUT      TXT   29-JUN-82   13:54:42     143
TASK2.OUT      TXT   29-JUN-82   13:54:42    1696
) X MAIN5; F/AS TASK1.OUT TASK2.OUT ↵
PRIORITY OF TASK1?   5 ↵
PRIORITY OF TASK2?   4 ↵
MAIN PROGRAM   MAIN5   EXECUTES NOW
DIRECTORY :UDD2:F77:MARLL
TASK1.OUT    TXT   29-JUN-82   13:55:28     143
TASK2.OUT    TXT   29-JUN-82   13:55:28    6144
)
```

If you create MAIN5.PR and execute it your results probably won't be exactly the same as these. TASK1 delays execution for a variable time period (about 1.5 seconds) and thus TASK2 writes varying numbers of lines into TASK2.OUT. The overall load on the system also affects the amount of output TASK2 creates.

Re-entrant Code

In certain situations, it is appropriate for two or more tasks to execute exactly the same sequence(s) of instructions yet still remain independent of one another and use their own sets of data. In such cases, it is more efficient for all of these tasks to share a single set of instructions than to duplicate the code several times. This sharing is possible provided that the code does not modify itself, and that F77 sets aside a separate data space for each task.

To provide this runtime space for each task, F77 allocates a part of the memory area known as its *runtime stack* for variables that the task uses. Thus, it separates the unmodified, shared code from the multiple modified data areas. We call the shared code *re-entrant* code since various tasks are entering and using the code at the same time.

NOTE: By default, F77 allocates variables on the runtime stack unless:

- DATA statements assign them initial values.
- A SAVE statement specifies or implies them.
- The program units they reside in are compiled with the /SAVEVARS switch.
- They exist in COMMON.

The actual sequence of events in the use of re-entrant code is as follows. Each time you initiate a task in a multitasking program, F77 assigns the task a *task control block* and a section of the runtime stack. This task control block keeps track of which instruction the task is executing and the data space allocated to the task. Two or more tasks can execute a single subroutine (the re-entrant code) at one time although the tasks cannot execute the same statement at a given instant. Figure 4-11 illustrates the status of the program at one point in time. It is not a dynamic picture of these operations.

For example, suppose you want two tasks to move concurrently through subroutine SUBRA, three tasks to move concurrently through subroutine SUBRB, and one task to move through subroutine SUBRC. Assume also that the main program is named MAIN12. The structure of MAIN12.F77 is as follows.

```
      PROGRAM MAIN12

      EXTERNAL SUBRA, SUBRB, SUBRC

      ...

C     CALL TQSTASK(SUBRA, 11, ...)     ! ID IS 11
C     CALL TQSTASK(SUBRA, 12, ...)     ! ID IS 12

C     START 3 TASKS THROUGH SUBROUTINE <SUBRB>.  ASSUME THEY
C         REMAIN ACTIVE UNTIL WE EXPLICITLY KILL THEM.
      CALL TQSTASK(SUBRB, 21, ...)     ! ID IS 21
      CALL TQSTASK(SUBRB, 22, ...)     ! ID IS 22
      CALL TQSTASK(SUBRB, 23, ...)     ! ID IS 23

      CALL TQSTASK(SUBRC, 31, ...)     ! ID IS 31

      ...

      CALL TQIDKIL (22, IER)           ! SUBRB IS STILL ACTIVE
      CALL TQIDKIL (23, IER)           ! SUBRB IS STILL ACTIVE
      CALL TQIDKIL (21, IER)           ! SUBRB IS FINALLY INACTIVE
C     ALL TASKS IN SUBROUTINE <SUBRB> ARE NOW INACTIVE.
      ...

      END
```

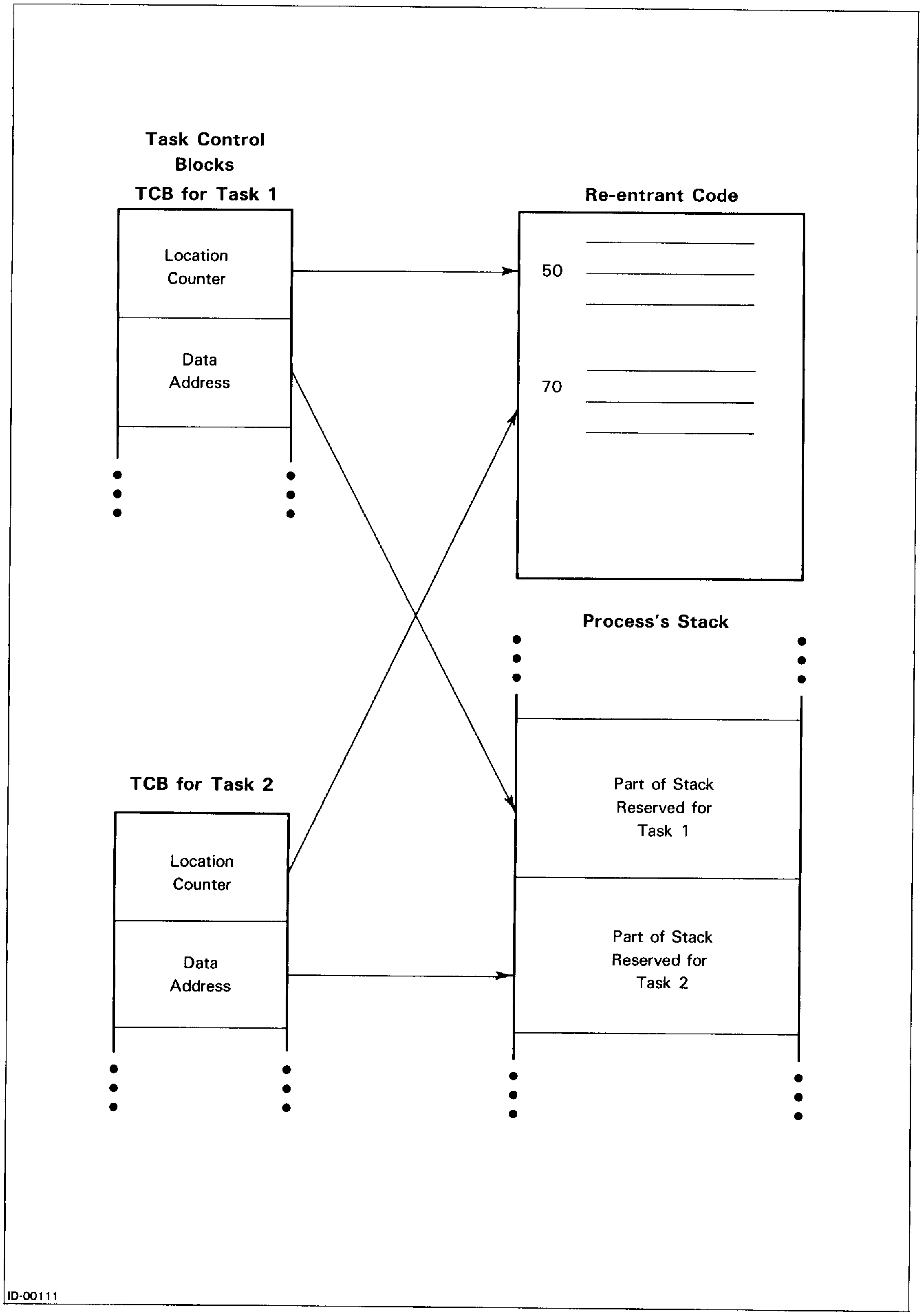


Figure 4-11. Task Control Blocks and the Use of Re-entrant Code

The compilation and Link instructions would have the following general outline.

```
F77 (MAIN12 SUBRA SUBRB SUBRC)
F77LINK/TASKS=7 MAIN12 SUBRA SUBRB SUBRC
```

Multitasking Subroutines

Table 4-1 shows the correspondence between called-by-F77 subroutines and the operating system calls (AOS/VS) that ultimately perform a subroutine's multitasking request. The "F77" column determines the alphabetical order of the three columns.

Table 4-1. F77 and AOS/VS Multitasking Calls and their Functions

F77	AOS/VS	Function
TQDQTSK	?DQTSK	Dequeue a previously queued tasks.
TQDRSCH	?DFRSCH, ?DRSCH	Disable a scheduling and optionally return a flag.
TQERSCH	?ERSCH	Enable scheduling.
TQIDKIL	?IDKIL	Kill a task specified by its ID.
TQIDPRI	?IDPRI	Change the priority of a task specified by its ID.
TQIDRDY	?IDRDY	Ready a task specified by its ID.
TQIDSTAT	?TIDSTAT	Get a task's status.
TQIDSUS	?IDSUS	Suspend a task specified by its ID.
TQIQTSK	?IQTSK	Create a queued task manager.
TQKILAD	?KILAD	Define a kill processing routine.
TQKILL	?KILL	Kill the calling task.
TQMYTID	?MYTID	Get the priority and ID of the calling task.
TQPRI	?PRI	Change the priority of the calling task.
TQPRKIL	?PRKIL	Kill all tasks of a specified priority.
TQPROT	none	Start a protected area.
TQPRRDY	?PRRDY	Ready all tasks of a specified priority.
TQPRSUS	?PRSUS	Suspend all tasks of a specified priority.
TQQTASK	?TASK	Create a queued task.
TQREC	?REC	Receive an intertask message.
TQRECNW	?RECNW	Receive an intertask message without waiting.
TQSTASK	?TASK	Initiate one task.
TQSUS	?SUS	Suspend the calling task.
TQUNPROT	none	Exit a protected area.
TQXMT	?XMT	Transmit an intertask message.
TQXMTW	?XMTW	Transmit an intertask message and wait for its reception.
none	?IDGOTO	Redirect a task.
none	?IFPU	Initialize the floating-point unit.
none	?TRCON	Read a task message from the process console.

For example, suppose that your AOS/VS F77 program unit contains a

CALL TQIDPRI (arguments)

statement. When Link processes the program unit's .OB file, it places code from LANG_RT.LB and F77IO_MT.LB into the main program's program (.PR) file. At runtime this code makes a ?IDPRI operating system call. However, not all F77 multitasking subroutines result in LANG_RT.LB code, F77IO_MT.LB code, and exactly one runtime operating system call.

Note in Table 4-1 that:

- ?IDGOTO, ?IFPU, and ?TRCON have no corresponding F77-callable subroutines. However, some of these subroutines make a ?IFPU system call; none of them makes a ?IDGOTO or ?TRCON call.
- TQPROT and TQUNPROT have no direct correspondence with any system calls.
- TQQTASK has no direct correspondence with any system calls. However, it uses ?TASK to carry out its function of queued task creation.

Assembly Language Interface

FORTRAN 77 also provides a set of routines to replace multitasking system calls. These routines are in F77IO_MT.LB and LANG_RT.LB. They:

- Take accumulator values and parameter packets identical to those of the corresponding system calls.
- Make a system call.
- Take the conventional error or normal return.

The difference is that the replacement routines provide the same protection of the runtime database integrity for the multitasking routines as do the F77-callable routines.

Assembly Language Calls

You can invoke these subroutines from assembly language programs, as well as from FORTRAN 77 programs. To do this, remove any multitasking statements of the form

```
?<call>     ; make a system call
```

The correct replacement is a statement of the form

```
LCALL T?<call> ; make a system call via a routine in LANG_RT.LB
```

In each of these two cases, AC2 must contain the packet address if required. All other statements and declarations related to the system call remain the same. You must also add .EXTL statements. For example, you would replace

```
?IDKIL with        .EXTL T?IDKIL
                   LCALL T?IDKIL
```

Such replacement results in protection of runtime database integrity.

Example

Suppose you want to change the priority of task number 7 to 5 by using subroutine T?IDPRI instead of by making a call to ?IDPRI. The skeleton assembly language code resembles the following.

```
        .EXTL   T?IDPRI ; DECLARE ROUTINES AS LONG EXTERNALS.
        ; ...
        NLDAI   7,0     ; TASK NUMBER 7 WILL HAVE A ...
        NLDAI   5,1     ;    ... PRIORITY OF 5.
        LCALL   T?IDPRI ; DO IT!  (FORMERLY: ?IDPRI ; DO IT!)
        WBR     ERIDPRI ; ERROR RETURN
                        ; NORMAL RETURN: CONTINUE
        ; ...
ERIDPRI:                ; RESPOND TO ERROR FROM T?IDPRI.
        ; ...
        ; ...
```

Routine Names

The complete list of multitasking routines accessible via the

LCALL <routine name>

mechanism is as follows.

T?DQTSK	T?KILAD	T?QTASK
T?DRSCH	T?KILL	T?REC
T?ERSCH	T?MYTID	T?RECNW
T?IDKIL	T?PRI	T?STASK
T?IDPRI	T?PRKIL	T?SUS
T?IDRDY	T?PRRDY	T?TIDSTAT
T?IDSUS	T?PRSUS	T?XMT
T?IQTSK		T?XMTW

There is no F77-callable subroutine named TQDFRSCH. However, your AOS/VS assembly language program can contain a

```
LCALL T?DFRSCH
```

statement to call ?DFRSCH. This way, your program both disables scheduling and knows (via a flag — the "F" of DFRSCH) whether or not scheduling already was disabled at the time ?DFRSCH executed. If it was, then ?DFRSCH places the value of ?DSCH in AC0.

LANG_RT.LB and F77IO_MT.LB provide you with entry points for the protected-against-KILLing-and-SUSPension code paths that TQPROT and TQUNPROT create. The names of these entry points are T?PROT and T?UNPROT.

Finally, after assembly, use macro F77LINK to create your program file. This macro has Link search LANG_RT.LB and F77IO_MT.LB (along with other FORTRAN 77 library files) according to the multitasking statements of your program.

Conversion of FORTRAN 5 Multitasking Programs

You might have AOS FORTRAN 5 or RDOS FORTRAN 5 multitasking programs and want to convert them to FORTRAN 77 programs. These FORTRAN 77 programs will use the multitasking routines from libraries LANG_RT.LB and F77IO_MT.LB.

You have two ways to convert FORTRAN 5 multitasking CALLs such as

CALL XMT (arguments)

and statements such as

ANTICIPATE 4

to FORTRAN 77 multitasking CALLs.

Rewrite Each Multitasking CALL or Statement

Rewrite each FORTRAN 5 multitasking CALL or statement according to the rules of its equivalent FORTRAN 77 CALLs. The names of these subroutines are in Table 4-1 at the beginning of this chapter; their explanations appear later in this chapter. For example, you might replace

```
CALL SUS        ; SUSPEND THIS TASK
```

with

```
CALL TQSUS (IER)     ! SUSPEND THIS TASK
```

You should include an error-processing routine for errors arising from the execution of the multitasking routines.

Use a Conversion Library

Use the set of F77 subroutines supplied with F77. These subroutines have the same names as FORTRAN 5 subroutines, and they convert a FORTRAN 5 name/arguments CALL to a FORTRAN 77 name/arguments CALL. Their location is directory F77_F5MT.

For example, the outline of ARDY.F77 is similar to the following:

```
C       SUBROUTINE ARDY.F77 TO PERFORM THE FUNCTION
C          OF READYING ALL TASKS OF A GIVEN PRIORITY
C          IN AN AOS/VS RUNTIME ENVIRONMENT.
        SUBROUTINE ARDY (PRIORITY_2)
        INTEGER*2 PRIORITY_2
        INTEGER*4 PRIORITY, IER
        PRIORITY = PRIORITY_2     ! DUPLICATE 2-BYTE PRIORITY
C                                   NUMBER AS 4 BYTES.
C       F77/TQPRRDY IS EQUIVALENT TO F5/ARDY
        CALL TQPRRDY (PRIORITY, IER)
        RETURN
        END
```

You might have to change some of the arguments in the FORTRAN 5 CALLs. For example,

```
CALL XMT (MAILBOX, MESSAGE, $100)
```

is correct in FORTRAN 5, but the "$" of the third argument makes the entire statement incorrect in FORTRAN 77. You must change this line to

```
CALL TQXMT (MAILBOX, MESSAGE, *100)
```

And, you might want to create a .LB file for the F77 source subroutines. This library file would become part of your F77LINK macro.

For example, suppose you decide to leave all FORTRAN 5 CALLs to subroutines AKILL, ARDY, and SUSP alone. This means that you must manually convert the other multitasking CALLs to FORTRAN 77 names and arguments. Suppose also that program TYPICAL.F5 has a maximum of five tasks and that you have edited it into program TYPICAL.F77 without making any changes to the AKILL, ARDY, and SUSP CALLs. Then, give the following CLI commands:

- F77 (AKILL.F77 ARDY.F77 SUSP.F77)
- X LFE N F5_MT/O AKILL ARDY SUSP
- F77 TYPICAL.F77
- F77LINK/TASKS=5 TYPICAL F5_MT.LB

The /TASKS=5 F77LINK switch directs Link to search LANG_RT.LB and F77IO_MT.LB. Program file TYPICAL.PR is now ready for execution.

Recommended Conversion Method

We recommend the first method of conversion — rewriting each FORTRAN 5 multitasking CALL or statement to its FORTRAN 77 equivalent statements. Your program will execute slightly faster than if you use a conversion library. More significantly, some FORTRAN 5 multitasking CALLs and statements are not in the conversion subroutines because they have no FORTRAN 77 equivalents. CALL GETEV, with its reference to an event number, is an example. You can print the Release 2.0 conversion subroutines from directory F77_F5MT and read the *FORTRAN 5 Programmer's Guide* to see what FORTRAN 5 multitasking CALLs and statements are missing in the conversion subroutines.

Multitasking via the ISYS Function?

So far we have mentioned the following three ways of hooking into the multitasking capabilities of AOS/VS:

- Using traditional system calls from assembly language programs, such as ?IDPRI.
- Using FORTRAN 77 CALLs such as CALL TQIDPRI (arguments) to multitasking routines in LANG_RT.LB and F77IO_MT.LB.
- Using assembly language interface routines for system calls resulting in statements such as LCALL T?IDPRI.

A fourth method is theoretically possible: the generalized system call mechanism, explained in Chapter 3. For example, you might be tempted to write FORTRAN 77 statements such as these:

```
C       SET THE PRIORITY OF TASK NUMBER 7 TO 5.
        INTEGER*4 AC0, AC1, AC2, IER, ISYS
C       ...
        AC0 = 7
        AC1 = 5
        IER = ISYS (ISYS_IDPRI, AC0, AC1, AC2)  ! DO IT!
C       ...
```

However, we do *not* recommend this fourth method. It may interfere with certain F77 runtime routines and internal databases.

Link Switches for F77 Multitasking

The addition of the multitasking routines could affect your commands to Link. The new F77LINK switches are /IOCONFLICT and /TASKS.

/IOCONFLICT Switch

This F77LINK switch has three values: ERROR, IGNORE, and QUEUE. QUEUE is the default. That is,

```
F77LINK MY_PROG
```

and

```
F77LINK/IOCONFLICT=QUEUE MY_PROG
```

give identical results.

As the name implies, programs Linked with this switch could report a runtime error when an I/O conflict occurs. Such a conflict happens when one task "A" attempts to access a unit that another task "B" is using. Then:

- If /IOCONFLICT=ERROR, task "A" receives an error value from its I/O statement that unsuccessfully attempted to access the unit. The success or failure of task "B" is unaffected by "A's" attempted simultaneous access of the unit.
- If /IOCONFLICT=IGNORE, the F77 runtime routines don't check for simultaneous access of a unit by more than one task. The results are unpredictable and the runtime I/O databases could be compromised. You would use this switch setting if speed is important and you can guarantee that only one task will access a given unit at any time.
- If /IOCONFLICT=QUEUE or is not specified, task "A" does not receive an error value from its I/O statement that attempted to access the unit. It waits until "B" is finished with the unit before continuing with its I/O operation.

/TASKS=n Switch

F77LINK.CLI passes this switch down to Link. For multitasking programs you must specify it to either F77LINK.CLI or to the Link command. For example, suppose your program file (.PR file) will have at most five tasks, and it uses F77 multitasking routines. Then

For an	Specify
F77 program:	F77LINK/TASKS=5 MY_PROG ...
Assembly language program:	F77LINK/TASKS=5 MY_PROG ...

Task Fatal Errors

Several types of runtime errors that were previously fatal to a process are now fatal to a task. These errors are:

- I/O runtime.
- Arithmetic exceptions (such as overflow).
- Subscript/substring addressing.
- Stack overflow/underflow.

Previously, these errors resulted in the process' termination. In general, only internal consistency errors will now terminate a process.

Initial Task

The initial task — the main program — has an ID of 1 and a priority of 0 when it begins execution. Keep this in mind as you code CALLs to TQQTASK and to TQSTASK which, in turn, initiate tasks.

Documentation of Multitasking Calls

The rest of this chapter describes the individual F77-callable multitasking routines alphabetically. The explanation of each routine includes:

- Its name and function.
- Its format and argument names for CALLing by F77.
- Descriptions of each argument.
- If possible, a sample CALL and related statements.

The Result Code Argument

All the multitasking subroutines have an argument that receives a code to indicate the result of the subroutine's execution. This argument appears in this chapter as ier. It is always the last argument in the argument list. If no exceptional condition occurs during the subroutine's execution, ier contains zero. Otherwise, ier contains one of the following:

- An operating system error code. (See the beginning of PARU.32.SR; or, give the CLI command MESSAGE/D ier.) You can also use subroutine ERRCODE, explained in Chapter 2, to report the error.
- An error code from ERR.F77.IN, which contains the same codes as F77ERMES.SR.
- An error code from PARLANG_RT.SR.
- An error code from LANG_RTERMES.SR.

For example, suppose a F77 program contains the statements

```
INTEGER*4 TASK_ID
TASK_ID = 8
CALL TQIDKIL (TASK_ID, IER)
```

If IER is zero after your program returns control from TQIDKIL, then no exceptional condition has occurred. Otherwise, IER contains an error code from one of the above files.

TQDQTSK

Dequeue a previously queued task.

Format

CALL TQDQTSK(task_definition_packet, ier)

Arguments

task_definition_packet	is an INTEGER*2 (*not* INTEGER*4) array that contains the task definition packet. Read the restrictions on certain words of the packet in the "Arguments" section of the explanation of TQQTASK.
ier	is an INTEGER*4 variable or array element that receives the result code.

Example

A program unit must execute CALL TQIQTSK and CALL TQQTASK statements in this order before it can execute a CALL TQDQTSK statement. The following program creates a queued task manager to initiate six tasks whose IDs are 14, 15, 16, 17, 18, and 19. Then, it dequeues all six tasks. Program CALL_TQDQTSK follows.

```
C       SAMPLE AOS/VS F77 PROGRAM CALL_TQDQTSK
        EXTERNAL SUB_QDTASK   ! SUBROUTINE WHOSE NAME IS AN ARGUMENT
C                                  TO    TQQTASK
        INTEGER*2 ETDP(0:21)  ! EXTENDED TASK DEFINITION PACKET
C                                  MUST BE INTEGER*2
        INTEGER*4 TASK_ID, PRIORITY, IER
C       ...
C       CREATE A TASK WHICH IS THE QUEUED TASK MANAGER.
        TASK_ID = 4
        PRIORITY = 2
        CALL TQIQTSK(TASK_ID, PRIORITY, IER)
        IF ( IER .NE. 0 ) GO TO 9000
C       ...
```

```
C       SET UP THE 22-WORD EXTENDED TASK DEFINITION PACKET.  TQQTASK
C          WILL USE THIS PACKET AND   TQDQTSK   WILL ALSO USE IT.

        ETDP(00) = 0   ! ?DLNK:    0 FOR THE EXTENDED PACKET

        ETDP(01) = 0   ! ?DLNL:    0 FOR THIS RESERVED WORD
        ETDP(02) = 0   ! ?DLNKB:   0 FOR THIS RESERVED WORD
        ETDP(03) = 0   ! ?DLNKBL:  0 FOR THIS RESERVED WORD

        ETDP(04) = 7   ! ?DPRI:    THE PRIORITY NUMBER FOR
                       !              EACH TASK

        ETDP(05) = 14  ! ?DID:     TASK IDS ARE THE NONZERO
                       !              NUMBERS 14, 15, 16, ...

        ETDP(06) = 0   ! ?DPC:     TASK STARTING ADDRESS ...
        ETDP(07) = 0   ! ?DPCL:       ... IS SUPPLIED BY F77.

        ETDP(08) = 0   ! ?DAC2:    TASK MESSAGE ...
        ETDP(09) = 0   ! ?DCL2:       ... IS SUPPLIED BY F77.

        ETDP(10) = 0   ! ?DSTB:    ACCEPT F77'S ...
        ETDP(11) = 0   ! ?DSTL:       ... STACK BASE.

        ETDP(12) = 0   ! ?DSFLT:   ACCEPT F77'S STACK FAULT HANDLER.

        ETDP(13) = 0       ! ?DSSZ:   EACH (OF THE SIX) TASK HAS ...
        ETDP(14) = 6*0     ! ?DSSL:      ... BY DEFAULT A 5000-WORD STACK.

        ETDP(15) = 0       ! ?DFLGS:  THE TASK FLAG WORD ...
                           !             ... IS SUPPLIED BY F77.

        ETDP(16) = 0       ! ?DRES:   0 FOR THIS RESERVED WORD

        ETDP(17) = 6       ! ?DNUM:   THERE ARE SIX TASKS.

        ETDP(18) = 14      ! ?DSH:    INITITATE THE TASKS AT THE
        ETDP(19) = 906     ! ?DSMS:      NEXT OCCURRENCE OF 2:15:06 PM.

        ETDP(20) = 3       ! ?DCC:    THREE INITIALIZATION ATTEMPTS
                           !             ARE ENOUGH, ...
        ETDP(21) = 10      ! ?DCI:    ... SPACED 10 SECONDS APART.
```

```
C       ...
C       CREATE THE QUEUED TASK.
        CALL TQQTASK(SUB_QDTASK, ETDP, IER)
        IF ( IER .NE. 0 ) GO TO 9010

C
C       ...
C       ...
C       ...
C
C       NOW REMOVE ALL SIX TASKS (IDS 14-19) PREVIOUSLY QUEUED
C          FOR INITIATION BY A CALL TO   TQQTASK   .  WE DON'T
C          ALTER THE PACKET GIVEN TO   TQQTASK   .
        CALL TQDQTSK(ETDP, IER)
        IF ( IER. NE. 0 ) GO TO 9020

C       ...
C       ERROR ROUTINES ARE NEXT.
 9000   CONTINUE  ! HANDLE AN ERROR FROM TQIQTSK.
C       ...
 9010   CONTINUE  ! HANDLE AN ERROR FROM TQQTASK.
C       ...
 9020   CONTINUE  ! HANDLE AN ERROR FROM TQDQTSK.
C       ...

        STOP
        END
```

TQDRSCH

Disable scheduling and optionally return a previous status.

Format

CALL TQDRSCH(*[previously_disabled,]* ier)

Arguments

previously_disabled	is an optional LOGICAL*4 variable or array element, which if supplied: • Receives a value of .TRUE., if scheduling was disabled before the call. • Receives a value of .FALSE., if scheduling was not disabled before the call.
ier	is an INTEGER*4 variable or array element that receives the result code.

Example

```
C       SAMPLE F77 PROGRAM CALL_TQDRSCH
        LOGICAL*4 PREV_DIS
        INTEGER*4 IER
C       ...
        CALL TQDRSCH(PREV_DIS, IER)
C
C       ... DO THINGS WITH SCHEDULING DISABLED ...
C
        IF ( .NOT. PREV_DIS )
     1     CALL TQERSCH(IER)          ! IF SCHEDULING WAS NOT PREVIOUSLY
C                                        DISABLED, THEN RE-ENABLE IT
C                                        SINCE I'VE DONE MY THINGS WITH
C                                        SCHEDULING DISABLED.
C       ...
        END
```

TQERSCH
Enable scheduling.

Format

CALL TQERSCH(ier)

Argument

ier   is an INTEGER*4 variable or array element that receives the result code.

Example

```
C       SAMPLE F77 PROGRAM CALL_TQERSCH
        INTEGER*4 IER
C       ...
        CALL TQERSCH(IER)
        PRINT 10, IER
   10   FORMAT (' ERROR CODE RETURNED FROM TQERSCH IS ', O6, 'K')
C       ...
        END
```

TQIDKIL

Kill a task specified by its ID.

Format

CALL TQIDKIL(taskid, ier)

Arguments

taskid — is an INTEGER*4 expression that contains the ID of the task you want to kill.

ier — is an INTEGER*4 variable or array element that receives the result code.

Example

```
C       SAMPLE F77 PROGRAM CALL_TQIDKIL
        INTEGER*4 TASK_ID, IER
C       ...
C       NOW KILL TASK NUMBER 9.
        TASK_ID = 9
        CALL TQIDKIL(TASK_ID, IER)
        PRINT 10, IER
   10   FORMAT (' ERROR CODE RETURNED FROM TQIDKIL IS ', O6, 'K')
C       ...
        END
```

TQIDPRI

Change the priority of a task specified by its ID.

Format

CALL TQIDPRI(taskid, priority, ier)

Arguments

taskid — is an INTEGER*4 expression that contains the ID of the task whose priority you want to change.

priority — is an INTEGER*4 expression that contains the new priority of the task.

ier — is an INTEGER*4 variable or array element that receives the result code.

Example

```
C       SAMPLE F77 PROGRAM CALL_TQIDPRI
        INTEGER*4 IER
C       ...
C       CHANGE THE PRIORITY OF TASK NUMBER 7 TO 5
        CALL TQIDPRI(7, 5, IER)
        PRINT 10, IER
   10   FORMAT (' ERROR CODE RETURNED FROM TQIDPRI IS ', O6, 'K')
C       ...
        END
```

TQIDRDY

Ready a task specified by its ID.

Format

CALL TQIDRDY(taskid, ier)

Arguments

taskid is an INTEGER*4 expression that contains the ID of the task you want to make ready.

ier is an INTEGER*4 variable or array element that receives the result code.

Example

```
C       SAMPLE AOS/VS PROGRAM CALL_TQIDRDY
        INTEGER*4 TASK_ID, IER
C       ...
C       MAKE READY TASK NUMBER 19.
        TASK_ID = 19
        CALL TQIDRDY (TASK_ID, IER)
        PRINT 10, IER
   10   FORMAT (' ERROR CODE RETURNED FROM TQIDRDY IS ', O6, 'K')
C       ...
        END
```

TQIDSTAT

Get a specified task's status.

Format

CALL TQIDSTAT(taskid, status, ier)

Arguments

taskid — is an INTEGER*4 expression that contains the task's ID.

status — is an INTEGER*4 variable or array element that receives the task's status word. This word is offset ?TSTAT of the task's task control block (TCB).

ier — is an INTEGER*4 variable or array element that receives the result code.

Example

```
C       SAMPLE F77 PROGRAM CALL_TQIDSTAT
        INTEGER*4 TASK_ID, STATUS, IER
C       ...
C       GET AND PRINT TASK 16'S STATUS WORD.
        TASK_ID = 16
        CALL TQIDSTAT(TASK_ID, STATUS, IER)
        PRINT 10, STATUS
   10   FORMAT (" TASK 16'S STATUS WORD IS ", O12, "K")
C       ...
        END
```

TQIDSUS

Suspend a task specified by its ID.

Format

CALL TQIDSUS(taskid, ier)

Arguments

taskid is an INTEGER*4 expression that contains the ID of the task you want to suspend.

ier is an INTEGER*4 variable or array element that receives the result code.

Example

```
C       SAMPLE AOS/VS PROGRAM CALL_TQIDSUS
        INTEGER*4 IER
C       ...
C       SUSPEND TASK NUMBER 18.
        CALL TQIDSUS (18, IER)
        PRINT 10, IER
   10   FORMAT (' ERROR CODE RETURNED FROM TQIDSUS IS ', O6, 'K')
C       ...
        END
```

TQIQTSK

Create a queued task manager.

Format

CALL TQIQTSK(taskid, priority, ier)

Arguments

taskid — is an INTEGER*4 expression that specifies the ID of the queued task manager; the task manager is itself a task. Count this task as you calculate *n* for the /TASKS=n F77LINK switch.

priority — is an INTEGER*4 variable or array element that specifies the priority of the task.

ier — is an INTEGER*4 variable or array element that receives the result code.

Example

```
C       SAMPLE F77 PROGRAM CALL_TQIQTSK
        INTEGER*4 TASK_ID, PRIORITY, IER
C       ...
C       CREATE A TASK TO SERVE AS THE QUEUED TASK MANAGER FOR
C          THIS PROGRAM WITH AN ID OF 5 AND A PRIORITY OF 3.
        TASK_ID = 5
        PRIORITY = 3
        CALL TQIQTSK(TASK_ID, PRIORITY, IER)
        PRINT 10, IER
   10   FORMAT (' ERROR CODE RETURNED FROM TQIQTSK IS ', O6, 'K')
C       ...
        END
```

TQKILAD

Define a kill processing routine.

Format

CALL TQKILAD(subroutine-name, ier)

Arguments

subroutine-name — is the name of a subroutine that will receive control the first time that another task ("A") attempts to KILL the task ("B") containing a CALL TQKILAD statement. However, this latter task ("B") is terminated by its own CALL TQKILL, STOP, or RETURN statements without its CALLing subroutine-name. Declare subroutine-name EXTERNAL in any task containing a CALL to TQKILAD.

ier — is an INTEGER*4 variable or array element that receives the result code.

Example

```
C       ASSUME THAT THIS IS AOS/VS TASK "UNIT_B.F77".  ASSUME ALSO
C          THAT WE WANT IT TO CALL SUBROUTINE "C_SUB" WHENEVER SOME
C          OTHER TASK (CALL IT "UNIT_A.F77") ATTEMPTS TO TERMINATE
C          TASK "UNIT_B.F77".  HOWEVER, SUBROUTINE TQKILL OR THE
C          RETURN AND STOP STATEMENTS IN "UNIT_B.F77" WILL TERMINATE
C          "UNIT_B.F77" WITHOUT RESULTING IN A CALL TO "C_SUB".
C       ...
        INTEGER*4 IER
        EXTERNAL C_SUB
C       ...
        CALL TQKILAD (C_SUB, IER)
        PRINT 10, IER
   10   FORMAT (' ERROR CODE RETURNED FROM TQKILAD IS ', O6, 'K')
C       ...
        END
```

TQKILL

Kill the calling (current) task.

Format

CALL TQKILL(ier)

Argument

ier   is an INTEGER*4 variable or array element that receives the result code.

Example

```
C       SAMPLE AOS/VS PROGRAM CALL_TQKILL
        INTEGER*4 IER
C       ...
C       KILL THE CALLING (I.E., THE CURRENT = THIS) TASK
        CALL TQKILL(IER)
        PRINT 10, IER
   10   FORMAT (' ERROR CODE RETURNED FROM TQKILL IS ', O6, 'K')
C       ...
        END
```

TQMYTID

Get the priority and ID of the calling (current) task.

Format

CALL TQMYTID(taskid, priority, ier)

Arguments

taskid — is an INTEGER*4 variable or array element that receives the ID of the calling (i.e., the current) task.

priority — is an INTEGER*4 variable or array element that receives the priority of the calling (i.e., the current) task.

ier — is an INTEGER*4 variable or array element that receives the result code.

Example

```
C       SAMPLE F77 PROGRAM CALL_TQMYTID
        INTEGER*4 TASK_ID, PRIORITY, IER
C       ...
C       OBTAIN AND PRINT THE ID AND PRIORITY OF THE CURRENT TASK.
        CALL TQMYTID(TASK_ID, PRIORITY, IER)
        PRINT 10, TASK_ID, PRIORITY, IER
   10   FORMAT (' ID OF CURRENT TASK IS:       ', I6, /,
     1          ' PRIORITY OF CURRENT TASK IS: ', I6, /,
     2          ' ERROR CODE FROM TQMYTID IS:  ', O6, 'K')
C       ...
        END
```

TQPRI

Change the priority of the calling (current) task.

Format

CALL TQPRI(priority, ier)

Arguments

priority   is an INTEGER*4 expression that specifies the new priority of the calling (i.e., the current) task.

ier   is an INTEGER*4 variable or array element that receives the result code.

Example

```
C       SAMPLE PROGRAM CALL_TQPRI TO CHANGE THE PRIORITY OF
C          THE CURRENT TASK
        INTEGER*4 NEW_PRIORITY, IER
C       ...
        NEW_PRIORITY = 5
        CALL TQPRI(NEW_PRIORITY, IER)
        PRINT 10, IER
   10   FORMAT (' ERROR CODE RETURNED FROM TQPRI IS ', O6, 'K')
C       ...
        END
```

TQPRKIL

Kill all tasks of a specified priority.

Format

CALL TQPRKIL(priority, ier)

Arguments

priority is an INTEGER*4 expression that specifies the priority of the tasks to be killed.

ier is an INTEGER*4 variable or array element that receives the result code.

Example

```
C       SAMPLE AOS/VS PROGRAM CALL_TQPRKIL
        INTEGER*4 PRIORITY_7 /7/, IER
C       ...
C       KILL ALL TASKS WHOSE PRIORITY IS THE VALUE OF PRIORITY_7.
        CALL TQPRKIL(PRIORITY_7, IER)
        PRINT 10, IER
   10   FORMAT (' ERROR CODE RETURNED FROM TQPRKIL IS ', O6, 'K')
C       ...
        END
```

TQPROT

Start a protected area.

Format

CALL TQPROT(ier)

Argument

ier is an INTEGER*4 variable or array element that receives the result code.

Explanation

This routine has no direct counterpart in AOS/VS.

When a task (we'll label it A) successfully returns from this routine, no other task (labeled B) can suspend (TQIDSUS, TQPRSUS) or kill (TQIDKIL, TQPRKIL) task A until two events occur:

- Task A successfully returns from a matching TQUNPROT (exit a protected path) routine.
- Task A has no other levels of protection because of previous calls to TQPROT.

Any such task B becomes suspended until A successfully executes all necessary calls to TQUNPROT; then B's request is processed, and A becomes suspended or killed. If two or more tasks try to suspend or kill A while it is protected, the one that eventually kills or suspends A is undefined.

F77 assigns each task a *protect count* field whose value at initiation is zero. CALLing TQPROT increments a task's protect count by one. CALLing TQUNPROT decrements a task's protect count by one (unless it's already zero). Thus, a task is protected if, and only if, its protect count is greater than zero.

Example

```
C       SAMPLE F77 PROGRAM CALL_TQPROT
        INTEGER*4 IER1, IER2
C       ...
        CALL TQPROT(IER1)
C               AS LONG AS IER1=0, I CAN'T BE SUSPENDED OR KILLED BY ANY
C               OTHER TASK; IF ONE TRIES, IT BECOMES SUSPENDED UNTIL
C               AT LEAST I'M FINISHED AND CALL TQUNPROT.
                PRINT 10, IER1
   10           FORMAT (' ERROR CODE RETURNED FROM TQPROT IS ',
     1                  O6, 'K')
C               ...
C               I'VE COMPLETED MY PROTECTED PATH.
        CALL TQUNPROT(IER2)
        PRINT 20, IER2
   20   FORMAT (' ERROR CODE RETURNED FROM TQUNPROT IS ', O6, 'K')
C       ...
        END
```

TQPRRDY

Ready all tasks of a specified priority.

Format

CALL TQPRRDY(priority, ier)

Arguments

priority — is an INTEGER*4 expression that specifies the priority of the tasks to be made ready.

ier — is an INTEGER*4 variable or array element that receives the result code.

Example

```
C       SAMPLE AOS/VS PROGRAM CALL_TQPRRDY
        INTEGER*4 IER
C       ...
C       MAKE READY ANY TASK WHOSE PRIORITY NUMBER IS 8.
        CALL TQPRRDY(8, IER)
        PRINT 10, IER
   10   FORMAT (' ERROR CODE RETURNED FROM TQPRRDY IS ', 06, 'K')
C       ...
        END
```

TQPRSUS

Suspend all tasks of a specified priority.

Format

CALL TQPRSUS(priority, ier)

Arguments

priority — is an INTEGER*4 expression that specifies the priority of the tasks to be suspended.

ier — is an INTEGER*4 variable or array element that receives the result code.

Example

```
C       SAMPLE F77 PROGRAM CALL_TQPRSUS
        INTEGER*4 PRIORITY_5, IER
C       ...
C       SUSPEND ANY TASK WHOSE PRIORITY NUMBER IS 5.
        PRIORITY_5  = 5
        CALL TQPRSUS(PRIORITY_5, IER)
        PRINT 10, IER
   10   FORMAT (' ERROR CODE RETURNED FROM TQPRSUS IS ', O6, 'K')
C       ...
        END
```

TQQTASK

Create a queued task.

Format

CALL TQQTASK(subroutine, task_definition_packet, ier)

Arguments

subroutine	is the name of the subroutine you are placing on a queue for execution. Declare it EXTERNAL.
task_definition_packet	is an INTEGER*2 (*not* INTEGER*4) array that contains the task definition packet. Don't alter this array while it is in the task queue. You can alter it after a corresponding execution of TQDQTSK.
ier	is an INTEGER*4 variable or array element that receives the result code.

Explanation

This routine assumes you have built task_definition_packet according to the operating system programmer's manual. However, this routine (and its complement, TQDQTSK) will restrict or overwrite the following words in the parameter packet:

- ?DID (task ID) cannot be zero. And, every task must have a unique ID number.
- ?DAC2/?DCL2 is replaced by F77's own value.
- ?DSTB/?DSTL (stack base), if zero or negative, is replaced by F77's own value. Otherwise, F77 uses any positive number as the address of the stack base. Then, you must declare an array of length ?DNUM*?DSSZ/?DSSL and use the WORDADDR function to place the address of this array in ?DSTB/?DSTL.
- ?DSFLT (stack fault handler) is replaced by F77's own value.
- If you set ?DSSZ/?DSSL (stack size) to zero, F77 provides a default size. Any number you specify should be at least 1200 words. This number may change; see your Release Notice.
- ?DFLGS is replaced by F77's own value.

Example

Read the sample program CALL_TQDQTSK that is part of the explanation of the TQDQTSK subroutine. This program shows one way to set up a task definition packet for the TQQTASK subroutine.

TQREC

Receive an intertask message.

Format

CALL TQREC(mailbox, message, ier)

Arguments

mailbox — is an INTEGER*4 variable or array element that specifies the word from which you will receive a message from another task. mailbox must be in a named COMMON area shared by both this and the other task.

message — is an INTEGER*4 variable or array element that contains a nonzero message; this message arrives from the previous argument, mailbox.

ier — is an INTEGER*4 variable or array element that receives the result code.

Example

```
C       SAMPLE F77 PROGRAM CALL_TQREC
        INTEGER*4 MAILBOX, MESSAGE, IER
        COMMON /COLD/ MAILBOX
C       ...
C       WAIT FOR AOS/VS TO PLACE A TWO-WORD MESSAGE IN VARIABLE
C          MAILBOX, TO MOVE THE CONTENTS OF MAILBOX TO VARIABLE
C          MESSAGE, AND TO PLACE A ZERO IN VARIABLE MAILBOX.
        CALL TQREC (MAILBOX, MESSAGE, IER)
        PRINT 10, MESSAGE, IER
   10   FORMAT (' MESSAGE RECEIVED IS: ', 012, 'K', /,
     1          ' ERROR CODE VALUE IS: ', 06 , 'K')
C       ...
        END
```

TQRECNW

Receive an intertask message without waiting.

Format

CALL TQRECNW(mailbox, message, ier)

Arguments

mailbox — is an INTEGER*4 variable or array element that specifies the word from which you will receive a message from another task. mailbox must be in a named COMMON area shared by both this and the other task.

message — is an INTEGER*4 variable or array element that contains a nonzero message; this message arrives from the previous argument, mailbox.

ier — is an INTEGER*4 variable or array element that receives the result code.

Example

```
C       SAMPLE F77 PROGRAM CALL_TQRECNW
        INTEGER*4 MAILBOX, MESSAGE, IER
        COMMON /COLD/ MAILBOX
C       ...
C       SEE IF AOS/VS HAS PLACED A TWO-WORD MESSAGE IN VARIABLE
C          MAILBOX, MOVED THE CONTENTS OF MAILBOX TO VARIABLE
C          MESSAGE, AND PLACED A ZERO IN VARIABLE MAILBOX; THEN
C          DISPLAY THE FINDING.
        MESSAGE = 0     ! INITIAL ASSUMPTION: NO MAIL FOR ME
        CALL TQRECNW (MAILBOX, MESSAGE, IER)
        IF ( MESSAGE .EQ. 0 ) THEN
                PRINT *, 'NO MESSAGE RECEIVED'
        ELSE
                PRINT 10, MESSAGE
   10           FORMAT (' MESSAGE RECEIVED IS: ', O12, 'K')
        ENDIF
C       ...
        END
```

TQSTASK

Initiate one task.

Format

CALL TQSTASK(subroutine, taskid, priority, stacksize, ier)

Arguments

subroutine — is the name of the subroutine you want to initiate. Declare it EXTERNAL.

taskid — is an INTEGER*4 expression that contains the task's ID number.

priority — is an INTEGER*4 expression between 0 and 255, inclusive, which specifies the task's priority.

stacksize — is an INTEGER*4 expression that specifies the size of your stack in words. You can specify zero, and F77 will handle the stack size for you. If you specify stacksize, it should be at least 1200. This number may change; see your Release Notice.

ier — is an INTEGER*4 variable or array element that receives the result code.

Example

```
C       SAMPLE AOS/VS PROGRAM CALL_TQSTASK
        INTEGER*4 IER
        EXTERNAL SUB_14
C       ...
C       START THE TASK IN SUBROUTINE "SUB_14" WHOSE ID IS 14, WHOSE
C          PRIORITY NUMBER IS 18, AND WHOSE STACK SIZE IS SELECTED BY F77.
        CALL TQSTASK (SUB_14, 14, 18, 0, IER)
        PRINT 10, IER
   10   FORMAT (' ERROR CODE RETURNED FROM TQSTASK IS ', O6, 'K')
C       ...
        END
```

TQSUS

Suspend the calling (current) task.

Format

CALL TQSUS(ier)

Argument

ier is an INTEGER*4 variable or array element that receives the result code.

Example

```
C       SAMPLE AOS/VS PROGRAM CALL_TQSUS
        INTEGER*4 IER
C       ...
C       SUSPEND THE CALLING (I.E., THE CURRENT = THIS) TASK
        CALL TQSUS (IER)
        PRINT 10, IER
   10   FORMAT (' ERROR CODE RETURNED FROM TQSUS IS ', 06, 'K')
C       ...
        END
```

TQUNPROT

Exit a protected area.

Format

CALL TQUNPROT(ier)

Argument

ier is an INTEGER*4 variable or array element that receives the result code.

Explanation

This routine has no direct counterpart in AOS/VS.

Any protected path in a task begins with a call to the TQPROT routine and ends with a call to the TQUNPROT routine. See the explanation of TQPROT for more information about TQUNPROT and how these two calls affect a task's protect count field.

Example

See the sample program CALL_TQPROT under the explanation of subroutine TQPROT.

TQXMT

Transmit an intertask message.

Format

CALL TQXMT(mailbox, message, flag, ier)

Arguments

mailbox — is an INTEGER*4 variable or array element that specifies the word into which you will place a message for transmission to another task or tasks. You must place mailbox in a named COMMON area shared by the receiving task or tasks, and mailbox must contain zero before the call.

message — is an INTEGER*4 expression that contains a nonzero message; this message goes to the previous argument, mailbox.

flag — is an INTEGER*4 expression whose values and corresponding directions are

-1	Transmit the message to all waiting receiving tasks.
Not -1	Transmit the message to only the waiting receiving task with the highest priority.

ier — is an INTEGER*4 variable or array element that receives the result code.

Example

```
C       SAMPLE F77 PROGRAM CALL_TQXMT
        INTEGER*4 MAILBOX, IER
        COMMON /COLD/ MAILBOX
C       ...
C       SEND THE "MESSAGE" 377K TO VARIABLE MAILBOX AND THEN FROM
C          THERE TO ALL AWAITING TASKS REGARDLESS OF THEIR PRIORITIES.
        MAILBOX = 0
        CALL TQXMT (MAILBOX, 377K, -1, IER)
        PRINT 10, IER
   10   FORMAT (' ERROR CODE RETURNED FROM TQXMT IS ', O6, 'K')
C       ...
        END
```

TQXMTW

Transmit an intertask message and wait for its reception.

Format

CALL TQXMTW(mailbox, message, flag, ier)

Arguments

mailbox — is an INTEGER*4 variable or array element that specifies the word into which you will place a message for transmission to another task or tasks. You must place mailbox in a named COMMON area shared by the receiving task or tasks, and mailbox must contain zero before the call.

message — is an INTEGER*4 expression that contains a nonzero message; this message goes to the previous argument, mailbox.

flag — is an INTEGER*4 expression whose values and corresponding directions are

- -1 — Transmit the message to all waiting receiving tasks.
- Not -1 — Transmit the message to only the waiting receiving task with the highest priority.

ier — is an INTEGER*4 variable or array element that receives the result code.

Example

```
C       SAMPLE F77 PROGRAM CALL_TQXMTW
        INTEGER*4 MAILBOX, IER
        COMMON /COLD/ MAILBOX
C       ...
C       SEND THE "MESSAGE" 377K TO VARIABLE MAILBOX AND THEN FROM
C          THERE TO ONLY THE TASK WITH THE HIGHEST POSSIBLE PRIORITY.
        MAILBOX = 0
        CALL TQXMTW (MAILBOX, 377K, 1, IER)
        PRINT 10, IER
   10   FORMAT (' ERROR CODE RETURNED FROM TQXMTW IS ', O6, 'K')
C       ...
        END
```

Another Sample Multitasking Program

We have created a sample multitasking program with its program units TASK0, TASK11, TASK12, TASK13, TASK14, and TASK15. At runtime:

- TASK0 initiates TASK11, TASK12, TASK13, TASK14, and TASK15; it also opens a fresh file, TASK0.OUT, to receive some of the tasks' output.
- TASK11 writes a message into TASK0.OUT every 5 seconds.
- TASK12 repeatedly creates, writes into, and deletes file TASK12.OUT.
- TASK13 accepts 10 integers into array IARRAY from the console.
- TASK14 sorts the elements of IARRAY into ascending order.
- TASK15 displays IARRAY and kills TASK11, TASK12, TASK13, TASK14, and itself.

Listings of TASK0, TASK11, TASK12, TASK13, TASK14, and TASK15 appear in respective Figures 4-12, 4-13, 4-14, 4-15, 4-16, and 4-17.

```
Source file: TASKO.F77
Compiled on 1-Dec-82 at 16:40:13 by AOS/VS F77 Rev 02.10.00.00
Options: F77/L=TASKO.LS

          PROGRAM TASKO          ! MAIN PROGRAM TO INITIALIZE TASKS
C                                       TASK11, TASK12, TASK13, TASK14,
C                                       AND TASK15.

          EXTERNAL TASK11, TASK12, TASK13, TASK14, TASK15

          DIMENSION ITIME(3)

          COMMON /COLD/ MAIL34, MAIL45, IARRAY(10) ! FOR TASK13 -> TASK14
C                           COMMUNICATION, TASK14 -> TASK 15 COMMUNICATION,
C                           AND THE ARRAY TO BE OBTAINED, SORTED, AND PRINTED.
          MAIL34 = 0
          MAIL45 = 0

C         ALL OUTPUT GOES TO FRESH FILE <TASKO.OUT>.
          OPEN (1, FILE='TASKO.OUT', STATUS = 'FRESH',
     1          RECFM='DATASENSITIVE', CARRIAGECONTROL='LIST')
          CALL TIME(ITIME)
          WRITE (1, 10) ITIME
   10     FORMAT ('IN FILE TASKO.OUT: TASKO  HAS BEGUN AT ',
     1          I2, ':', I2, ':', I2, '<NL>')

C         INITIATE THE TASKS VIA SUBROUTINE <TQSTASK> BY GIVING AS
C            ARGUMENTS EACH TASKS'S NAME, ID NUMBER, PRIORITY,
C            AND SYSTEM-SELECTED STACK SIZE.

          CALL TQSTASK (TASK11, 11, 7, 0, IER)
          IF ( IER .NE. 0 ) THEN
                  PRINT *, 'ERROR ', IER, ' OCCURRED IN TASKO WHILE ',
     1                     'INITIATING TASK11'
                  STOP '-- PROGRAM ENDS NOW'
          ENDIF

          CALL TQSTASK (TASK12, 12, 7, 0, IER)
          IF ( IER .NE. 0 ) THEN
                  PRINT *, 'ERROR ', IER, ' OCCURRED IN TASKO WHILE ',
     1                     'INITIATING TASK12'
                  STOP '-- PROGRAM ENDS NOW'
          ENDIF

          CALL TQSTASK (TASK13, 13, 7, 0, IER)
          IF ( IER .NE. 0 ) THEN
                  PRINT *, 'ERROR ', IER, ' OCCURRED IN TASKO WHILE ',
     1                     'INITIATING TASK13'
                  STOP '-- PROGRAM ENDS NOW'
          ENDIF

```

DG-25227

Figure 4-12. Listing of Program TASKO.F77 (continues)

```
      CALL TQSTASK (TASK14, 14, 7, 0, IER)
      IF ( IER .NE. 0 ) THEN
              PRINT *, 'ERROR ', IER, ' OCCURRED IN TASKO WHILE ',
     1                 'INITIATING TASK14'
              STOP '-- PROGRAM ENDS NOW'
      ENDIF

      CALL TQSTASK (TASK15, 15, 7, 0, IER)
      IF ( IER .NE. 0 ) THEN
              PRINT *, 'ERROR ', IER, ' OCCURRED IN TASKO WHILE ',
     1                 'INITIATING TASK15'
              STOP '-- PROGRAM ENDS NOW'
      ENDIF

C     I'M DONE.
      PRINT *, 'TASKO IS DYING'
      CALL TQKILL (IER)
      IF ( IER .NE. 0 ) THEN
              PRINT *, 'ERROR ', IER, ' OCCURRED IN TASKO WHILE ',
     1                 'KILLING (TQKILL) TASKO'
              STOP '-- PROGRAM ENDS NOW'
      ENDIF

      END
```

DG-25227

Figure 4-12. Listing of Program TASK0.F77 (concluded)

```
Source file: TASK11.F77
Compiled on 1-Dec-82 at 16:40:45 by AOS/VS F77 Rev 02.10.00.00
Options: F77/L=TASK11.LS

          SUBROUTINE TASK11

C         THIS TASK WRITES A MESSAGE INTO FILE <TASKO.OUT> EVERY 5
C            SECONDS.  <TASKO.OUT> IS OPENED BY MAIN PROGRAM <TASKO>.

          COMMON /COLD/ MAIL34, MAIL45, IARRAY(10)
          DIMENSION ITIME(3)

%INCLUDE 'TASK11_SYMBOLS.F77.IN'    ! FOR ?WDELAY SYSTEM CALL
****  F77 INCLUDE file for system parameters ****

****   INTEGER*4 parameters for SYSID  ****


          INTEGER*4  ISYS_WDELAY
          PARAMETER (ISYS_WDELAY = 179)   ! ?.WDELAY = 263K

****      Parameters for PARU   ****



****  END of F77 INCLUDE file for system parameters ****

          CALL TIME(ITIME)
          WRITE (1, 10) ITIME
     10   FORMAT ('IN FILE TASKO.OUT: TASK11 HAS BEGUN AT ',
       1          I2, ':', I2, ':', I2, '<NL>')

     20   IAC0 = 5000     ! SPECIFY A DELAY OF 5000 MILLISECONDS
          IAC1 = 0
          IAC2 = 0
C         DELAY (SUSPEND) THIS TASK FOR 5 SECONDS.
          IER = ISYS(ISYS_WDELAY, IAC0, IAC1, IAC2)
          IF ( IER .NE. 0 ) THEN
                  PRINT *, 'ERROR ', IER, ' OCCURRED IN TASK11 DURING ',
       1                   'A ?WDELAY SYSTEM CALL'
                  STOP '-- PROGRAM ENDS NOW'
          ENDIF

          CALL TIME (ITIME)
          WRITE (1, 30) ITIME
     30   FORMAT ('TASK11 REPORTS AFTER A  5-SECOND DELAY AT ',
       1                  I2, ':', I2, ':', I2)
          GO TO 20

          END
```

DG-25228

Figure 4-13. A Listing of Subroutine TASK11.F77

```
Source file: TASK12.F77
Compiled on 1-Dec-82 at 16:41:06 by AOS/VS F77 Rev 02.10.00.00
Options: F77/L=TASK12.LS

          SUBROUTINE TASK12

C
C         THIS TASK REPEATEDLY CREATES AND DELETES A FILE, THUS
C            PERFORMING MANY SYSTEM CALLS.  THE FILE, NAMED
C            "TASK12.OUT", CONTAINS A TABLE OF SINES AND COSINES.
C

          COMMON /COLD/ MAIL34, MAIL45, IARRAY(10)


          DO 40 I = 1, 32000

                  OPEN (2, FILE='TASK12.OUT',     STATUS='FRESH',
     +                     RECFM='DATASENSITIVE', CARRIAGECONTROL='LIST')

                  WRITE (2, 10) I
   10             FORMAT ('X', 10X, 'SINE', 10X, 'COSINE', 10X,
     +                    '(I IS NOW ', I5, ')', /)

                  DO 30 J = 1, 40
                          XJ = FLOAT(J)/10  ! XJ = .1, .2, ..., 4.0
                          WRITE (2, 20) XJ, SIN(XJ), COS(XJ)
   20                     FORMAT (F3.1, 7X, F7.4, 7X, F7.4)
   30             CONTINUE

                  CLOSE(2)

   40     CONTINUE

          RETURN  ! KILL THIS TASK (BUT -- IT'S UNLIKELY THAT
C                         THIS STATEMENT WILL EXECUTE.)
          END
```

DG-25229

Figure 4-14. A Listing of Subroutine TASK12.F77

```
Source file: TASK13.F77
Compiled on 1-Dec-82 at 16:41:24 by AOS/VS F77 Rev 02.10.00.00
Options: F77/L=TASK13.LS

      SUBROUTINE TASK13

C     THIS TASK ACCEPTS INTO <IARRAY> 10 INTEGERS FROM THE CONSOLE
C        AND THEN SENDS A MESSAGE TO <TASK14>.

      COMMON /COLD/ MAIL34, MAIL45, IARRAY(10)

      PRINT *
      PRINT *, 'GIVE ME 10 INTEGERS'
      PRINT *

      DO 10 I = 1, 10
      PRINT *, 'INTEGER NUMBER ', I, ' ?  '
      READ *, IARRAY(I)
   10 CONTINUE

      PRINT *

C     NOTIFY <TASK14> THAT I'M DONE SO IT CAN SORT <IARRAY>.
C        I'LL SEND IT THE NUMBER   3   AS THE MESSAGE.

      CALL TQXMT (MAIL34, 3, -1, IER)

      IF ( IER .NE. 0 ) THEN
              PRINT *, 'ERROR ', IER, ' OCCURRED IN TASK13 DURING ',
     1                 'A CALL TO   TQXMT   .'
              STOP '-- PROGRAM ENDS NOW'
      ENDIF

      END
```

DG-25230

Figure 4-15. A Listing of Subroutine TASK13.F77

```
Source file: TASK14.F77
Compiled on 1-Dec-82 at 16:42:01 by AOS/VS F77 Rev 02.10.00.00
Options: F77/L=TASK14.LS

      SUBROUTINE TASK14

C     THIS TASK AWAITS THE RECEIPT OF THE MESSAGE WHOSE VALUE IS 3
C        FROM <TASK13>.  THEN, IT SORTS THE ELEMENTS OF <IARRAY>
C        INTO ASCENDING ORDER AND FINISHES BY SENDING A MESSAGE TO
C        <TASK14>.

      COMMON /COLD/ MAIL34, MAIL45, IARRAY(10)

   10 CALL TQREC (MAIL34, MESSAGE, IER)
      IF ( IER .NE. 0 ) THEN
              PRINT *, 'ERROR ', IER, ' OCCURRED IN TASK14 DURING ',
     1                 'A CALL TO   TQREC    '
              STOP '-- PROGRAM ENDS NOW'
      ENDIF

C     <MESSAGE> MUST BE 3; WAIT SOME MORE IF IT ISN'T
      IF ( MESSAGE .EQ. 3 ) GO TO 20
      GO TO 10        ! <MESSAGE> DOES NOT CONTAIN 3.

   20 CONTINUE        ! <MESSAGE> DOES CONTAIN 3.
   30 KSWAP = 0       ! COUNT OF SWAPS FOR THE NEXT PASS THROUGH <IARRAY>

      DO 40 I = 1, 9
      IF ( IARRAY(I) .LE. IARRAY(I+1) ) GO TO 40
C     SWAP THE CONTENTS OF THE CURRENT TWO <IARRAY> ELEMENTS.
      ITEMP = IARRAY(I)
      IARRAY(I) = IARRAY(I+1)
      IARRAY(I+1) = ITEMP
      KSWAP = KSWAP + 1   ! COUNT THIS SWAP
   40 CONTINUE
      IF ( KSWAP .GE. 1 ) GO TO 30 ! <IARRAY> MIGHT NOT BE SORTED YET

C     <IARRAY> IS SORTED NOW, SO SEND A MESSAGE WHOSE VALUE IS 4
C        TO <TASK15>.

      CALL TQXMT (MAIL45, 4, -1, IER)
      IF ( IER .NE. 0 ) THEN
              PRINT *, 'ERROR ', IER, ' OCCURRED IN TASK14 DURING ',
     1                 'A CALL TO   TQXMT    '
              STOP '-- PROGRAM ENDS NOW'
      ENDIF

      END
```

DG-25231

Figure 4-16. A Listing of Subroutine TASK14.F77

```
Source file: TASK15.F77
Compiled on 1-Dec-82 at 16:42:33 by AOS/VS F77 Rev 02.10.00.00
Options: F77/L=TASK15.LS

      SUBROUTINE TASK15

C     THIS TASK AWAITS THE RECEIPT OF THE MESSAGE WHOSE VALUE IS 4
C        FROM <TASK14>.  THEN, IT DISPLAYS THE SORTED ELEMENTS OF
C        <IARRAY> AND SEQUENTIALLY KILLS ALL ACTIVE TASKS, INCLUDING
C        ITSELF.

      COMMON /COLD/ MAIL34, MAIL45, IARRAY(10)

   10 CALL TQREC (MAIL45, MESSAGE, IER)
      IF ( IER .NE. 0 ) THEN
              PRINT *, 'ERROR ', IER, ' OCCURRED IN TASK15 DURING ',
     1                 'A CALL TO   TQREC    '
              STOP '-- PROGRAM ENDS NOW'
      ENDIF

C     <MESSAGE> MUST BE 4; WAIT SOME MORE IF IT ISN'T
      IF ( MESSAGE .EQ. 4 ) GO TO 20
      GO TO 10         ! <MESSAGE> DOES NOT CONTAIN 4.

   20 CONTINUE         ! <MESSAGE> DOES CONTAIN 4.

      DO 30 I = 1, 10
      PRINT *, I, '<TAB>', IARRAY(I)
   30 CONTINUE

      WRITE (1, 40)    ! CLEAN-UP MESSAGE
   40 FORMAT ('<NL>*** TASK 15 REPORTS: THIS IS THE LAST RECORD ***<NL>')

C     KILL THE OTHER TASKS AND THEN MYSELF.
      PRINT *
      PRINT *, 'TASK15 IS ABOUT TO KILL ALL OTHER TASKS AND THEN ITSELF'
      PRINT *

      DO 50 I = 11, 14
      CALL TQIDKIL(I, IER)
      IF ( IER .NE. 0 ) THEN
              PRINT *
              PRINT *, 'ERROR ', IER, ' OCCURRED IN TASK15 DURING ',
     1                 'A CALL TO   TQIDKIL    '
              PRINT *, 'THE ID OF THE TASK   TQIDKIL   FAILED ON IS ', I
              PRINT *
      ENDIF
   50 CONTINUE

      CALL EXIT  ! THIS TASK (THE LAST ACTIVE ONE) NOW KILLS ITSELF,
C                  AND THE ENTIRE PROCESS TERMINATES.

      END
```

DG-25232

Figure 4-17. A Listing of Subroutine TASK15.F77

The commands

```
F77 (TASK0 TASK11 TASK12 TASK13 TASK14 TASK15)
F77LINK/TASKS=6 TASK0 TASK11 TASK12 TASK13 TASK14 TASK15
```

create TASK0.PR. F77LINK.CLI by default includes its /IOCONFLICT=QUEUE switch and value, so there is no possibility of an I/O conflict problem with file TASK0.OUT at runtime.

The results of a typical execution of TASK0.PR are next.

) X TASK0 ⤶

TASK0 IS DYING

GIVE ME 10 INTEGERS

INTEGER NUMBER 1 ? 85 ⤶

INTEGER NUMBER 2 ? 941 ⤶

INTEGER NUMBER 3 ? -17 ⤶

INTEGER NUMBER 4 ? 40 ⤶

INTEGER NUMBER 5 ? 129 ⤶

INTEGER NUMBER 6 ? -3 ⤶

INTEGER NUMBER 7 ? 178 ⤶

INTEGER NUMBER 8 ? 58 ⤶

INTEGER NUMBER 9 ? 0 ⤶

INTEGER NUMBER 10 ? 9 ⤶

```
1     -17
2     -3
3     0
4     9
5     40
6     58
7     85
8     129
9     178
10    941
```

TASK15 IS ABOUT TO KILL ALL OTHER TASKS AND THEN ITSELF

ERROR 12 OCCURRED IN TASK15 DURING A CALL TO TQIDKIL
THE ID OF THE TASK TQIDKIL FAILED ON IS 13

ERROR 12 OCCURRED IN TASK15 DURING A CALL TO TQIDKIL
THE ID OF THE TASK TQIDKIL FAILED ON IS 14

) TYPE TASK0.OUT ⤶

IN FILE TASK0.OUT: TASK0 HAS BEGUN AT 16:51:44

IN FILE TASK0.OUT: TASK11 HAS BEGUN AT 16:51:46

TASK11 REPORTS AFTER A 5-SECOND DELAY AT 16:51:51
TASK11 REPORTS AFTER A 5-SECOND DELAY AT 16:51:56
TASK11 REPORTS AFTER A 5-SECOND DELAY AT 16:52: 1

*** TASK 15 REPORTS: THIS IS THE LAST RECORD ***

You may ask about the display of the two error 12's, "TASK I.D. ERROR" (from the symbol ERTID in PARU.SR), when TASK15 issues a TQIDKIL call to tasks with ID numbers 13 and 14. Why?

TASK13 and TASK14 are inactive at this time. They have executed all their statements, and thus the task scheduler has already killed them. An attempt by TQIDKIL to kill a task that is inactive results in an ERTID error.

Furthermore, I varies from 11 to 14 (instead of from 11 to 15) in the DO 50 loop of TASK15. In other words, the CALL EXIT statement kills TASK15 instead of the TQIDKIL subroutine in the DO 50 loop. Why?

When I = 15, the IDKIL call results in a fatal error message. If we changed the terminal value of I in the DO 50 loop of TASK15.F77 from 14 to 15 and deleted the CALL EXIT statement, then execution of the resulting TASK0.PR program would display the following:

```
...
...
...
ERROR 12 OCCURRED IN TASK15 DURING A CALL TO    TQIDKIL
THE ID OF THE TASK    TQIDKIL    FAILED ON IS 14

*ABORT*
LAST TASK WAS KILLED
ERROR: FROM PROGRAM
X,TASK0
```

End of Chapter

Chapter 5
Debugging

Programmers commonly use the word *debug* to describe the process of locating and eliminating errors from their programs. A *bug* is simply an error.

This chapter explains possible errors in terms of their symptoms, their causes, and finding those causes. The resulting changes to your programs, F77 commands, F77LINK commands, and program execution commands are then largely your responsibility. This chapter now proceeds with the following sections:

- Traditional Debugging Methods
- The SWAT Debugger
- Avoid Errors BEFORE Coding
- Data General Bugs?

Traditional Debugging Methods

Typically, you begin the process of eliminating bugs when you first see a symptom. Symptoms include:

- Compiler error messages (i.e., from F77.CLI).
- Link error messages (i.e., from F77LINK.CLI).
- Abnormal program termination at runtime.
- Incorrect output at runtime.

It's natural to ask "What about doing something to eliminate errors *before* beginning to write F77 statements?" We address this later in the "Avoid Errors BEFORE Coding" section of this chapter. But first, we'll discuss how to detect errors *after* they occur.

The F77 compiler, Link, and the runtime routines report errors they find in your instructions and in data the instructions process. The error messages summarize the problem. You correct it based on the error messages, your knowledge of F77, and F77 documentation.

Data General F77 does not have a TRACE option to print the values of variables that the program assigns as it proceeds. Instead, you can follow these traditional steps:

- Insert extra PRINT (or WRITE) statements for key variables at important places.
- Recompile and relink.
- Execute the program and examine the values of the key variables.

- If the examination reveals the cause, then:
  - Make corrections to the source program.
  - Recompile and relink.
  - Execute the program to ensure the elimination of the error.
  - Eliminate the extra PRINT (or WRITE) statements from the source program.
  - Recompile and relink.
- If the examination doesn't reveal the cause, then begin again at the first item in this list.

You can ease this process somewhat by declaring a logical named constant and making the extra output statements depend on that constant. Then, redefinition of that constant will switch modes. For example,

```
      LOGICAL DEBUG
      PARAMETER (DEBUG = .TRUE.)
C     ...
      IF ( DEBUG ) THEN
C             PRINT THE VALUE OF KEY VARIABLES.
      ENDIF
C     ...
      END
```

These steps, while fairly effective, can be quite time consuming. The mechanics of editing the source program modules, compiling, linking, and executing require far more time than the creative aspects of deciding which variables to print, when to print them, and how to interpret them. Is there a better way? Yes — continue reading.

The SWAT Debugger

The SWAT Debugger does not debug in the sense of *removing* errors. However, it is a big help in *finding* errors; then it's up to you to change your program to eliminate the errors.

To use the SWAT debugger, you should read the *SWAT™ Debugger User's Manual* and the Release Notice for the current revision of the documentation. However, a brief explanation of SWAT software fundamentals and a sample SWAT debugging session follow. They will show you the features of the SWAT debugger and should whet your appetite to use it.

Sample Program Modules SORT10.F77 and TEST_SORT10.F77

Subroutine SORT10.F77 contains instructions to sort a character array of up to 100 10-byte elements into alphabetical order. The main program, TEST_SORT10.F77, contains an unsorted character array of 10-byte elements. At runtime, TEST_SORT10 CALLs SORT10 to sort the array, and then the main program displays the sorted array. Following are the first pages of TEST_SORT10.LS and SORT10.LS after the compiler has created them. The respective compilation commands are

```
F77/DEBUG/L=TEST_SORT10.LS TEST_SORT10
F77/DEBUG/L=SORT10.LS SORT10
```

The /DEBUG switch has the compiler generate symbols and code for SWAT.

```
Source file: TEST_SORT10.F77
Compiled on 19-Jul-82 at 15:18:39 by AOS/VS F77 Rev 01.32.00.00
Options: F77/DEBUG/L=TEST_SORT10.LS

      PROGRAM TEST_SORT10           !  TO TEST SUBROUTINE SORT10

      CHARACTER*80 ALL_OF_THE_NAMES  ! ALL THE NAMES, IN ONE CONVENIENT
C                                           AND EASY-TO-CONSTRUCT STRING
      CHARACTER*10 NAMES(8)          ! <NAMES> WILL CONTAIN THE EIGHT
C                                           ELEMENTS THAT <SORT10> WILL
C                                           SORT ALPHABETICALLY.

C     THE NEXT TWO LINES HELP TO CONSTRUCT <ALL_OF_THE_NAMES>.
C00000000111111111122222222223333333333444444444455555555556666666666777
C23456789012345678901234567890123456789012345678901234567890123456789012
      DATA  ALL_OF_THE_NAMES  / 'MIKE      HENRIETTA ENRICO    LISA
     + JEFFREY   BETSY     ALICE     NORMAN    ' /

C     PLACE THE 8 INDIVIDUAL FIRST NAMES INTO <NAMES> FROM THE SINGLE
C        STRING <ALL_OF_THE_NAMES>.
      DO 10 I = 1, 8
            NAMES(I) = ALL_OF_THE_NAMES(10*I-9 : 10*I)  ! EXAMPLE: IF
C                      I = 2, THEN <ALL_OF_THE_NAMES(11:20)> IS
C                      'HENRIETTA ' AND <NAMES(2)> IS ALSO 'HENRIETTA '.
   10 CONTINUE

C     SORT THE NAMES INTO ALPHABETICAL ORDER.
      CALL SORT10 (NAMES, 8)

C     PRINT THE RESULTS.
      WRITE (10, *)
      WRITE (10, *) 'THE SORTED NAMES ARE:'
      WRITE (10, *)
      DO 30 I = 1, 8
            WRITE (10, *) NAMES(I)
   30 CONTINUE

      WRITE (10, *)
      WRITE (10, *) '*** END OF JOB ***'
      STOP
      END
```

```
Source file: SORT10.F77
Compiled on 19-Jul-82 at 15:19:36 by AOS/VS F77 Rev 01.32.00.00
Options: F77/DEBUG/L=SORT10.LS

              SUBROUTINE SORT10 (C_ARRAY, N)

        C     THIS SUBROUTINE SORTS THE FIRST <N> ELEMENTS OF A
        C        CHARACTER*10 ARRAY, <C_ARRAY>, WITH AT MOST 100 ELEMENTS
        C        (EACH 10 BYTES LONG).

        C     SORTING METHOD: TRADITIONAL "BUBBLE" SORT WHICH MOVES THE
        C        HIGHER-VALUED  ELEMENTS (SUCH AS "ZACHARY") TO THE RIGHT IN
        C        THE ARRAY AND THE LOWER-VALUED ELEMENTS (SUCH AS "AMANDA")
        C        TO THE LEFT ELEMENTS OF THE ARRAY.

              CHARACTER*10 C_ARRAY(100)
              CHARACTER*10 TEMP          ! TEMPORARY STORAGE AREA REQUIRED
        C                                     BY THE SORT ROUTINE

              IF ( N .LT. 2 ) GO TO 30   ! NO NEED TO SORT.

              N_LESS_1 = N - 1

        C     HERE WE GO ...

              DO 20 J = 1, N_LESS_1
                    M = N-J

                    DO 10 I = 1, M
                    IF ( C_ARRAY(I)     .LE.
             1           C_ARRAY(I+1) )      GO TO 10

        C           IT'S NECESSARY TO SWAP TWO ADJACENT ELEMENTS OF <C_ARRAY>.
        C           FOR EXAMPLE, <C_ARRAY(2)> MIGHT CONTAIN "EDWARD    " AND
        C           <C_ARRAY(3)> MIGHT CONTAIN "BEVERLY   "; THEN THE NEXT
        C           THREE STATEMENTS EXECUTE TO PERFORM THE SWAP.  AFTER THE
        C           SWAP, <C_ARRAY(2)> WILL CONTAIN "BEVERLY   " AND
        C           <C_ARRAY(3)> WILL CONTAIN "EDWARD    ".

                    TEMP = C_ARRAY(I)
                    C_ARRAY(I) = C_ARRAY(I+1)
                    C_ARRAY(I+1) = TEMP

           10       CONTINUE
           20 CONTINUE

        C     DONE!

           30 RETURN

              END
```

Sample Execution without the SWAT Debugger

The command to create TEST_SORT10.PR so that we can execute it either with or without the SWAT debugger is

```
F77LINK/DEBUG TEST_SORT10 SORT10
```

If we give the CLI command

```
X TEST_SORT10
```

then TEST_SORT10.PR displays the following.

```
THE SORTED NAMES ARE:

            ALICE
            NORMA
ENRICO
EY          BETSY
HENRIETTA
LISA    JEFFR
MIKE
N

*** END OF JOB ***
STOP
```

Obviously, this program has at least one bug that results in the mixing of names. We also observe that the garbled names appear in alphabetical order. For the time being, resist the temptation to search TEST_SORT10.F77 and SORT10.F77 for bugs. Read the following summary of the SWAT debugger, and then you'll see how it can help locate the bug.

SWAT Debugger Fundamentals

The SWAT debugger executes to allow easy tracing of your program. Basically, you select places in your program where you wish to know the values of key variables. You tell the debugger to execute your program and pause at the selected places. There, you have the debugger display the key variables' values. Next, you can terminate program execution and fix the source code or continue to the next selected place.

You need only a subset of SWAT debugger commands to locate the problem in program units TEST_SORT10 and SORT10. The command names, descriptions, and examples are as follows.

Command	Description	Example
BREAKPOINT	Set a place in the program where the SWAT debugger will suspend its execution. You specify a line number from the program unit's compiler-created .LS file. The debugger suspends the program just *before* executing the first machine language instruction that the specified source program instruction resulted in.	BREAKPOINT 10
BYE	Terminate the execution of both the SWAT debugger and the program file, and return to the CLI.	BYE
CLEAR	Remove a breakpoint from a program.	CLEAR 10
CONTINUE	Resume execution at a breakpoint.	CONTINUE
ENVIRON-MENT	Select the program unit, usually for moving from one to the other (such as from the main program to a subroutine to set a breakpoint).	ENVIRONMENT SORT10
LIST	List a range of source program lines on the console. Use of LIST frees you from constant reference to a printed .LS file.	LIST 20, 30
TYPE	Display the value of one or more variables on the console.	TYPE I, ARR(3)
%	If you execute the SWAT debugger with the AUDIT switch, then all text appearing on the console goes into an audit file for later printing. The debugger places lines from you that begin with "%" into the audit file, but it does nothing else with these lines.	% Now display J

Sample Execution with the SWAT Debugger

Instead of giving the CLI command

```
X TEST_SORT10
```

as we did before, type

```
X SWAT/AUDIT TEST_SORT10
```

SWAT.PR executes and creates TEST_SORT10.PR as a son process. Here, all dialog between you and the debugger goes into audit file TEST_SORT10.AU. Records in TEST_SORT10.AU beginning with "> " represent commands you give in response to the SWAT debugger prompt "> ". Records that don't begin with "> " represent the debugger's output. Not including the /AUDIT switch means that the dialog appears on the console only.

Marll is the programmer who has created TEST_SORT10.F77 and SORT10.F77. Following is the dialog he and the SWAT debugger created in TEST_SORT10.AU. The records in TEST_SORT10.AU are numbered to make it easier to refer to them. The SWAT debugger does *not* place such record numbers in the audit (.AU) files it creates.

Marll created an unusually large number of comment lines (the ones beginning with "> %") as he located his error. Read TEST_SORT10.AU very carefully to learn how you can use the SWAT debugger. You might have to refer several times to TEST_SORT10.LS and to SORT10.LS as you read TEST_SORT10.AU.

```


---------------------------------------------------------------
USER PROGRAM TEST_SORT10 SWAT AUDIT ON 07/20/82 AT 13:37:31

AOS/VS SWAT Revision 02.19.00.00 ON 07/20/82 AT 13:37:36
PROGRAM -- :UDD2:F77:MARLL:TEST_SORT10
> %
> % Set a breakpoint to see if <NAMES> receives its elements correctly
> %    from <ALL_OF_THE_NAMES>.
> BREAKPOINT 21
Set at :TEST_SORT10:21
> %
> % Also set a breakpoint just before the CALL to SORT10.
> BREAKPOINT 24
Set at :TEST_SORT10:24
> %
> % Verify the breakpoints.
> LIST 20, 25
20 C                      'HENRIETTA ' AND <NAMES(2)> IS ALSO 'HENRIETTA '.
21B          10 CONTINUE
22
23 C     SORT THE NAMES INTO ALPHABETICAL ORDER.
24B               CALL SORT10 (NAMES, 8)
25
> %
> % Move to subroutine SORT10 and set appropriate breakpoints.
> ENVIRONMENT :SORT10
:SORT10
> BREAKPOINT 22, 36
Set at :SORT10:22
Set at :SORT10:36
> %
> % Verify the breakpoints.
> LIST 22, 36
22B               DO 20 J = 1, N_LESS_1
23              M = N-J
24
25              DO 10 I = 1, M
26              IF ( C_ARRAY(I)     .LE.
27       1           C_ARRAY(I+1) )      GO TO 10
28
29 C            IT'S NECESSARY TO SWAP TWO ADJACENT ELEMENTS OF <C_ARRAY>.
30 C            FOR EXAMPLE, <C_ARRAY(2)> MIGHT CONTAIN "EDWARD    " AND
31 C            <C_ARRAY(3)> MIGHT CONTAIN "BEVERLY   "; THEN THE NEXT
32 C            THREE STATEMENTS EXECUTE TO PERFORM THE SWAP.  AFTER THE
33 C            SWAP, <C_ARRAY(2)> WILL CONTAIN "BEVERLY   " AND
34 C            <C_ARRAY(3)> WILL CONTAIN "EDWARD    ".
35
36B                     TEMP = C_ARRAY(I)
```

```
> %
> % Return to the main program ...
> ENVIRONMENT @MAIN
:TEST_SORT10
> % ... and begin program execution.
> CONTINUE

Breakpoint trap at :TEST_SORT10:21
> %
> % Look at the first few elements of <NAMES> while the program
> %   continues to execute.
> TYPE I, NAMES(I) ; CONTINUE
1
"MIKE      "

Breakpoint trap at :TEST_SORT10:21
> TYPE I, NAMES(I) ; CONTINUE
2
"HENRIETTA "

Breakpoint trap at :TEST_SORT10:21
> TYPE I, NAMES(I) ; CONTINUE
3
"ENRICO    "

Breakpoint trap at :TEST_SORT10:21
> %
> % So far, so good.  Since <NAMES> seems OK, I'll clear this breakpoint
> %   and continue.
> CLEAR 21
Cleared at :TEST_SORT10:21
> CONTINUE

Breakpoint trap at :TEST_SORT10:24
> %
> % Go ahead and let SORT10 execute.
> CONTINUE

Breakpoint trap at :SORT10:22
%
% Now I'm in subroutine SORT10.
> TYPE N, N_LESS_1
8
7
> %
> % OK -- move into the   DO 20   and   DO 10   loops that sort <C_ARRAY>.
> CONTINUE

Breakpoint trap at :SORT10:36
> TYPE J, I, C_ARRAY(I), C_ARRAY(I+1)
1
1
"MIKE      "
"HENRIETTA "
> %
> % OK -- C_ARRAY(1) and C_ARRAY(2) have to swap their values.
```

```
107  > CONTINUE
108
109  Breakpoint trap at :SORT10:36
110  > TYPE J, I, C_ARRAY(I), C_ARRAY(I+1)
111  1
112  2
113  "MIKE      "
114  "ENRICO    "
115  > %
116  > % OK -- C_ARRAY(2) and C_ARRAY(3) have to swap their values.
117  > CONTINUE
118
119  Breakpoint trap at :SORT10:36
120  > TYPE J, I, C_ARRAY(I), C_ARRAY(I+1)
121  1
122  3
123  "MIKE      "
124  "LISA JEFFR"
125  > %
126  > % I've got a problem!  "MIKE      " is a valid name but "LISA JEFFR" is
127  > %    wrong.  Somehow "LISA      " and "JEFFREY   " have been incorrectly
128  > %    mixed together.  Now I'll display all the elements of <C_ARRAY> to
129  > %    see if there are any other such mixtures.
130  > TYPE C_ARRAY(1), C_ARRAY(2), C_ARRAY(3), C_ARRAY(4)
131  "HENRIETTA "
132  "ENRICO    "
133  "MIKE      "
134  "LISA JEFFR"
135  > TYPE C_ARRAY(5), C_ARRAY(6), C_ARRAY(7), C_ARRAY(8)
136  "EY   BETSY"
137  "     ALICE"
138  "     NORMA"
139  "N         "
140  > %
141  > % The last five elements of <C_ARRAY> are wrong.  I'll quit the debugger
142  > %    and take a close look at main program TEST_SORT10, which is the
143  > %    source of <C_ARRAY>.
144  > BYE
145
146  SWAT TERMINATED
```

TEST_SORT10.AU is largely self-explanatory. Pay special attention to the following lines.

7 The SWAT debugger gives the pathname of the program file.

21,24 Marll's instructions in lines 11 and 15 set breakpoints at lines 21 and 24 of TEST_SORT10. LISTing lines 20 through 25 verifies the setting of these breakpoints by showing a "B" next to line numbers 21 and 24.

30 Marll set two breakpoints with one statement.

36,50 Note again the letter "B" to signify a breakpoint next to line numbers 22 and 36 of SORT10.

What is *not* self-explanatory is the bug. Somehow the last five elements of C_ARRAY in SORT10 — which originate from NAMES in TEST_SORT10 — have mixed together. Marll decides to execute the debugger again and look more carefully at NAMES instead of moving to subroutine SORT10. Perhaps he was too hasty with his comments in lines 77 through 80 of TEST_SORT10.AU.

Marll gives the CLI commands

```
DELETE TEST_SORT10.AU
X SWAT/AUDIT TEST_SORT10
```

It's necessary to delete the audit file because SWAT/AUDIT appends to <PROGRAM NAME>.AU instead of deleting and recreating it. Again, the /AUDIT switch isn't necessary, but it lets him have a hardcopy of the dialog for later analysis. The resulting TEST_SORT10.AU that points to the error follows.

```
----------------------------------------------------------------
USER PROGRAM test_sort10 SWAT AUDIT ON 07/26/82 AT 09:10:44

AOS/VS SWAT Revision 02.19.00.00 ON 07/26/82 AT 09:10:47
PROGRAM -- :UDD2:F77:MARLL:TEST_SORT10
> %
> % I'll set a breakpoint where I can display ALL the elements of <NAMES>.
> BREAKPOINT 21
Set at :TEST_SORT10:21
> LIST 15, 21
15 C      PLACE THE 8 INDIVIDUAL FIRST NAMES INTO <NAMES> FROM THE SINGLE
16 C          STRING <ALL_OF_THE_NAMES>.
17        DO 10 I = 1, 8
18                NAMES(I) = ALL_OF_THE_NAMES(10*I-9 : 10*I)  ! EXAMPLE: IF
19 C                             I = 2, THEN <ALL_OF_THE_NAMES(11:20)> IS
20 C                             'HENRIETTA ' AND <NAMES(2)> IS ALSO 'HENRIETTA '.
21B            10 CONTINUE
> %
> % Here we go!
> CONTINUE

Breakpoint trap at :TEST_SORT10:21
> TYPE I, NAMES(I) ; CONTINUE
1
"MIKE      "

Breakpoint trap at :TEST_SORT10:21
> TYPE I, NAMES(I) ; CONTINUE
2
"HENRIETTA "

Breakpoint trap at :TEST_SORT10:21
> TYPE I, NAMES(I) ; CONTINUE
3
"ENRICO    "

Breakpoint trap at :TEST_SORT10:21
> TYPE I, NAMES(I) ; CONTINUE
4
"LISA JEFFR"

Breakpoint trap at :TEST_SORT10:21
> TYPE I, NAMES(I) ; CONTINUE
5
"EY   BETSY"

Breakpoint trap at :TEST_SORT10:21
```

```
> TYPE I, NAMES(I) ; CONTINUE
6
"     ALICE"

Breakpoint trap at :TEST_SORT10:21
> TYPE I, NAMES(I) ; CONTINUE
7
"     NORMA"

Breakpoint trap at :TEST_SORT10:21
> TYPE I, NAMES(I)
8
"N         "
> %
> % The first three elements of <NAMES> are OK and I can't see any
> %    immediate reason for the error (the mixing) in the last five
> %    elements.  I'll investigate by going backwards and LISTing the
> %    CHARACTER string <ALL_OF_THE_NAMES>, from which <NAMES>
> %    obtains its elements.
> LIST 9, 13
9  C     THE NEXT TWO LINES HELP TO CONSTRUCT <ALL_OF_THE_NAMES>.
10 C000000001111111111222222222233333333334444444444555555555566666666667777
11 C23456789012345678901234567890123456789012345678901234567890123456789012
12        DATA  ALL_OF_THE_NAMES  / 'MIKE      HENRIETTA ENRICO    LISA
13      + JEFFREY   BETSY     ALICE     NORMAN    ' /
> %
> % Rather puzzling.  I can see that "LISAbbbbbb" (b = blank) is in
> %    lines 12 and 13.  The first five blanks of "LISAbbbbbb" come
> %    from line 12 and the last blank comes from line 13.  <NAMES(4)>
> %    is "LISAbJEFFR" with just one blank.  It looks like only the
> %    blank in "+ JEFFREY" of line 13 has arrived in the incorrect
> %    <NAMES(4)>.  In other words, the five blanks after "LISA" in
> %    line 12 have disappeared.  What's going on here?  I'm going to
> %    terminate SWAT and think of why the five blanks after "LISA"
> %    in line 12 have disappeared.
> %
> % However, before terminating SWAT I'll display <ALL_OF_THE_NAMES>.
> TYPE ALL_OF_THE_NAMES
"MIKE      HENRIETTA ENRICO    LISA JEFFREY   BETSY     ALICE     NORMAN
 "
> %
> % This display also shows that the first five of the necessary six
> %    blank characters after "LISA" have disappeared.
> BYE

SWAT TERMINATED
```

The key question is "What has happened to the first five of the six blanks in 'LISA□□□□□□' (□ = blank)?" One thing you have to remember about the F77 compiler is that, by default, it reads a line from the source module *and ignores any trailing blanks.* In our case, the last characters of line 12 of TEST_SORT10.F77 were either

LISA□□□□□ <NL>

or

LISA<NL>

The F77 compiler ignored any blanks at the end of line 12 and processed the blank in "+ JEFFREY" of line 13. This ignoring effectively shifted the last four elements of ALL_OF_THE_NAMES left by five spaces. Thus, the DO 10 loop of TEST_SORT10 constructed NAMES with the following contents:

```
MIKE□□□□□□
HENRIETTA□
ENRICO□□□□
LISA□JEFFR
EY□□□BETSY
□□□□□ALICE
□□□□□NORMA
N□□□□□□□□□
```

Even though SORT10 worked correctly with the array it received from TEST_SORT10, the array was wrong in the first place, and thus the sorted displayed output from TEST_SORT10 was wrong. This is a perfect example of GIGO — garbage in, garbage out!

Corrections to Sample Program Modules

How do we correct TEST_SORT10 and SORT10? First, SORT10 is fine; it properly sorts the array it receives. There are at least two ways to correct line 12 of TEST_SORT10.F77:

1. Leave it alone and change the compilation command for TEST_SORT10 from

   F77 TEST_SORT10

   to

   F77/CARDFORMAT TEST_SORT10

   The /CARDFORMAT switch directs the compiler to pad (with blanks) to 72 characters any source program line that is less than 72 characters long. F77 then would combine characters 64 through 72 of line 12 with character 7 of line 13 to form the desired "LISA□□□□□□□".

2. Delete lines 3 and 4 of TEST_SORT10.F77. Then, replace lines 9 through 22 with the following.

```
      DATA NAMES / 'MIKE      ', 'HENRIETTA ', 'ENRICO ',
     +             'LISA      ', 'JEFFREY   ', 'BETSY  ',
     +             'ALICE     ', 'NORMAN    '           /
```

The SWAT Debugger — a Summary

SWAT is a very flexible and powerful programming aid. The key to its use is the effective placing of breakpoints and the displaying of the proper variables and arrays at those breakpoints. There is no convenient formula for this placing and displaying. You'll have to employ a fair amount of trial and error as you learn to use the SWAT debugger.

Avoid Errors BEFORE Coding

The old saying that "an ounce of prevention is worth a pound of cure" applies to FORTRAN 77 programming. You have seen that the SWAT debugger makes debugging much easier than the traditional method of placing extra WRITE statements and then later removing them. Even so, you're better off to follow certain techniques before and during the coding stage. Improving the design of a program often reduces the need for debugging it.

The subject of proper program design and coding is a broad one — far too broad for explanation here. However, we list several books next. Each of them contains many suggestions for creating program units that should reduce the need for later debugging. Data General in no way endorses these books or requires that you read any of them; the list is merely for your convenience. The books' authors and titles are:

- Henry F. Ledgard, "Programming Proverbs for FORTRAN Programmers", Hayden Book Company, Inc., Rochelle Park, New Jersey (1975).
- Brian W. Kernighan and P.J. Plauger, "The Elements of Programming Style", McGraw-Hill Book Company, New York, New York (1974).
- Charles B. Kreitzberg and Ben Shneiderman, "The Elements of FORTRAN Style: Techniques for Effective Programming", Harcourt Brace Jovanovich, Inc., New York, New York (1972).
- Dennie Van Tassel, "Program Style, Design, Efficiency, Debugging, and Testing", Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1974).
- Louis A. Hill, Jr., "Structured Programming in FORTRAN", Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1981).

Mr. Van Tassel's book contains an entire chapter on debugging.

Data General Bugs?

The F77 compiler is a large and complicated program. The runtime libraries are a collection of many subroutines. We honestly state that bugs could exist somewhere among all this software. In fact, several compiler error messages have the form "Possible compiler error .... If this message persists, please submit software trouble report."

Your system manager should let you have access to the Software Release Notice that applies to the revision of FORTRAN 77 you are using. Among other things, the Release Notice tells you about:

- The newest features of F77.
- Problems corrected since the last release of F77.
- Problems remaining in F77 with possible ways to work around them.
- Changes to the F77 documentation, including this manual.
- Using Software Trouble Reports.

In particular, if you suspect you've found an error in the compiler or in the runtime routines, then read the section of the Release Notice about a Software Trouble Report (STR). This section explains how to verify that you really have found a problem in Data General software. It also explains how to use an STR to communicate with Data General about the problem.

End of Chapter

Chapter 6
Subprograms

You FORTRAN 77 programmers often create program files (.PR files) that are a collection of one main program unit and one or more subprograms (subroutine and function). The *FORTRAN 77 Reference Manual* describes how to create such program files when the main program unit and all of its subprograms are written in F77.

You actually have a wide choice in selecting languages for a main program unit and its subprograms. For example, you can write a F77 program unit that calls a subroutine subprogram written in assembly language. And, a COBOL program can call a subroutine written in F77 to perform extensive calculations.

The three major parts of this chapter present:

- The structure of F77/assembly language interfaces.
- An overview of high-level-language/F77 interfaces.
- Examples of specific high-level-language/F77 interfaces, such as a BASIC program and its called F77 subroutine.

F77 and Assembly Language Subprograms

This section assumes you are familiar with assembly language and want to use it to write subprograms for calling from F77. Before reading on, remember that Chapter 3 explains how you can use the ISYS function to access the operating system. Thus, you may have no need to write assembly language subroutines whose sole purpose is to perform an operating system call.

VS/ECS Calling Conventions

The F77 compiler generates code to implement the CALL statement or reference to a function subprogram. This code observes the following three conventions. They are part of the Virtual System / External Calling Sequence (VS/ECS).

1. At runtime, the code works with the *addresses* of the arguments. It pushes them onto the stack in reverse order of their appearance in the argument list. Each such pushed addressed is a 32-bit WORD address for Hollerith constants and for most data types. The exceptions are:
   - The code pushes a 32-bit BYTE address for CHARACTER variables and character constants.
   - The code pushes the WORD address of the first of two 32-bit words for statement labels (e.g., *90). The first of these two words contains the new value of the program counter (PC). That is, the first word contains the address in the compiled code of the first instruction resulting from the statement. The second of the two words contains the frame pointer (FP) value for use with this compiled code.
   - The code pushes the WORD address of arguments declared as EXTERNAL or INTRINSIC. This word contains the Link-resolved address in the program file that satisfies the global reference these declarations have made. When the argument is an executable routine, this word contains the new value of the PC for use when the LCALL instruction accesses the routine.

If any argument is type CHARACTER, then the runtime code places extra arguments on the stack. These extra arguments are called *dope vectors*. They inform the called routine of the actual size of the CHARACTER arguments. Either the compiler or the runtime code builds the dope vectors. The runtime code pushes the addresses of all required dope vectors onto the stack; then it pushes the addresses of all the arguments (non-CHARACTER and CHARACTER) onto the stack as described previously.

NOTE: If you're writing assembly language subprograms for reference from F77 programs, you need to know that these dope vectors exist on the stack. However, Data General determines the count and content of them and they might change over time. Your subprograms should not attempt to refer to or use the dope vectors in any way. Instead, the F77 program unit should use extra arguments to pass length information. Your subprogram can then obtain the length argument via the appropriate argument address and be independent of any dope vector.

2. The LCALL instruction calls EXTERNAL arguments and includes the argument count and a relocated absolute memory reference. This argument count includes any dope vectors and can be greater than the number of arguments the program specified in its CALL statement or function reference.
3. The WRTN instruction will "pop" from the stack all argument addresses that were previously "pushed" there. More specifically, the stack pointer value is restored to its value before step 1 occurred.
4. After the calling program unit regains control:
   - All floating-point accumulators are undefined, except FPAC0 can contain the result of a function subprogram reference that returns a floating-point value. The "Function Subprograms" section of this chapter discusses this reference.
   - All fixed-point accumulators are preserved except AC3, which will contain the frame pointer, and AC0, which could contain the result of a function reference that returns a LOGICAL or INTEGER value.
   - Function results returned in a temporary location will be in the location, the address of which was in AC2 at the time of the reference.

Finally, F77 passes all arguments to subprograms by reference. That is, the subprograms perform operations directly on the arguments, and not on local copies of them.

VS/ECS Return Block

F77 and other AOS/VS languages use the Virtual System / External Calling Sequence (VS/ECS). The VS/ECS *return block* is the fundamental data structure for linkage between routines in the F77 runtime environment. The block is built on the stack of the calling routine. The execution of a CALL statement reaches this linkage for subroutine subprograms; a function reference reaches this linkage for function subprograms. Software constructs the block in two separate steps:

1. The CALLING routine pushes onto the stack the addresses of the arguments it is passing.
2. The CALLED routine, as its first instruction, executes a WSAVS or WSAVR. This instruction pushes a *wide return block* onto the stack. It also allocates the CALLED routine's stack frame, if needed, beyond this wide return block.

The CALLED routine finishes by executing a WRTN instruction. This instruction:

- Pops the CALLED routine's stack frame from the stack.
- Pops the wide return block from the stack.
- Returns to the CALLING routine.

Figure 6-1 contains a general diagram of the VS/ECS return block, and is followed by notes that apply to the different items depicted in the figure. Next, Figures 6-2 and 6-3 further illustrate Figure 6-1 because they contain listings of a specific main program and CALLed subroutine. The subroutine is named TYP_SUB — an abbreviation of "typical subroutine." The main program, since it tests subroutine TYP_SUB, is named TEST_TYP_SUB. These listings, created by the F77 compiler with the "/CODE" switch, confirm the way a subroutine accesses its arguments.

Several notes apply to phrases appearing in Figure 6-1.

Pointer to arg i

This is a word or byte pointer (depending on the data type) that points to argument *i*.

The first argument is always the 12th (decimal) double word off the frame, the second argument is always the 14th, etc.

REMEMBER — USE THE PARAMETERS FROM LANG_RT_PARAMS.SR!

Flags

These are status bits that the WSAVS or WSAVR instruction sets.

n

This number shares a word with Flags. It is the number of double words on the stack that are used for argument and dope vector addresses. WRTN uses n to pop the complete contents of the return block off the stack. (n is *not* always equal to the number of user-specified arguments on the stack.)

The parameter ARGS from LANG_RT_PARAMS.SR is the word offset to the 16 bits in which n resides. n is the right-most 15 of these 16 bits.

Old AC0

This is the saved value of AC0 at the time of the call.

If a function subprogram is returning a value to the caller in AC0, then the result is placed here where the WRTN instruction will pop it into the caller's AC0.

To access this entry in the return block in order to refer to the caller's AC0, use the parameter offset SAVE0. To modify this entry in order to return a fixed-point function value, use the parameter offset FRTN.

Decimal Frame Offset	Contents	Symbolic offset from FP (**)
	. . .	
		← Old SP
-10 - 2*n	Pointer to arg n	ARG"N"
	:	:
-10 - 2*i	Pointer to arg i	ARG"I"
	:	:
-10 - 4	Pointer to arg 2	ARG2
-10 - 2	Pointer to arg 1	ARG1
-10	Flags | n	(n/a) /ARGS
-8	Old AC0	SAVE0 (= FRTN)
-6	Old AC1	SAVE1
-4	Old AC2	SAVE2 (= FTMP)
-2	Old FP	OLDFP
0	C | Return PC (*)	← FP OLDPC
+2	First User Temporary	TMP
	Additional Procedure Data	
	. . .	

Increasing Addresses ↓

* The carry is bit 0 of this double word, and the PC is in bits 1 through 31.

** These symbolic offsets are defined in the source file, LANG_RT_PARAMS.SR, included with the AOS/VS Common Language Library. *In all cases* you should always use these symbolic offsets, and not numeric offsets, to address entries in the return block. The easiest *way* to work with these parameters is to add them to your permanent symbol table file, MASM.PS. See the *AOS/VS Macroassembler Reference Manual* for more information.

ID-00112

Figure 6-1. The VS/ECS Return Block

```
Source file: TEST_TYP_SUB.F77
Compiled on 2-Aug-82 at 11:49:52 by AOS/VS F77 Rev 02.00.00.00
Options: F77/CODE/L=TEST_TYP_SUB.LS

    1          PROGRAM TEST_TYP_SUB
    2
    3   C      THIS PROGRAM TESTS SUBROUTINE <TYP_SUB>.  THIS PROGRAM'S
    4   C         LISTING FILE, <TEST_TYP_SUB.LS>, SHOWS THE VS/ECS
    5   C         RETURN BLOCK USED FOR LINKAGE WITH SUBROUTINES.
    6
    7          R1 = 25.2
    8          R2 = 16.8
    9          I1 = 33
   10          I2 = 872
   11
   12   C      FIND THE OVERALL SUM.
   13
   14          CALL TYP_SUB (R1, R2, I1, I2, SUM4)
   15
   16          PRINT *, 'THE OVERALL SUM IS ', SUM4
   17
   18          STOP
   19          END
```

```
Source file: TEST_TYP_SUB.F77
Compiled on 2-Aug-82 at 11:50:00 by AOS/VS F77 Rev 02.00.00.00

                         Code Listing

Reloc  Opcode     Instruction                  Reference
--------------------------------------------------------------------------------

Line 1
              TEST_TYP_SUB:
00000  123471     WSAVS   20                  ; 16.

Line 7
00002  121011     XFLDS   0,.-4               ; [-2] 25.2
00004  161051     XFSTS   0,+14,3             ; 12. R1

Line 8
00006  121011     XFLDS   0,.-12              ; [-4] 16.8
00010  161051     XFSTS   0,+16,3             ; 14. R2

Line 9

00012  143051     NLDAI   41,0                ; 33.
00014  161431     XWSTA   0,+20,3             ; 16. I1
```

DG-25233

Figure 6-2. A Listing of TEST_TYP_SUB.F77 and Its Generated Code (continues)

```
Line 10
00016  147051     NLDAI   1550,1                ; 872.
00020  165431     XWSTA   1,+22,3               ; 18. I2

Line 14
00022  117051     XPEF    +24,3                 ; 20. SUM4
00024  117051     XPEF    +22,3                 ; 18. I2
00026  117051     XPEF    +20,3                 ; 16. I1
00030  117051     XPEF    +16,3                 ; 14. R2
00032  117051     XPEF    +14,3                 ; 12. R1
00034* 123311     LCALL   TYP_SUB,5

Line 16
00040  107051     XPEF    .+43                  ; [103] $.536
00042  107051     XPEF    .-64                  ; [-22] <dope>
00044* 123311     LCALL   F77E?EXTERNAL_FILE_INIT_LD_WRITE,2
00050  147051     NLDAI   12,1                  ; 10.
00052  132011     XLEF    2,.-70                ; [-16] THE OVERALL SUM IS
00054  176011     XLEF    3,+26,3               ; 22. <temp>
00056  163511     WBLM
00057* 123371     LPEF    +0
00062  157151     LDAFP   3
00063  137051     XPEFB   +54,3                 ; 44. <temp>
00065* 123311     LCALL   F77E?FILE_LD_WRITE_CHAR,2
00071  117051     XPEF    +24,3                 ; 20. SUM4
00073* 123311     LCALL   F77E?FILE_LD_WRITE_REAL4,1
00077* 123311     LCALL   F77E?FILE_TERM_LD_WRITE,0

Line 18
                $.536:
00103* 123311     LCALL   F.STOPN,0
```

DG-25233

Figure 6-2. A Listing of TEST_TYP_SUB.F77 and Its Generated Code (concluded)

```
Source file: TYP_SUB.F77
Compiled on 2-Aug-82 at 11:35:47 by AOS/VS F77 Rev 02.00.00.00
Options: F77/CODE/L=TYP_SUB.LS

    1          SUBROUTINE TYP_SUB (REAL_1, REAL_2, INT_1, INT_2, OVERALL)
    2
    3          SUM_REALS = REAL_1 + REAL_2
    4
    5          SUM_INTS = FLOAT(INT_1 + INT_2)
    6
    7          OVERALL = SUM_REALS + SUM_INTS
    8
    9          RETURN
   10          END


Source file: TYP_SUB.F77
Compiled on 2-Aug-82 at 11:35:54 by AOS/VS F77 Rev 02.00.00.00

                        Code Listing


Reloc  Opcode     Instruction                 Reference
--------------------------------------------------------------------------------


Line 1
              TYP_SUB:
00000  123471     WSAVS    11                 ; 9.

Line 3
00002  161011     XFLDS    0,@-14,3           ; -12. REAL_1
00004  160011     XFAMS    0,@-16,3           ; -14. REAL_2
00006  161051     XFSTS    0,+16,3            ; 14. SUM_REALS

Line 5
00010  161411     XWLDA    0,@-20,3           ; -16. INT_1
00012  160430     XWADD    0,@-22,3           ; -18. INT_2
00014  102251     WFLAD    0,0
00015  102330     FRDS     0,0
00016  161051     XFSTS    0,+20,3            ; 16. SUM_INTS

Line 7
00020  161011     XFLDS    0,+20,3            ; 16. SUM_INTS
00022  160011     XFAMS    0,+16,3            ; 14. SUM_REALS
00024  161051     XFSTS    0,@-24,3           ; -20. OVERALL

Line 9
00026  103651     WRTN
```

DG-25234

Figure 6-3. A Listing of TYP_SUB.F77 and Its Generated Code

Old AC1

This is the saved value of AC1 at the time of the call.

To access this entry in the return block, use the parameter offset SAVE1.

Old AC2

This is the saved value of AC2 at the time of the call.

If a function subprogram is returning a value in a temporary location, AC2 will have been loaded (prior to the call) with a word pointer to a suitable temporary location.

To access this entry in the return block in order to refer to the caller's AC2, use the parameter offset SAVE2. To refer to this entry as a pointer to the temporary location, use the parameter offset FTMP.

Old FP

This is the caller's frame pointer.

To access this entry in the return block, use the parameter offset OLDFP.

C | Return PC

These are the values of the carry bit and of the PC. The WRTN instruction restores these values.

To access these entries in the return block, use the parameter offset OLDPC.

Note how Figures 6-2 and 6-3 illustrate the general principles of Figure 6-1. For example, the fourth argument in both program units is the second of the two integer numbers to be added. Its name is I2 in TEST_TYP_SUB.F77 and INT_2 in TYP_SUB.F77. Both I2 and INT_2 refer to the same memory location; its offset is 18 words from the frame pointer. Observe that the compiler has generated code that places this *fourth* argument on the stack *after* it has placed the *fifth* argument there.

Function Subprograms

Function subprograms return one value to the caller. Under the VS/ECS convention, this value arrives to the caller in either an accumulator or in a compiler-generated temporary location, depending on the data type.

The following list shows where a VS/ECS function returns its value and under what circumstances

In AC0:	If the function specifies a result data type of INTEGER*2, INTEGER*4, LOGICAL*2, or LOGICAL*4, then the VS/ECS software returns the value in AC0 by placing it in FRTN in the return block just before returning to the caller.
In FPAC0:	If the function specifies a result data type of REAL*4 or REAL*8 (DOUBLE PRECISION), then the VS/ECS software returns the value by moving it into floating-point accumulator FPAC0 just before returning to the caller.
In a TEMPORARY whose address is in AC2:	If the function specifies any data type other than the above six, then the VS/ECS software places the result in a suitable temporary. The called routine finds this temporary by using the word address found in FTMP of the return block. Instructions could need to copy and convert this address to a byte address inside character functions. However, the called routine must not change the actual value in FTMP.

Coding Assembly Language Routines for Use with F77 with Macros

When writing assembly language routines for F77, you may want to use the set of macros and symbols supplied in the files VF77SYM.SR, F77_FMAC.SR, PARF77.SR, and LANG_RT_PARAMS.SR. This section describes the use of the FORTRAN CALL macro set contained in the first two of these files.

These macros are supplied to aid in converting existing FORTRAN 5 programs that have used FMAC.SR . The AOS/VS file F77_FMAC.SR is a subset of the FORTRAN 5 file FMAC.SR, because some concepts and features present in the AOS environment do not transport to the AOS/VS environment.

Some of the things that the macros are used for are:

- To handle passed-in arguments (NXTARG, SKPARG).
- To manipulate the stack- and frame-pointers (ISZSP, DSZSP, ISZFP, DSZFP).
- To set up entry points to your routine (NENTRY, PENTRY, FENTRY, ZENTRY).
- To define arguments and temporaries (DEFARGS, DEFTEMPS).
- To return from your routine (ISA.NORM, ISA.ERR).

Documentation on all the macros is available in file F77_FMAC.SR and in the *FORTRAN 5 Programmer's Guide (AOS)*. The macros summarized here are

TITLE
S?ATTR
DEFARGS
DEFTMPS
DEF
FENTRY
FRET
END

If these macros are used, **TITLE** must be the first one invoked (except for preliminary comment lines). This macro specifies the title of the routine you are writing and initializes the environment for the other macros.

When using FMAC.SR, you needed to use the **S?ATTR** macro. Under AOS/VS, this is no longer needed. Any use of S?ATTR will be a no-op.

DEFARGS immediately follows TITLE. This macro is used to start the definition of your routine's arguments. You should define each argument using the DEF macro. For example:

```
TITLE   ESSAY
DEFARGS
  DEF   SOUND
  DEF   SPECIOUS
   .
   .
```

These four lines declare two arguments, SOUND and SPECIOUS, in the routine ESSAY. Even if your routine has no arguments, you must use DEFARGS.

DEFTMPS follows DEFARGS and DEF's (if any). **DEFTMPS** is used to start the definitions of your routine's temporaries. You use **DEF** to define each temporary. For example:

```
     .
     .

DEFTMPS
  DEF  B0 (10.)       ; Argument is size in 16-bit words
                      ;    (must be in parentheses).
                      ; When no argument is given,
                      ;    the default length of 1 word is assumed.
```

DEFTMPS must appear even if your routine does not require any temporaries.

DEF names each of your routine's arguments and temporaries. You must name the arguments in the order in which they appear when the routine is CALLed. In FORTRAN programming environments, it is always your responsibility to ensure that the arguments provided by the calling routine match those expected by the called routine in number, order and type.

DEF assigns to the symbol you supply a unique, sequential offset on the stack. Entries on the stack are addressed by indexing from the current frame pointer, which must be loaded into either AC2 or AC3. At the beginning of your routine, AC3 contains the value of the frame pointer. To access an argument passed by the caller, use the symbol for the argument, indexed by the AC containing the frame pointer, as an indirect address. Temporaries on the stack are accessed by using the symbol for the temporary, indexed by the AC containing the frame pointer.

FENTRY follows DEFARGS and DEFTMPS. **FENTRY** generates a WSAVS instruction and defines your entry point. AC3 contains the frame pointer when the first instruction after FENTRY is executed.

Finally, your subprogram code can be written. You can use any AC's or FPAC's you need — the AC's will be restored as required when your routine completes.

FRET returns control to the calling routine. This macro generates a WRTN instruction, which restores the caller's environment, and resumes execution of the caller.

END must be the last line of your routine. This macro generates a .END assembler directive, and terminates the environment set up by the previous macros.

See the next section for an examples of complete assembly language subroutines.

F77-to-Assembly Interface Examples

Figure 6-4 contains a listing of program TEST_RUNTM.F77. As its name implies, the program tests subroutine RUNTM which, in turn, makes a ?RUNTM system call to obtain process statistics. Figure 6-5 contains a listing of the first version of assembly language subroutine RUNTM.SR. It uses the symbols for stack displacement from the files VF77SYM.SR (and F77_FMAC.SR to access the arguments from the calling routine. Figure 6-6 contains a listing of the second version of assembly language subroutine RUNTM.SR. It also uses symbols for stack displacement from VF77SYM.SR and F77_FMAC.SR and, in addition, uses FORTRAN 77 CALL macros from these files.

NOTE: The first pages of both versions of RUNTM.SR are identical, except for the instructions to assemble RUNTM.SR.

```
      PROGRAM TEST_RUNTM

C     THIS PROGRAM TESTS SUBROUTINE <RUNTM> WHICH RETURNS THE
C        PROCESS'S RUNTIME STATISTICS.


C     THE ARGUMENTS GIVEN TO <RUNTM> ARE:
C        NONE

C     THE ARGUMENTS RETURNED BY <RUNTM> ARE:
         INTEGER*4 ELAPSED          ! ELAPSED TIME IN SECONDS
C                                      SINCE PROCESS'S CREATION
         INTEGER*4 CPU              ! PROCESS'S CPU TIME IN
C                                      MILLISECONDS
         INTEGER*4 IO_BLOCKS        ! NUMBER OF I/O BLOCKS READ
C                                      OR WRITTEN
         INTEGER*4 PAGE_MILSECS     ! NUMBER OF PAGE/MILLISECONDS
         INTEGER*4 IER              ! ERROR CODE FROM <RUNTM>


C     CRUNCH SOME NUMBERS TO ACCUMULATE SOME CPU TIME.
      DO 10 I = 1, 10000
              X = FLOAT(I)
              VARIABLE1 = SIN(X) + TAN(X) - SQRT(X)
              VARIABLE2 = 1.0/VARIABLE1
   10 CONTINUE

C     OBTAIN THE PROCESS'S RUNTIME STATISTICS.
      CALL RUNTM(ELAPSED, CPU, IO_BLOCKS, PAGE_MILSECS, IER)

C     DISPLAY THE RESULTS.
      IF ( IER .NE. 0 ) THEN
              PRINT *, 'ERROR ', IER, ' OCCURRED DURING EXECUTION ',
     1                 'OF SUBROUTINE RUNTM.'
      ELSE
              PRINT *, 'PROCESS ELAPSED TIME IN SECONDS:    ', ELAPSED
              PRINT *, 'PROCESS CPU TIME IN MILLISECONDS:   ', CPU
              PRINT *, 'NUMBER OF I/O BLOCKS:               ', IO_BLOCKS
              PRINT *, 'NUMBER OF PAGE/MILLISECONDS:        ', PAGE_MILSECS
      ENDIF

      PRINT *
      PRINT *, '*** END OF JOB ***'

      CALL EXIT
      END
```

DG-25235

Figure 6-4. Main Program TEST_RUNTM.F77

```
;                        SUBROUTINE RUNTM.SR
;
; This F77-callable assembly subroutine obtains process runtime
;               statistics by making a "?RUNTM" system call.  It uses
;               the VS/ECS conventions.
;
; This routine executes in the sharable code area, but builds the packet
;               for the system call on the user's stack, in unshared
;               memory.  Note carefully how the offsets that define
;               the system call packet are used for addressing the stack.
;
; CALL Syntax:
;               CALL RUNTM (IELAPSED, ICPU, IIO_BLKS, IP_MS, IER)
;
; Arguments (all returned to caller):
;               IELAPSED:     INTEGER*4   (elapsed time in seconds
;                                             since process's creation)
;               ICPU:         INTEGER*4   (process's CPU time in
;                                             milliseconds)
;               IIO_BLKS      INTEGER*4   (number of I/O blocks read
;                                             or written)
;               IP_MS:        INTEGER*4   (number of page/milliseconds)
;               IER:          INTEGER*4   (error code from ?RUNTM)
;
;
; To assemble this routine:
;
;       *  With LANG_RT_PARAMS.SR built into MASM.PS:
;
;                 X MASM RUNTM
;
;       *  With LANG_RT_PARAMS.SR not built into MASM.PS:
;
;                 X MASM/O=RUNTM.OB LANG_RT_PARAMS.SR/PASS1 RUNTM
;
; To link this routine with F77 programs:
;
;       F77LINK main-program-name RUNTM
;
```

DG-25236

Figure 6-5. Subroutine RUNTM.SR, Version 1 (continues)

```
; **************************    Version 1    **************************

        .TITLE          RUNTM
        .ENT    RUNTM
        .NREL   1                       ; Shared.


        PACKET = TMP                    ; To build ?RUNTM packet on the stack,
                                        ;  define packet start as the offset to
                                        ;  the first user temporary,
        PACKETLEN = (?GRLTH+1)/2
                                        ;  and calculate the maximum number of
                                        ;  double words on the stack which will
                                        ;  be needed to build the packet -- by
                                        ;  adding 1 to packet length in single
                                        ;  words, and dividing by 2.

RUNTM:                                  ; Routine entry:
        WSAVS   PACKETLEN               ; Save the state, and enough stack
                                        ;  space for the packet, and put
                                        ;  AC3 <== my FRAME POINTER.

                                        ; Make system call:
        WADC    0,0                     ; AC0 <== -1 to indicate this process
        XLEF    2,PACKET,3              ; AC2 <== address of packet
        ?RUNTM                          ; Get runtime stats
        WBR     RUNTERROR               ; Error on system call
                                        ; Good return:
                                         ; Move values into caller's arguments
        XWLDA   0,PACKET+?GRRH,3         ; Get elapsed time in seconds
        XWSTA   0,@ARG1,3                ; Put into 1st argument via pointer
        XWLDA   0,PACKET+?GRCH,3         ; Get CPU time in milliseconds
        XWSTA   0,@ARG2,3                ; Put into 2nd argument via pointer
        XWLDA   0,PACKET+?GRIH,3         ; Get I/O blocks read or written
        XWSTA   0,@ARG3,3                ; Put into 3rd argument via pointer
        XWLDA   0,PACKET+?GRPH,3         ; Get # page/milliseconds
        XWSTA   0,@ARG4,3                ; Put into 4th argument via pointer

        WSUB    0,0                      ; Zero AC0 to show good return

RUNTERROR:                              ; Enter here if error.  Common path
                                        ;  for setting error return variable:
        XWSTA   0,@ARG5,3               ; Put (AC0) into 5th argument.

        WRTN                            ; Go back to F77 caller.

        .END
```

DG-25236

Figure 6-5. Subroutine RUNTM.SR, Version 1 (concluded)

```
;                         SUBROUTINE RUNTM.SR
;
; This F77-callable assembly subroutine obtains process runtime
;               statistics by making a "?RUNTM" system call.  It uses
;               the VS/ECS conventions.
;
; This routine executes in the sharable code area, but builds the packet
;               for the system call on the user's stack, in unshared
;               memory.  Note carefully how the offsets that define
;               the system call packet are used for addressing the stack.
;
; CALL Syntax:
;               CALL RUNTM (IELAPSED, ICPU, IIO_BLKS, IP_MS, IER)
;
; Arguments (all returned to caller):
;               IELAPSED:    INTEGER*4   (elapsed time in seconds
;                                            since process's creation)
;               ICPU:        INTEGER*4   (process's CPU time in
;                                            milliseconds)
;               IIO_BLKS     INTEGER*4   (number of I/O blocks read
;                                            or written)
;               IP_MS:       INTEGER*4   (number of page/milliseconds)
;               IER:         INTEGER*4   (error code from ?RUNTM)
;
;
; To assemble this routine:
;
;       *  With LANG_RT_PARAMS.SR built into MASM.PS:
;
;                 X MASM RUNTM
;
;       *  With LANG_RT_PARAMS.SR not built into MASM.PS:
;
;                 X MASM/O=RUNTM.OB LANG_RT_PARAMS.SR/PASS1
;                                VF77SYM.SR/PASS1 F77_FMAC.SR/PASS1 RUNTM
;
;
; To link this routine with F77 programs:
;
;       F77LINK main-program-name RUNTM
;
```

DG-25237

Figure 6-6. Subroutine RUNTM.SR, Version 2 (continues)

```
; **************************  Version 2  **************************

; Macros defined in F77_FMAC.SR are identified by "@FMAC" in comment field.

       TITLE  RUNTM            ; Name the object module, generate      @FMAC
                               ;  a language identifying tag comment,
                               ;  and specify shared code.

DEFARGS                        ; Begin argument definitions:           @FMAC
  DEF IELAPSED                 ;                                       @FMAC
  DEF ICPU                     ;                                       @FMAC
  DEF IIO                      ;                                       @FMAC
  DEF IPMS                     ;                                       @FMAC
  DEF ERR__STAT                ; (Use two underscores since this is    @FMAC
                               ;  an argument to a macro that removes
                               ;  one of them: "ERR__STAT" becomes
                               ;  "ERR_STAT" as desired.)

DEFTMPS                        ; Begin temporary definitions:          @FMAC
  DEF PACKET (?GRLTH)          ;  To build ?RUNTM packet on the stack, @FMAC
                               ;  define PACKET as a temporary, with
                               ;  length equal to the maximum number of
                               ;  words needed to build the packet.

FENTRY RUNTM                   ; Routine entry:                        @FMAC

       WADC     0,0            ; AC0 <== -1 to indicate this process
       XLEF     2,PACKET,3     ; AC2 <== address of packet
       ?RUNTM                  ; Get runtime stats
        WBR     RUNTERR        ; Error on system call
                               ; Good return:
                               ; move values into caller's arguments
       XWLDA  0,PACKET+?GRRH,3 ; Get elapsed time in seconds
       XWSTA  0,@IELAPSED,3    ; Put into IELAPSED via address on stack
       XWLDA  0,PACKET+?GRCH,3 ; Get CPU time in milliseconds
       XWSTA  0,@ICPU,3        ; Put into ICPU via address on stack
       XWLDA  0,PACKET+?GRIH,3 ; Get I/O blocks read or written
       XWSTA  0,@IIO,3         ; Put into IIO via address on stack
       XWLDA  0,PACKET+?GRPH,3 ; Get # page/milliseconds
       XWSTA  0,@IPMS,3        ; Put into IPMS via address on stack

       WSUB    0,0             ; Zero AC0 to show good return

RUNTERR:                       ; Enter here if error.  Common path
                               ;  for setting error return variable:
       XWSTA  0,@ERR_STAT,3    ; Put code into ERR_STAT via argument.

       FRET                    ; Go back to F77 caller.                @FMAC

       END                     ;                                       @FMAC
```

DG-25237

Figure 6-6. Subroutine RUNTM.SR, Version 2 (concluded)

The following commands assemble the first version of RUNTM.SR (assuming that LANG_RT_PARAMS.SR is not built into MASM.PS), compile TEST_RUNTM.F77, and create TEST_RUNTM.PR:

```
X MASM/O=RUNTM.OB LANG_RT_PARAMS.SR/PASS1 RUNTM
F77 TEST_RUNTM
F77LINK TEST_RUNTM RUNTM
```

The following commands assemble the second version of RUNTM.SR (assuming that LANG_RT_PARAMS.SR is not built into MASM.PS), compile TEST_RUNTM.F77, and create TEST_RUNTM.PR:

```
X MASM/O=RUNTM.OB LANG_RT_PARAMS.SR/PASS1 &
      VF77SYM.SR/PASS1 F77_FMAC.SR/PASS1 RUNTM
F77 TEST_RUNTM
F77LINK TEST_RUNTM RUNTM
```

Let's look at the results of executing TEST_RUNTM.PR (with either version of RUNTM.SR):

```
) X TEST_RUNTM )
PROCESS ELAPSED TIME IN SECONDS:       3
PROCESS CPU TIME IN MILLISECONDS:      1381
NUMBER OF I/O BLOCKS:                  0
NUMBER OF PAGE/MILLISECONDS:           13
*** END OF JOB ***
```

The results usually vary slightly each time TEST_RUNTM.PR executes.

Incompatibilities Between AOS and AOS/VS F77 Macro F77_FMAC.SR

Argument Names

The symbols ARG0, ARG1, ..., ARG15 were used in AOS F77 F77_FMAC.SR to provide symbolic access to arguments on the stack that are passed to assembly language subroutines. AOS/VS LANG_RT_PARAMS.SR defines different values of these symbols because of the different method of passing arguments. AOS F77 made a distinction between functions and subroutines. If the routine was a subroutine, then the first argument in the list was referred to as ARG0; if it was a function, then ARG1 was the first argument, and ARG0 designated the slot to be used for the function result. With AOS/VS F77, the last slot pushed is the first argument that the user wrote; no ARG0 is needed.

This might cause a problem with existing assembly language routines that are migrating from the AOS F77 or AOS FORTRAN 5 environment. Consider the two possible cases:

Routine type	Conversion action required
Subroutine (no result argument)	Add 1 to the ARGn references (e.g., change ARG1 to ARG2).
Function	Results are handled DIFFERENTLY. See the previous section "Function Subprograms."

Differences in Macros ISZFP, DSZFP, ISZSP, and DSZSP

The macros ISZFP, DSZFP, ISZSP, and DSZSP are different under AOS/VS because the stack- and frame-pointers are not present in a memory location in the address space as they are under AOS.

Under AOS, these macros are invoked with no argument to increment or decrement the stack or frame pointer, and with the argument "@" to increment or decrement the location pointed to by the corresponding pointer. The AOS/VS versions need to be invoked with an accumulator that they can use for scratch purposes for the operation. The previous contents of the accumulator will be lost. If the "@" argument is used, then the next sequential word will be skipped if the result of the operation is zero.

AOS Form	AOS/VS Form
ISZFP	ISZFP n
ISZFP @	ISZFP @,n

If you use the the "@" option, n must be 2 or 3 (an index AC).

Nonsupported Macros

AOS/VS F77 does not support the following macros in F77_FMAC.SR.

ARGS	C?WNL	P?SET
C?BAD	IS1.ERR	TMPS
C?BNL	IS1.NORM	U?DATA
C?DEF	ISAENTRY	U?DSZ
C?NIL	LDC	U?ISZ
C?NOD	M?BARG	U?LDA
C?WAD	M?WARG	U?STA

New Macros ISA.NORM and ISA.ERR

The macros ISA.NORM and ISA.ERR have been changed in the F77_FMAC.SR file supplied with AOS F77 and AOS/VS F77 from the previous versions of FMAC.SR. The AOS F77 and AOS/VS F77 functionality of these two changed macros is identical.

The macros ISA.NORM and ISA.ERR have been changed because of a side effect of the presence of character data in FORTRAN 77. When you pass a character argument, F77 also passes a "dope vector" for that argument which describes the length of the character argument. This length is used by the called routine when the character argument is referenced. A call of the form

CALL SUB(C1, I, C2, J)

where C1 and C2 are CHARACTER variables, is really treated by the compiler as

CALL SUB(C1, I, C2, J, <dope for C1>, #, <dope for C2>)

Here "#" is simply a placeholder because "I", not being a character argument, does not need a dope vector. Note that there is no corresponding placeholder for "J" at the end of the list because it would have been the first argument whose address is pushed (such addresses are pushed in reverse order) and would be as useless as extra leading zeros when writing numbers.

The ISA.NORM and ISA.ERR macros from FMAC.SR assumed that the last argument in the list (whose address was the first one pushed) was the ier argument. The macro had no way of knowing that the last argument was not really the ier argument, but rather a dope vector, when character entities were passed.

F77_FMAC.SR contains modified versions of ISA.NORM and ISA.ERR:

Old Syntax	New Syntax
ISA.NORM	ISA.NORM *[ier_pos]*
ISA.ERR *[errorcode]*	ISA.ERR *[new_errorcode [,ier_pos]]*

If the routine you are writing is not called with character arguments, then you may omit "ier_pos". The presence of "ier_pos" tells ISA.NORM and ISA.ERR not to assume that the last argument is the "ier" argument, and to use the supplied position.

"new_errorcode" is used exactly as "errorcode" except that it can additionally take the value "*", which means to use the value of the errorcode that is in AC0. The symbol "*" is a placeholder, which allows you to specify a nondefault "ier_pos" and to supply the errorcode in AC0.

Examples:	ISA.ERR	*,3	- ier is argument 3, error code is in AC0
	ISA.NORM	5	- ier is argument 5

Compatibility Between Languages

One of the features of F77 is that the calling conventions and the return block format it uses are compatible with other AOS/VS languages that also use the Common Code Generator. We refer to these conventions as the "VS/ECS" — an acronym for Virtual System / External Calling Sequence. The languages using the Common Code Generator are BASIC, C, COBOL, PASCAL, and PL/I.

For example, you can write a subroutine in F77 to call a procedure written in PL/I; a PL/I procedure can refer to an F77 function subprogram in the same way it would refer to a PL/I procedure with a RETURNS attribute; and BASIC programs can access subroutines written in F77. The rest of this chapter explains subprograms written in F77 and linkage to them.

The arguments in the parameter lists of the calling and called routines must agree in number, order, and type. Furthermore, you must make sure that the internal representations of any arguments or returned values are compatible. For example, an F77 argument declared as INTEGER*2 requires a PL/I caller to declare its corresponding argument as FIXED BIN(15). Some data types in other languages may not have a corresponding data type in F77, and vice versa. For example:

- F77 does not support any data types that correspond to PL/I's ALIGNED CHARACTER, VARYING CHARACTER, or BIT data types.
- F77 does not support any data type that corresponds to BASIC's variable length strings.
- COBOL does not support any data type that corresponds to F77's COMPLEX data type.

You must be familiar with the internal data representation of both languages.

Multidimension Array Storage

F77 stores the elements of a multidimension array differently from other languages. It stores them by varying the left-most subscript most rapidly, while other languages vary the right-most subscript most rapidly. For example, the northern New England states have the abbreviations VT, NH, and ME (for Vermont, New Hampshire, and Maine) while the abbreviations for the southern New England states are MA, CT, and RI (for Massachusetts, Connecticut, and Rhode Island). It seems natural to place these six abbreviations in a two-dimension array with two rows and three columns. The following sequences of F77 and PL/I statements accomplish this.

```
PROGRAM STATES                    STATES: PROCEDURE;
CHARACTER*2 NE_STATES(2,3)        DECLARE NE_STATES(2,3)
                                     CHARACTER(2);
NE_STATES(1,1) = 'VT'             NE_STATES(1,1) = 'VT';
NE_STATES(1,2) = 'NH'             NE_STATES(1,2) = 'NH';
NE_STATES(1,3) = 'ME'             NE_STATES(1,3) = 'ME';
NE_STATES(2,1) = 'MA'             NE_STATES(2,1) = 'MA';
NE_STATES(2,2) = 'CT'             NE_STATES(2,2) = 'CT';
NE_STATES(2,3) = 'RI'             NE_STATES(2,3) = 'RI';
```

We can think that the six elements of NE_STATES are stored as

```
          Column 1      Column 2      Column 3

Row 1     VT            NH            ME
Row 2     MA            CT            RI
```

to aid in the coding process.

Such thinking helps in constructing statements to interchange the corresponding elements in the rows so that NE_STATES would then contain

	Column 1	Column 2	Column 3
Row 1	MA	CT	RI
Row 2	VT	NH	ME

F77 and the other Common Code Generator languages store the six elements of NE_STATES in six sequential storage locations with increasing addresses. F77 stores the six elements *differently* from the other languages. See Figure 6-7.

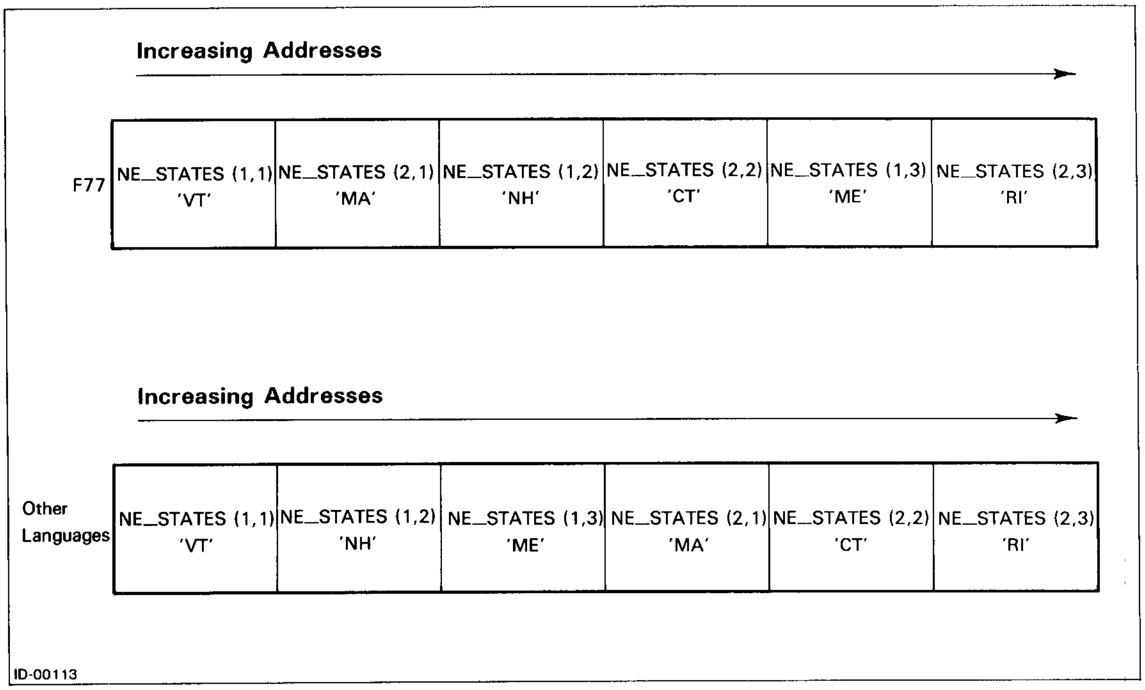


Figure 6-7. An Example of Storage of Multidimension Arrays by F77 and Other Languages

We write a rather specialized F77 subroutine to swap the corresponding elements of an array such as NE_STATES. The resulting subroutine SWAP_ROWS.F77 appears next.

```
      SUBROUTINE SWAP_ROWS (ARRAY)
      INTEGER*2 COLUMN
      CHARACTER*2 ARRAY(2,3), TEMP

      DO 10 COLUMN = 1, 3
              TEMP = ARRAY(1,COLUMN)
              ARRAY(1,COLUMN) = ARRAY(2,COLUMN)
              ARRAY(2,COLUMN) = TEMP
10    CONTINUE
      RETURN
      END
```

If we add the statement

```
CALL SWAP_ROWS (NE_STATES)
```

to STATES.F77, then its compilation and linking with SWAP_ROWS correctly results at runtime in

	Column 1	Column 2	Column 3
Row 1	MA	CT	RI
Row 2	VT	NH	ME

However, if we add the statement

```
CALL SWAP_ROWS (NE_STATES);
```

to STATES.PL1, then its compilation and linking with SWAP_ROWS *incorrectly* results at runtime in

	Column 1	Column 2	Column 3
Row 1	NH	VT	MA
Row 2	ME	RI	CT

The difference in the results occurs because of the different sequential storage of array NE_STATES by F77 and by PL/I.

To generalize from this example, you must be careful when you write F77 subroutines to process multidimension arrays from calling programs that are in a language different from F77. You have to allow for F77's different storage of these arrays. Single dimension arrays and simple variables present no such problem.

Case Sensitivity

F77 and BASIC are case-insensitive because they map all external references to uppercase letters. For example, a CALL to subroutine **VaRiEs** compels Link to locate and load the module with external entry point "VARIES" into the program file. PL/I and Link are case sensitive. So

- You must declare in uppercase letters the name of any F77 subprogram that you call or refer to in a PL/I source module.
- You should declare in uppercase letters the name of any PL/I subprogram that you call or refer to in an F77 source module.

A general way to avoid problems is to use uppercase letters in any program module name and in commands to Link.

LANG_RT.LB

F77 and the other Common Code Generator languages use the same set of common mathematics and system interface routines, all of which conform to VS/ECS. This set is in LANG_RT.LB, the AOS/VS Common Language Library. Each language also uses a separate set of runtime routines to handle I/O and certain support functions. These routines are language-specific. If you try to link these separate runtime routines into the same program file, conflicts could arise between the names of (and operations performed by) routines from F77, and the names and operations from another language. To avoid this situation, design your program so that only one language does all of the program's I/O.

A Sample Subprogram and its Caller

Figure 6-8 contains a listing of subroutine subprogram GENERAL.F77. This subroutine:

- Receives an array of single-precision floating-point numbers.
- Receives an array of INTEGER*2 numbers.
- Receives a single-precision floating-point number that is an angle measurement (in degrees).
- Returns the largest of the single-precision floating-point numbers.
- Returns the smallest of the INTEGER*2 numbers.
- Returns the trigonometric sine of the received angle.
- Returns 1 in an error variable if there are too few elements in either array; otherwise, returns 0.

GENERAL.F77 exists so that the other Common Code Generator languages can call it to process their data. You will soon see sample programs, written in BASIC, C, PASCAL, and PL/I (as well as F77) that call GENERAL. COBOL is different enough to require a modification of GENERAL.F77 whose name is GENERAL1.F77.

Figure 6-9 contains a listing of main program TEST_GENERAL.F77. As its name implies, TEST_GENERAL.F77 is an F77 program to test subroutine GENERAL.

Note that all the variables in GENERAL.F77 and TEST_GENERAL.F77 are either REAL*4 or INTEGER*2. Each of the Common Code Generator Languages supports these two data types.

The compilation, link, and execution commands for TEST_GENERAL.F77 and GENERAL.F77 are

```
F77 TEST_GENERAL
F77 GENERAL
F77LINK TEST_GENERAL GENERAL
XEQ TEST_GENERAL
```

The output displayed in response to the last command is

```
THE LARGEST REAL*4 NUMBER IS:            8.94
THE SMALLEST INTEGER*2 NUMBER IS:        -2846
THE SINE OF 30. DEGREES IS: .5

STOP
```

```
      SUBROUTINE GENERAL (REAL_ARRAY, REAL_SIZE, INT_ARRAY, INT_SIZE,
     +   ANGLE, LARGEST_REAL, SMALLEST_INT, SINE_ANGLE, ERROR)

      INTEGER*2 REAL_SIZE
      REAL*4 REAL_ARRAY(REAL_SIZE)
      INTEGER*2 INT_SIZE
      INTEGER*2 INT_ARRAY(INT_SIZE)
      REAL*4 ANGLE
      REAL*4 LARGEST_REAL
      INTEGER*2 SMALLEST_INT
      REAL*4 SINE_ANGLE
      INTEGER*2 ERROR

      ERROR = 0         ! Assume there's no error in the array sizes.
C                         But, check the sizes and RETURN with the
C                         error variable set if there is an error.
      IF ( REAL_SIZE .LT. 1  .OR.  INT_SIZE .LT. 1 ) THEN
              ERROR = 1
              RETURN
      ENDIF

C     Find the largest element in <REAL_ARRAY> and place it in
C        <LARGEST_REAL>.
      LARGEST_REAL = REAL_ARRAY(1)
      DO 10 I = 2, REAL_SIZE
              IF ( REAL_ARRAY(I) .GT. LARGEST_REAL )
     1             LARGEST_REAL = REAL_ARRAY(I)
   10 CONTINUE

C     Find the smallest element in <INT_ARRAY> and place it in
C        <SMALLEST_INT>.
      SMALLEST_INT = INT_ARRAY(1)
      DO 20 I = 2, INT_SIZE
              IF ( INT_ARRAY(I) .LT. SMALLEST_INT )
     1             SMALLEST_INT = INT_ARRAY(I)
   20 CONTINUE

C     Compute the sine of <ANGLE> after converting <ANGLE> from degrees to
C        radians.
      SINE_ANGLE = SIN(3.141593*ANGLE/180.0) ! PI radians = 180 degrees.

C     Done!
      RETURN
      END
```

DG-25238

Figure 6-8. Subroutine Subprogram GENERAL.F77

```
      PROGRAM TEST_GENERAL    ! to test subroutine GENERAL

      REAL*4 REALS(10)    /  3.40,  8.61, -6.00,  8.94,  4.18,
     1                       7.56, -9.57,  0.00, -1.24,  0.52 /
      INTEGER*2 R_SIZE    / 10 /

      INTEGER*2 INTS(5)   /   386, -2846,  3091,   -33,  5104 /
      INTEGER*2 I_SIZE    /     5 /

      REAL*4 ANGLE        /  30.0 /

      REAL*4 BIG_REAL
      INTEGER*2 SMALL_INT
      REAL*4 SINE_ANGLE
      INTEGER*2 IER

C     Here we go ...

      CALL GENERAL (REALS, R_SIZE, INTS, I_SIZE, ANGLE,
     1   BIG_REAL, SMALL_INT, SINE_ANGLE, IER)

      IF ( IER .EQ. 0 ) THEN

              PRINT *
              PRINT *, 'THE LARGEST REAL*4 NUMBER IS:        ', BIG_REAL
              PRINT *, 'THE SMALLEST INTEGER*2 NUMBER IS:    ', SMALL_INT
              PRINT *, 'THE SINE OF ', ANGLE, ' DEGREES IS: ', SINE_ANGLE
              PRINT *
      ELSE
              PRINT *
              PRINT *, 'ERROR OCCURRED IN SUBROUTINE GENERAL.'
              PRINT *
      ENDIF

      STOP
      END
```

DG-25239

Figure 6-9. Main Program TEST_GENERAL.F77

High-Level Languages and F77 Subroutines

BASIC, C, COBOL, F77, PASCAL, and PL/I follow VS/ECS. The rest of this chapter consists of the following for each language, except F77:

- A list of F77 data types and the language's corresponding data types.
- A sample program in the language that calls GENERAL.F77 (or, in the case of COBOL, GENERAL1.F77).
- An explanation of any peculiarities of the language that affect F77 subroutines. You're already aware of COBOL.

BASIC and F77

This section lists F77 data types and their BASIC correspondents. It also shows the BASIC program TEST_GENERAL.BASIC, that calls subroutine GENERAL.F77.

F77 and BASIC Data Types

F77	BASIC
INTEGER*2	INTEGER and INTEGER*2
INTEGER*4	INTEGER*4
REAL*4	REAL and REAL*4
REAL*8 and DOUBLE PRECISION	REAL*8
COMPLEX	None
COMPLEX*16 and DOUBLE PRECISION COMPLEX	None
LOGICAL*2	None — But, a BASIC INTEGER*2 variable whose value is 0 or -1 is the same as a respective F77 LOGICAL*2 variable whose value is .FALSE. or .TRUE..
LOGICAL*4	None — But, a BASIC INTEGER*4 variable whose value is 0 or -1 is the same as a respective F77 LOGICAL*4 variable whose value is .FALSE. or .TRUE..
CHARACTER*N ("N" is a constant.)	string_name%*N (fixed-length string)

Sample Program

Program TEST_GENERAL.BASIC calls subroutine GENERAL. This program's listing is shown in Figure 6-10. You can create file TEST_GENERAL.BASIC by using SED or SPEED, or by using the BASIC interpreter itself. TEST_GENERAL.BASIC is a data-sensitive file.

```
00100 REM PROGRAM TEST_GENERAL   ! to test subroutine GENERAL
00110
00120 OPTION BASE 1
00130
00140 DECLARE REAL*4 REALS(10)
00150 DATA 3.40,  8.61, -6.00,  8.94,  4.18
00160 DATA 7.56, -9.57,  0.00, -1.24,  0.52
00170 MAT READ REALS
00180
00190 DECLARE INTEGER*2 R_SIZE
00200 DATA 10
00210 READ R_SIZE
00220
00230 DECLARE INTEGER*2 INTS(5)
00240 DATA 386, -2846,  3091,   -33,  5104
00250 MAT READ INTS
00260
00270 DECLARE INTEGER*2 I_SIZE
00280 DATA 5
00290 READ I_SIZE
00300
00310 DECLARE REAL*4 ANGLE
00320 DATA 30.0
00330 READ ANGLE
00340
00350 DECLARE REAL*4 BIG_REAL
00360 DECLARE INTEGER*2 SMALL_INT
00370 DECLARE REAL*4 SINE_ANGLE
00380 DECLARE INTEGER*2 IER
00390
00400 REM Assigning values to the arguments of GENERAL that this
00410 REM    program doesn't compute with avoids a code 71167
00420 REM    warning message ("Identifier not assigned").
00430 LET BIG_REAL, SMALL_INT, SINE_ANGLE, IER = 0
00440
00450 REM Here we go ...
00460 ASM GENERAL(REALS(),R_SIZE,INTS(),I_SIZE,ANGLE,&
                  BIG_REAL,SMALL_INT,SINE_ANGLE,IER)
00470
00480 IF IER = 0 THEN
00490       PRINT
00500       PRINT "THE LARGEST REAL*4 NUMBER IS:       "; BIG_REAL
00510       PRINT "THE SMALLEST INTEGER*2 NUMBER IS:   "; SMALL_INT
00520       PRINT "THE SINE OF ";ANGLE;" DEGREES IS: "; SINE_ANGLE
00530       PRINT
00540 ELSE
00550       PRINT
00560       PRINT "ERROR OCCURRED IN SUBROUTINE GENERAL."
00570       PRINT
00580 END IF
00590
00600 BYE
00610 END
```

DG-25240

Figure 6-10. Program TEST_GENERAL.BASIC

The AOS/VS BASIC software includes the BASIC interpreter as file BASIC.PR. The software also includes file BASICASM.SR and BASICLINK.CLI. You must make changes to these files and create a new BASIC.PR before executing TEST_GENERAL.BASIC. Proceed as follows.

1. Ask your system manager for the locations of these files. It's likely that BASIC.PR is in a *public* directory such as :UTIL. Then, BASICASM.SR and BASICLINK.CLI are likely in a directory set aside for the BASIC software that Data General has released to your installation. Thus, you could well have to make this latter directory your working directory and move the BASIC.PR file you create there to the public directory.
2. Edit BASICASM.SR. After the line

   ```
   .ENT  BASICASM
   ```

   add the line

   ```
   .EXTL  GENERAL
   ```

   Also, after the line

   ```
   BASICASM:
   ```

   add the two lines

   ```
   .TXT  'GENERAL'
   GENERAL
   ```

3. Assemble BASICASM.SR.
4. If you haven't already done so, compile GENERAL.F77 to obtain GENERAL.OB. You must be sure that BASICLINK.CLI has access to GENERAL.OB.
5. Use the macro BASICLINK.CLI to create a new BASIC.PR that contains instructions from GENERAL.OB. The specific command is

   ```
   BASICLINK GENERAL
   ```

   BASICLINK invokes Link; in turn, Link could take several minutes to build BASIC.PR.
6. Execute BASIC.PR and then have it run program TEST_GENERAL.BASIC. The CLI command to do these things is

   ```
   XEQ BASIC TEST_GENERAL.BASIC
   ```

   You will see the following output on your console.

   AOS/VS BASIC Revision ...
   *THE LARGEST REAL*4 NUMBER IS:* *8.94*
   *THE SMALLEST INTEGER*2 NUMBER IS:* *-2846*
   THE SINE OF 30 DEGREES IS: .5

   AOS/VS BASIC terminated on ...

C and F77

This section lists F77 data types and their C correspondents. It also shows the C program TEST_GENERAL.C, that calls subroutine GENERAL.F77.

F77 and C Data Types

F77	C
INTEGER*2	int or short int
INTEGER*4	long int
REAL*4	float or short float
REAL*8 and DOUBLE PRECISION	double or long float
COMPLEX	None
COMPLEX*16 and DOUBLE PRECISION COMPLEX	None
LOGICAL*2	None
LOGICAL*4	None
CHARACTER*1	char
CHARACTER*N ("N" is a constant.)	None
CHARACTER*(*)	None

Sample Program

Program TEST_GENERAL.C calls subroutine GENERAL. This program's listing is shown in Figure 6-11.

```
/* Program TEST_GENERAL to test subroutine GENERAL */

#nolist
#include <stdio.h>
#include <math.h>
#list

main ()

{

float           reals[] = {  3.40,  8.61, -6.00,  8.94,  4.18,
                             7.56, -9.57,  0.00, -1.24,  0.52  };
short int       r_size = 10;

short int       ints[] = {  386, -2846, 3091, -33, 5104  };
short int       i_size = 5;

float           angle = 30.0;

float           big_real;
short           int small_int;
float           sine_angle;
short           ier;

$fortran void GENERAL();

/* Next call subroutine GENERAL.  "GENERAL" must be capitalized for  */
/*     Link and the non-array arguments must be pointers.            */

GENERAL (reals, &r_size, ints, &i_size, &angle,
         &big_real, &small_int, &sine_angle, &ier);

if ( ier == 0 )
{
        printf ("\n");
        printf ("THE LARGEST REAL*4 NUMBER IS:       %f\n", big_real);
        printf ("THE SMALLEST INTEGER*2 NUMBER IS:   %hd\n", small_int);
        printf ("THE SINE OF %f DEGREES IS: %f\n", angle, sine_angle);
}
else
{
        printf ("\n");
        printf ("ERROR OCCURRED IN SUBROUTINE GENERAL.\n");
        printf ("\n");
}

}
```

DG-25241

Figure 6-11. Program TEST_GENERAL.C

Assume that you have the directory with the C software on your searchlist and that you have compiled GENERAL.F77 to create GENERAL.OB. Then, use the following commands to compile, link, and execute TEST_GENERAL.C.

```
CC TEST_GENERAL
CCL TEST_GENERAL GENERAL
XEQ TEST_GENERAL
```

The output from the execution of TEST_GENERAL.PR is

```
THE LARGEST REAL*4 NUMBER IS:            8.939999
THE SMALLEST INTEGER*2 NUMBER IS:       -2846
THE SINE OF 30.000000 DEGREES IS:       0.500000
```

COBOL and F77

F77 normally creates *word addresses* for arguments when it compiles a main program or any subprogram. COBOL always creates *byte addresses* for its arguments. Recall from the "VS/ECS Calling Conventions" section of this chapter that F77 creates a byte address for CHARACTER arguments. Thus, it's necessary to rewrite GENERAL.F77 so it contains only CHARACTER arguments. The corresponding COBOL arguments must be declared PIC X(); then both compilers will create byte addresses to strings of ASCII characters.

We'll call the new subroutine GENERAL1.F77. It contains statements to convert CHARACTER arguments to numbers and vice versa by using internal files. Furthermore, the CHARACTER arguments in GENERAL1 must be declared fixed-length because the length of such an argument cannot be an argument itself. Why not? Consider the following statements.

```
1       SUBROUTINE WHYNOT (ARRAY, SIZE,     NAME, LENGTH)
2
3       INTEGER*4 SIZE
4       REAL*4 ARRAY(SIZE)
5
6       CHARACTER*2 LENGTH
7       CHARACTER*(LENGTH) NAME
8
9       PRINT *, 'LENGTH IS ', LENGTH
10      RETURN
11      END
8
```

F77 allows statements 3 and 4, but disallows statements 6 and 7. Furthermore,

```
CHARACTER*(*) NAME
```

will not work here because of the dope vectors F77 places on the stack. Also, you cannot replace line 6 by

```
INTEGER*2 LENGTH
```

because COBOL creates a byte address for LENGTH's corresponding argument in contrast to F77's word address.

F77 and COBOL Data Types

No table of F77 and COBOL data types appears here. For example, it's true that FORTRAN 77's internal storage of REAL*8 data is exactly the same as that of COBOL's COMPUTATIONAL-2 data. However, you have seen that *all* COBOL/F77 interprogram communication must be via a PIC X(N)/CHARACTER*N argument correspondence, where N is an integer constant (between 1 and 32767). Thus, a F77/COBOL data type table is useless.

Sample Program Units

Subroutine subprogram GENERAL1.F77 appears in Figure 6-12. It may not be as general as you would like, but COBOL's byte pointer convention for arguments and F77's creation of dope vectors for CHARACTER arguments forces GENERAL1.F77 to have specific declarations such as

```
CHARACTER*30 INT_ARRAY_RECORD
INTEGER*2 INT_ARRAY(5)
INTEGER*2 INT_SIZE  / 5 /
```

Let's trace some data through this subroutine. We'll choose bytes 10 through 18 of CHARACTER variable REAL_ARRAY_RECORD. GENERAL1 receives the address of REAL_ARRAY_RECORD from TEST_GENERAL1. (The 90 bytes of this CHARACTER variable are known to TEST_GENERAL1 as REALS-AS-CHARACTERS.)

1. The characters in these 9 bytes are " 8.94E+00".
2. The statement

   ```
   READ (REAL_ARRAY_RECORD, 10) REAL_ARRAY
   ```

   converts these 9 bytes to a single-precision floating-point number (4-bytes long) in REAL_ARRAY(2). Its value is 8.94.
3. The 8DO 408 loop results in LARGEST_REAL containing exactly the same 4 bytes as REAL_ARRAY(2).
4. The statement

   ```
   WRITE (LARGEST_REAL_RECORD, 60) LARGEST_REAL
   ```

   places " .894E+01" into the 9 bytes of LARGEST_REAL_RECORD. There is no practical way to force F77 to place " 8.94E+00" into LARGEST_REAL_RECORD.
5. When TEST_GENERAL1 regains control, it accesses these 9 bytes via the name BIG-REAL-AS-CHARACTERS.

Figure 6-13 contains COBOL program TEST_GENERAL1.CO.

```
      SUBROUTINE GENERAL1 (REAL_ARRAY_RECORD, INT_ARRAY_RECORD,
     1                     ANGLE_RECORD,
     2                     LARGEST_REAL_RECORD, SMALLEST_INT_RECORD,
     3                     SINE_ANGLE_RECORD, ERROR_RECORD)

C     This version of subroutine GENERAL is for calling by, and only
C        by, a COBOL program.  The caller passes and expects only
C        DISPLAYable arguments.  Thus, F77 must extract REAL and INTEGER
C        values from the the CHARACTER arguments it receives, then F77
C        must return its results as CHARACTER arguments.  This
C        subroutine uses internal files to convert CHARACTER variables
C        to REAL and INTEGER variables, and vice versa.

      CHARACTER*90 REAL_ARRAY_RECORD
      REAL*4 REAL_ARRAY(10)
      INTEGER*2 REAL_SIZE  / 10 /

      CHARACTER*30 INT_ARRAY_RECORD
      INTEGER*2 INT_ARRAY(5)
      INTEGER*2 INT_SIZE  / 5 /

      CHARACTER*9 ANGLE_RECORD
      REAL*4 ANGLE


      CHARACTER*9 LARGEST_REAL_RECORD
      REAL*4 LARGEST_REAL

      CHARACTER*6 SMALLEST_INT_RECORD
      INTEGER*2 SMALLEST_INT

      CHARACTER*9 SINE_ANGLE_RECORD
      REAL*4 SINE_ANGLE

      CHARACTER*5 ERROR_RECORD
      INTEGER*2 ERROR


      ERROR = 0          ! There's no error in the array sizes
C                             because they are fixed length.

C     Extract <REAL_ARRAY> from the string of ASCII characters in
C            <REAL_ARRAY_RECORD>.
      READ (REAL_ARRAY_RECORD, 10) REAL_ARRAY
   10 FORMAT (10E9.2)

C     Extract <INT_ARRAY> from the string of ASCII characters in
C            <INT_ARRAY_RECORD>.
      READ (INT_ARRAY_RECORD, 20) INT_ARRAY
   20 FORMAT (5I6)

C     Extract <ANGLE> from the string of ASCII characters in
C            <ANGLE_RECORD>.
      READ (ANGLE_RECORD, 30) ANGLE
   30 FORMAT (E9.2)
```
DG-25242

Figure 6-12. Subroutine Subprogram GENERAL1.F77 (continues)

```
C       Find the largest element in <REAL_ARRAY> and place it in
C          <LARGEST_REAL>.
        LARGEST_REAL = REAL_ARRAY(1)
        DO 40 I = 2, REAL_SIZE
                IF ( REAL_ARRAY(I) .GT. LARGEST_REAL )
     1               LARGEST_REAL = REAL_ARRAY(I)
   40   CONTINUE

C       Find the smallest element in <INT_ARRAY> and place it in
C          <SMALLEST_INT>.
        SMALLEST_INT = INT_ARRAY(1)
        DO 50 I = 2, INT_SIZE
                IF ( INT_ARRAY(I) .LT. SMALLEST_INT )
     1               SMALLEST_INT = INT_ARRAY(I)
   50   CONTINUE

C       Compute the sine of <ANGLE> after converting <ANGLE> from degrees
C          to radians.
        SINE_ANGLE = SIN(3.141593*ANGLE/180.0) ! pi radians = 180 degrees.

C       Place <LARGEST_REAL> into LARGEST_REAL_RECORD as a string of
C               ASCII characters.
        WRITE (LARGEST_REAL_RECORD, 60) LARGEST_REAL
   60   FORMAT (E9.3)

C       Place <SMALLEST_INT> into SMALLEST_INT_RECORD as a string of
C               ASCII characters.
        WRITE (SMALLEST_INT_RECORD, 70) SMALLEST_INT
   70   FORMAT (I6)

C       Place <SINE_ANGLE> into SINE_ANGLE_RECORD as a string of
C               ASCII characters.
        WRITE (SINE_ANGLE_RECORD, 80) SINE_ANGLE
   80   FORMAT (F9.7)

C       Place <ERROR> into ERROR_RECORD as a string of
C               ASCII characters.
        WRITE (ERROR_RECORD, 90) ERROR
   90   FORMAT (I5)

C       Done!
        RETURN
        END
```

DG-25242

Figure 6-12. Subroutine Subprogram GENERAL1.F77 (concluded)

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  TEST-GENERAL1.

ENVIRONMENT DIVISION.

DATA DIVISION.

WORKING-STORAGE SECTION.

77      ANGLE-AS-CHARACTERS        PIC X(9)  VALUE IS ' 3.00E+01'.
77      BIG-REAL-AS-CHARACTERS     PIC X(9).
77      SMALL-INT-AS-CHARACTERS    PIC X(6).
77      SINE-ANGLE-AS-CHARACTERS   PIC X(9).
77      IER-AS-CHARACTERS          PIC X(5).

01      REALS-AS-CONSTANTS.
        05  REAL-1   USAGE IS COMPUTATIONAL-2  VALUE IS  3.40.
        05  REAL-2   USAGE IS COMPUTATIONAL-2  VALUE IS  8.61.
        05  REAL-3   USAGE IS COMPUTATIONAL-2  VALUE IS -6.00.
        05  REAL-4   USAGE IS COMPUTATIONAL-2  VALUE IS  8.94.
        05  REAL-5   USAGE IS COMPUTATIONAL-2  VALUE IS  4.18.
        05  REAL-6   USAGE IS COMPUTATIONAL-2  VALUE IS  7.56.
        05  REAL-7   USAGE IS COMPUTATIONAL-2  VALUE IS -9.57.
        05  REAL-8   USAGE IS COMPUTATIONAL-2  VALUE IS  0.00.
        05  REAL-9   USAGE IS COMPUTATIONAL-2  VALUE IS -1.24.
        05  REAL-10  USAGE IS COMPUTATIONAL-2  VALUE IS  0.52.

01      INTS-AS-CONSTANTS.
        05  INT-1    USAGE IS COMPUTATIONAL   PIC IS S9999   VALUE IS   386.
        05  INT-2    USAGE IS COMPUTATIONAL   PIC IS S9999   VALUE IS -2846.
        05  INT-3    USAGE IS COMPUTATIONAL   PIC IS S9999   VALUE IS  3091.
        05  INT-4    USAGE IS COMPUTATIONAL   PIC IS S9999   VALUE IS   -33.
        05  INT-5    USAGE IS COMPUTATIONAL   PIC IS S9999   VALUE IS  5014.

01      REALS-AS-CHARACTERS.
        05  REAL-1   PIC IS +9.99E+99.
        05  REAL-2   PIC IS +9.99E+99.
        05  REAL-3   PIC IS +9.99E+99.
        05  REAL-4   PIC IS +9.99E+99.
        05  REAL-5   PIC IS +9.99E+99.
        05  REAL-6   PIC IS +9.99E+99.
        05  REAL-7   PIC IS +9.99E+99.
        05  REAL-8   PIC IS +9.99E+99.
        05  REAL-9   PIC IS +9.99E+99.
        05  REAL-10  PIC IS +9.99E+99.

01      INTS-AS-CHARACTERS.
        05  INT-1    PIC IS -----9.
        05  INT-2    PIC IS -----9.
        05  INT-3    PIC IS -----9.
        05  INT-4    PIC IS -----9.
        05  INT-5    PIC IS -----9.
```

DG-25243

Figure 6-13. Program TEST_GENERAL1.CO (continues)

```
PROCEDURE DIVISION.

        MOVE CORRESPONDING REALS-AS-CONSTANTS TO REALS-AS-CHARACTERS.
        MOVE CORRESPONDING  INTS-AS-CONSTANTS TO  INTS-AS-CHARACTERS.

*       Here we go ...
        CALL 'GENERAL1' USING REALS-AS-CHARACTERS,
                              INTS-AS-CHARACTERS,
                              ANGLE-AS-CHARACTERS,

                              BIG-REAL-AS-CHARACTERS,
                              SMALL-INT-AS-CHARACTERS,
                              SINE-ANGLE-AS-CHARACTERS,
                              IER-AS-CHARACTERS.

        IF IER-AS-CHARACTERS IS NOT EQUAL TO '     0'
                DISPLAY " "
                DISPLAY "ERROR OCCURRED IN SUBROUTINE GENERAL1."
                DISPLAY " "
        ELSE
                DISPLAY " "
                DISPLAY "THE LARGEST REAL*4 NUMBER IS:       ",
                                BIG-REAL-AS-CHARACTERS
                DISPLAY "THE SMALLEST INTEGER*2 NUMBER IS:  ",
                                SMALL-INT-AS-CHARACTERS
                DISPLAY "THE SINE OF ", ANGLE-AS-CHARACTERS,
                                " DEGREES IS: ", SINE-ANGLE-AS-CHARACTERS
                DISPLAY " ".

        STOP RUN.
```

DG-25243

Figure 6-13. Program TEST_GENERAL1.CO (concluded)

Note that the program converts all its COMPUTATIONAL-2 (i.e., REAL*8) numbers in REALS-AS-CONSTANTS to strings of ASCII characters (i.e., CHARACTER*9) in REALS-AS-CHARACTERS. For example, the compiler converts the four ASCII characters "8.94" to a REAL*8 number whose name is REAL-2 OF REALS-AS-CONSTANTS. The runtime routines convert this REAL*8 number to the 9 bytes " 8.94E+00" in REAL-2 OF REALS-AS-CHARACTERS as a result of the statement

```
MOVE CORRESPONDING REALS-AS-CONSTANTS TO REALS-AS-CHARACTERS.
```

You have seen that subroutine GENERAL1 receives access to this CHARACTER data item " 8.94E+00".

Note also that TEST_GENERAL1 receives the results of GENERAL1's efforts as PIC X (i.e., CHARACTER) data. IER-AS-CHARACTERS is an example.

Similarly, the PIC S9999 (i.e., INTEGER*2) data go through similar transformations. The COBOL compiler transforms the 3 bytes "386" into the binary digits 00000000000000000000000110000010 in the 4-byte area whose name is INT-1 OF INTS-AS-CONSTANTS. At runtime the statement

```
MOVE CORRESPONDING INTS-AS-CONSTANTS TO INTS-AS-CHARACTERS.
```

results in the 6 bytes "   386" being placed in INT-1 OF INTS-AS-CHARACTERS. Of course, GENERAL1 has access to these 6 bytes; it may refer to them by INT_ARRAY_RECORD(1:6). GENERAL1 converts these 6 bytes to an INTEGER*2 number via its

```
READ (INT_ARRAY_RECORD, 20) INT_ARRAY
```

statement.

Assume that you have the directory with the COBOL software on your searchlist and that you have compiled GENERAL1.F77 to create GENERAL1.OB. Then, use the following commands to compile, link, and execute TEST_GENERAL1.CO.

```
COBOL/O=TEST_GENERAL1.OB TEST_GENERAL1.CO
CLINK TEST_GENERAL1 GENERAL1 F77NOPCT.OB F77IO.LB/OVER F77ENV.LB F77MT.LB
XEQ TEST_GENERAL1
```

The FORTRAN libraries and F77NOPCT.OB must appear in the CLINK command because the instructions in GENERAL1.F77 result in calls to subroutines that are in the library files and to subroutine F77NOPCT.

The output from the execution of TEST_GENERAL1.PR is

*THE LARGEST REAL*4 NUMBER IS:* *.894E+01*
*THE SMALLEST INTEGER*2 NUMBER IS:* *-2846*
THE SINE OF 3.00E+01 DEGREES IS: *.5000000*

NOTE: COBOL considers data items declared as COMPUTATIONAL-1 or COMPUTATIONAL-2 identically (i.e., as double-precision floating-point). In other words, COMPUTATIONAL-1 data items are *not* single-precision floating-point numbers.

PASCAL and F77

This section lists F77 data types and their PASCAL correspondents. It also shows the PASCAL program TEST_GENERAL.PAS, that calls subroutine GENERAL.F77.

F77 and PASCAL Data Types

F77	PASCAL
INTEGER*2	SHORT_INTEGER or INTEGER with the /INTEGER=SHORT compiler switch
INTEGER*4	LONG_INTEGER or INTEGER with no /INTEGER compiler switch or INTEGER with the /INTEGER =LONG compiler switch
REAL*4	REAL
REAL*8 and DOUBLE PRECISION	DOUBLE_REAL
COMPLEX	None
COMPLEX*16 and DOUBLE PRECISION COMPLEX	None
LOGICAL*2	None — But, a PASCAL SHORT_INTEGER variable whose value is 0 or -1 is the same as a respective F77 LOGICAL*2 variable whose value is .FALSE. or .TRUE.
LOGICAL*4	None — But, a PASCAL LONG_INTEGER variable whose value is 0 or -1 is the same as a respective F77 LOGICAL*4 variable whose value is .FALSE. or .TRUE.
CHARACTER*1	CHAR
CHARACTER*N ("N" is a constant.)	PACKED ARRAY [1..N] OF CHAR

Sample Program

Program TEST_GENERAL.PAS calls subroutine GENERAL. This program's listing is shown in Figure 6-14.

```
program TEST_GENERAL(input,output);

type
        REAL_ARRAY=     array [1..10] of real;
        INT_ARRAY=      array [1..5] of short_integer;

var
        REALS:          REAL_ARRAY;
        R_SIZE:         short_integer;
        INTS:           INT_ARRAY;
        I_SIZE:         short_integer;
        ANGLE:          real;
        BIG_REAL:       real;
        SMALL_INT:      short_integer;
        SINE_ANGLE:     real;
        IER:            short_integer;

external procedure GENERAL( REALS:          REAL_ARRAY;
                            R_SIZE:         short_integer;
                            INTS:           INT_ARRAY;
                            I_SIZE:         short_integer;
                            ANGLE:          real;

                            var BIG_REAL:   real;
                            var SMALL_INT:  short_integer;
                            var SINE_ANGLE: real;
                            var IER:        short_integer
                                                           );


begin
    REALS[1]:=  3.40;
    REALS[2]:=  8.61;
    REALS[3]:= -6.00;
    REALS[4]:=  8.94;
    REALS[5]:=  4.18;
    REALS[6]:=  7.56;
    REALS[7]:= -9.57;
    REALS[8]:=  0.00;
    REALS[9]:= -1.24;
    REALS[10]:= 0.52;
    R_SIZE:= 10;

    INTS[1]:=   386;
    INTS[2]:= -2846;
    INTS[3]:=  3091;
    INTS[4]:=   -33;
    INTS[5]:=  5104;
    I_SIZE:= 5;

    ANGLE:= 30.0;
```

DG-25244

Figure 6-14. Program TEST_GENERAL.PAS (continues)

```
{Here we go ...}
GENERAL(REALS,R_SIZE,INTS,I_SIZE,ANGLE,
              BIG_REAL,SMALL_INT,SINE_ANGLE,IER);

if IER = 0 then begin
    writeln;
    writeln('THE LARGEST REAL*4 NUMBER IS:        ', {"5:2" = "F5.2"}
                                             BIG_REAL:5:2);
    writeln('THE SMALLEST INTEGER*2 NUMBER IS:   ',
                                             SMALL_INT);
    writeln('THE SINE OF ',ANGLE:5:2,' DEGREES IS: ',
                                             SINE_ANGLE:5:2);
    writeln
    end

else begin
    writeln;
    writeln('ERROR OCCURRED IN SUBROUTINE GENERAL.');
    writeln
    end

end.
```

DG-25244

Figure 6-14. Program TEST_GENERAL.PAS (concluded)

Assume that you have the directory with the PASCAL software on your searchlist and that you have compiled GENERAL.F77 to create GENERAL.OB. Then, use the following commands to compile, link, and execute TEST_GENERAL.PAS.

```
PASCAL TEST_GENERAL
PASLINK TEST_GENERAL GENERAL
XEQ TEST_GENERAL
```

The output from the execution of TEST_GENERAL.PR is

*THE LARGEST REAL*4 NUMBER IS: 8.94*
*THE SMALLEST INTEGER*2 NUMBER IS: -2846*
THE SINE OF 30.00 DEGREES IS: 0.50

PL/I and F77

This section lists F77 data types and their PL/I correspondents. It also shows the PL/I program TEST_GENERAL.PL1, that calls subroutine GENERAL.F77.

F77 and PL/I Data Types

F77	PL/I
INTEGER*2	FIXED BIN(1) through FIXED BIN(15)
INTEGER*4	FIXED BIN(16) through FIXED BIN(31)
REAL*4	FLOAT BIN(1) through FLOAT BIN(21)
REAL*4	FLOAT DEC(1) through FLOAT DEC(6)
REAL*8 and DOUBLE PRECISION	FLOAT BIN(22) through FLOAT BIN(53)
REAL*8 and DOUBLE PRECISION	FLOAT DEC(7) through FLOAT DEC(16)
COMPLEX	None
COMPLEX*16 and DOUBLE PRECISION COMPLEX	None
LOGICAL*2	ALIGNED BIT(16) variable, which is always either "0000"B4 or "FFFF"B4; or FIXED BIN(15) variable, which is always either 0 or -1
LOGICAL*2	ALIGNED BIT(32) variable, which is always either "00000000"B4 or "FFFFFFFF"B4; or FIXED BIN(31) variable, which is always either 0 or -1
CHARACTER*N ("N" is a constant.)	CHAR(N)
CHARACTER*(*)	CHAR(*)

Sample Program

Program TEST_GENERAL.PL1 calls subroutine GENERAL. This program's listing is shown in Figure 6-15.

```
TEST_GENERAL:
        PROCEDURE;
        DECLARE REALS(10)       FLOAT BINARY(15) STATIC INIT (
                                     3.40,  8.61, -6.00,  8.94, 4.18,
                                     7.56, -9.57,  0.00, -1.24, 0.52 ),

                R_SIZE          FIXED BINARY(15) STATIC INIT(10),

                INTS(5)         FIXED BINARY(15) STATIC INIT (
                                     386, -2846, 3091, -33, 5104 ),

                I_SIZE          FIXED BINARY(15) STATIC INIT(5),

                ANGLE           FLOAT BINARY(15) STATIC INIT(30),

                BIG_REAL        FLOAT BINARY(15),
                SMALL_INT       FIXED BINARY(15),
                SINE_ANGLE      FLOAT BINARY(15),
                IER             FIXED BINARY(15),

                GENERAL         ENTRY((10) FLOAT BIN(15), /* REALS    */
                                           FIXED BIN(15), /* R_SIZE   */
                                       (5) FIXED BIN(15), /* INTS     */
                                           FIXED BIN(15), /* I_SIZE   */
                                           FLOAT BIN(15), /* ANGLE    */
                                           FLOAT BIN(15), /* BIG_REAL */
                                           FIXED BIN(15), /* SMALL_INT*/
                                           FLOAT BIN(15), /*SINE_ANGLE*/
                                           FIXED BIN(15)  ),   /* IER */

                @OUTPUT         FILE;

        OPEN FILE(@OUTPUT) STREAM OUTPUT PRINT;

        /* Here we go ... */

        CALL GENERAL( REALS, R_SIZE, INTS, I_SIZE, ANGLE,
                      BIG_REAL, SMALL_INT, SINE_ANGLE, IER);

        IF IER = 0 THEN
                DO;
                        PUT FILE(@OUTPUT) SKIP LIST (" ");
                        PUT FILE(@OUTPUT) SKIP EDIT(
                                "THE LARGEST REAL*4 NUMBER IS:       ",
                                BIG_REAL ) (A, F(5,2));
                        PUT FILE(@OUTPUT) SKIP EDIT(
                                "THE SMALLEST INTEGER*2 NUMBER IS: ",
                                SMALL_INT ) (A, F(5));
                        PUT FILE(@OUTPUT) SKIP EDIT(
                                "THE SINE OF ", ANGLE, " DEGREES IS: ",

                                SINE_ANGLE ) (A, F(5,1), A, F(7,4));
                        PUT FILE(@OUTPUT) SKIP LIST (" ");
                END;
```

DG-25245

Figure 6-15. Program TEST_GENERAL.PL1 (continues)

```
            ELSE
            DO;
                    PUT FILE(@OUTPUT) SKIP LIST (" ");
                    PUT FILE(@OUTPUT) SKIP LIST (
                            "ERROR OCCURRED IN SUBROUTINE GENERAL.");
                    PUT FILE(@OUTPUT) SKIP LIST (" ");

            END;

    STOP;

    END;    /* OF PROGRAM TEST_GENERAL */
```

DG-25245

Figure 6-15. Program TEST_GENERAL.PL1 (concluded)

Assume that you have the directory with the PL/I software on your searchlist and that you have compiled GENERAL.F77 to create GENERAL.OB. Then, use the following commands to compile, link, and execute TEST_GENERAL.PL1.

```
PL1 TEST_GENERAL
PL1LINK TEST_GENERAL GENERAL
XEQ TEST_GENERAL
```

The output from the execution of TEST_GENERAL.PR is

*THE LARGEST REAL*4 NUMBER IS:*	*8.94*
*THE SMALLEST INTEGER*2 NUMBER IS:*	*-2846*
THE SINE OF 30.0 DEGREES IS:	*0.5000*

End of Chapter

Chapter 7
Programming Hints

This chapter presents several diverse topics that may help you implement F77 programs. The topics are as follows.

- The F77 Error File
- Improving Program Readability
- Program Enhancements
- F77 Output and Printing Special Forms

The F77 Error File

The *FORTRAN 77 Reference Manual* explains how to incorporate and use file ERR.F77.IN in your F77 program units. It's worth repeating that use of this error file means your program works with mnemonics. These mnemonics and their corresponding text explanations never change form one revision of F77 to another. This is in possible contrast to the use of "hard-wired" constant values for error identification.

ERR.F77.IN sometimes changes with a new release of F77. You usually don't have to recompile and relink any current programs just because they %INCLUDE ERR.F77.IN. New programs should %INCLUDE the latest error file.

Improving Program Readability

Chapter 5 mentions the importance of carefully designing programs to minimize the need for subsequent debugging. You should also create programs that *other* programmers can easily understand and maintain. Just remember that few things in electronic data processing are more permanent than "temporary" programs that departed programmers have written!

Program Enhancements

This section explains:

- The effect of certain compiler switches on performance.
- Ways to improve runtime computation speed.
- Ways to improve runtime I/O speed.

Compiler Switches and Program Performance

Compiler options can heavily influence F77 program performance. Some options depend on others, and selecting one could reduce the impact of others. The options could affect:

- The compilation time.
- The ability of the compiler to optimize.
- The disk space needed by compiler-generated files.
- The memory needed at runtime.
- The execution time.

The most significant effects of the compiler switches are:

/DEBUG
: slows the compilation because of the extra information it makes for the SWAT debugger. The generated code can't carry certain values in the accumulators from one statement to the next. Instead, the code must store newly computed values in memory at the end of some statements. Chapter 5 has shown you the convenience of using the SWAT debugger. Once you have used it to locate bugs, then recompile without this switch (delete any leftover .DL and .DS files) and relink to create a faster executing program file.

/DOTRIP=1
: generates code which is slightly more efficient than /DOTRIP=0. Be certain that the program logic will work correctly with this switch before using it.

/LINEID
: adds extra instructions to the generated code. Each source statement results in extra code to update specific locations to reflect the number of the source statement in the listing file. Not only do these extra instructions increase the execution time of the program file, but they might prevent the optimizer from doing a thorough job.

/PROCID
: adds extra instructions to the program file to allow the program to keep track of the currently executing procedure. If you want top performance, don't use it.

/SAVEVARS
: is often required to make programs from other vendors produce correct results, or sometimes even to run at all. Many non-DG FORTRANs provide static (nonstack) storage of variables by default. The result is that the program can subtly depend on such features as having uninitialized variables containing zero, and preserving the values of local variables in subprograms from one CALL or function reference to the next. The /SAVEVARS switch provides this preservation in F77; so does the SAVE statement. However, neither has uninitialized variables contain zero.

: There is another *potential* effect of the /SAVEVARS option: some program algorithms (most often those involving large amounts of subscript manipulation), can cause the generated code to "run out of accumulators." That is, the code must go to great lengths to free the resource called an "index register" (AC2 and AC3). If this "running out" occurs, /SAVEVARS (or SAVE) has the compiler allocate specific memory addresses, thus allowing faster calculation of offsets and less conflict among accumulator usage.

: There is no definite way to predict whether or not static allocation of variables will help a given program. You must experiment in each case.

/SUB
: has the compiler insert extra instructions in the generated code. Each time the code evaluates a subscript or substring expression and calculates the actual offset into the array or string, it also compares the offset to the appropriate limit. This comparison takes time, and also reduces the optimizer's ability to use the accumulators for storing data and expression values.

Usually, the simple compilation command line

```
F77/OPT your_program_name
```

produces the best code (and a longer compilation time). Sometimes adding /SAVEVARS or /DOTRIP=1 (or both) can produce better code.

Enhancing Computational Speed

Once you have selected compiler options to increase runtime performance of a debugged program, consider the effects of computation at runtime. This section gives tips and techniques to speed up computations.

First, integer arithmetic is faster than single-precision arithmetic, which is faster than double-precision arithmetic.

Second, truncation during floating-point operations is slightly faster and slightly more inaccurate than rounding. Truncation appears to be very common in the industry and programmers have lived with it for years. You select truncation or rounding at link time. The F77LINK macro selects rounding by default.

Third, you improve compilation and execution speed by running on an idle system with lots of physical memory and a large working set.

Enhancing I/O Speed

Data General created some F77 programs whose sole purpose was to read records from a common file via different I/O statements. This file contained thousands of 100-byte ASCII data strings that were separated by NEW-LINE characters. The slowest possible access technique was used as a basis for comparison with other techniques. Its relative speed is thus 1.00. The "Result" column below gives the quotient of a technique's records/second number divided by the records/second number of the slowest technique.

File Access Technique	Result
Read the file as a data-sensitive file into an integer array using the data descriptor "100A1" for each record.	1.00
Read the file as a data-sensitive file into a character variable using the data descriptor "A100" for each record.	4.54
Read the file as a fixed file into a character variable using the data descriptor "A100" for each record.	5.01
Read the file as a fixed file into a character variable with unformatted I/O for each record.	6.84
Read the file as a dynamic file into a character variable with unformatted I/O for each record and with the default BLOCKSIZE (512).	6.72
Read the file as a dynamic file into a character variable with unformatted I/O for each record and with a BLOCKSIZE value of 32767.	6.92

NOTE: These numbers reflect operation with a particular ECLIPSE computer, operating system, peripherals, and revision of F77. Use them as guides to show how to increase I/O performance and not as guaranteed results.

Here are some general and some F77-specific approaches to consider as you try to increase I/O speed.

- Use the record format of the file to your advantage. In general, RECFM=DATASENSITIVE will give the slowest file I/O, with VARIABLE, FIXED, and DYNAMIC successively faster. You can attain the fastest possible I/O by performing unformatted reads and writes of an array with a file whose records are dynamic. In this case, I/O occurs directly from and to an array without the F77 runtime routines doing any data movement.
- Define a large BLOCKSIZE in the OPEN statement to reduce the number of file accesses required for sequentially processing a file.
- To output an array using formatted I/O, use a sequence like

```
C       SEQUENCE A
        DIMENSION IARRAY(50)
              .
              .
              .
        WRITE (10, 100) IARRAY
100     FORMAT (50I5)
```

  It is much more efficient to do an I/O operation on an entire array rather than on its individual elements. While a sequence like

```
C       SEQUENCE B
        DIMENSION IARRAY(50)
              .
              .
              .
        WRITE (10, 100) (IARRAY(I), I = 1, 50)
100     FORMAT (50I5)
```

  displays identical results, it results in about *50* system calls (one for each element of IARRAY), instead of about one system call. In other words — avoid implied DO loops for I/O whenever possible. Finally, FORMAT statement 100 in both of the above sequences is more efficient than

```
100     FORMAT (50(I5))
```

  In general, avoid FORMAT statements that have sizable repeat counts outside specifications with parentheses.

- If you have to use only a known part of an array for I/O, then (as mentioned before) try to avoid implied DO loops. Instead, use EQUIVALENCE or assignment statements to define another array whose consecutive elements are those of the known subset. For instance, assume that the respective array names are A_ARRAY and B_ARRAY so that B_ARRAY contains the necessary subset of A_ARRAY's elements. Then write a statement pair such as

```
        WRITE (10, 110) B_ARRAY
110     FORMAT (12F6.2)
```

  instead of

```
        WRITE (10, 110) (A_ARRAY(I), I = 1, 23, 2)
110     FORMAT (12F6.2)
```

- Suppose you need to use a unit number that is normally preconnected to some other file. It is faster to CLOSE the preconnected unit and to OPEN the file you want on that unit than it is to directly

OPEN the file on that unit. Why? Directly OPENing the file on the unit is actually a reOPEN of a preconnected unit that hasn't been accessed yet — and extra processing is necessary to determine if such a reOPEN refers to the name of the preconnected file or to some new file. The CLOSE statement eliminates the need for the extra processing. For example:

Faster	**Slower**
`CLOSE (6)` `OPEN (6, FILE='FOO', ...)`	`OPEN (6, FILE='FOO', ...)`

F77 Output and Printing Special Forms

Suppose your F77 program writes to a data-sensitive file and the output fincludes a form-feed character (whose octal value is <014>). When you print the file via a QPRINT command, XLPT.PR (as part of AOS/VS) sends this character to the printer which advances the paper to the next page.

At most installations:

- The printer then advances three lines and printing resumes on the fourth line.
- The printer prints only 63 lines on a page and then advances to the fourth line of the next page to resume its output.

In addition, the first response to the QPRINT command is frequently a header page (filename in large letters, pathname, times, dates, etc.) and a form feed.

You can have the printer behave differently. For example, you might want to print special forms that are not the default 66 lines long (i.e., 11 inches for a switch setting of 6 lines per inch). And, you might want printing on the first line of the form.

What software steps are necessary to change the default behavior of the printer? You must use the Forms Control Utility (FCU) program and sometimes place special nonprinting control characters in the output files your FORTRAN 77 programs create. You or your system's operator must also give specific commands to EXEC to print the special forms.

If you aren't familiar with EXEC commands to control the printer, or with FCU.PR, then read the appropriate manuals — the *Advanced Operating System/Virtual Storage (AOS/VS) Operator's Guide* and the *AOS & AOS/VS CLI User's Manual.*

The basic steps to prepare and print a file on nonstandard forms are:

- Determine the layout of the form. You have to know the first line of printing, the length and width of the form, the last line printing can occur on, and any lines that the paper should advance to by skipping to channels 2 through 11 of a vertical forms unit (VFU); i.e., a *carriage control tape.*
- Write, compile, Link, and execute the F77 program that inserts form-feed characters and VFU control characters in the output file. The *AOS/VS CLI User's Manual* explains the VFU control characters and their effects. And the output file should have data-sensitive records.
- Execute FCU.PR and describe your special form to it.
- Your system operator should:
  - Record the current LPP, CPL, and HEADERS values for the selected printer (with its VFU).
  - PAUSE the printer and change the lines-per-page (LPP) and characters-per-line (CPL) settings to the true length and width of the special form. You must have already given these same numbers when you executed FCU.PR for the form. Also, insure that the HEADERS setting is correct (frequently, zero). If you don't do this, unwanted header page information might print on at least the first form.
  - Insert and align the special forms.
  - CONTINUE the printer.

- Print (QPRINT) the output file.
- PAUSE the printer. Reset the LPP, CPL, and HEADERS settings to those of the next form.
- Remove the special forms.
- Insert and align the next forms.
- CONTINUE the printer.

Background for Two Examples

Marll is the corresponding secretary of his antique auto club. Part of his job is to keep track of members and their autos. He creates a file called MEMBERS.DATA with data-sensitive organization because programs that contain LIST-directed READ statements will read the file. These programs will create two files: MEMBERS.LABELS — for printing on address labels, and MEMBERS.CARDS — for printing on index cards. The filenames of these respective programs are PRINT_LABELS.F77 and PRINT_CARDS.F77.

Figure 7-1 contains a listing of file MEMBERS.DATA.

```
"MARLL DALRYMPLE","64 WOOSTER DRIVE"," ","FRAMINGHAM","MA","01701"
"31 FORD MODEL A PHAETON", "40 FORD CONVERTIBLE", "40 FORD COUPE"
"47 FORD 'WOODIE' WAGON", " ", " "
"GORDON CLIFFORD","501 BELKNAP ROAD","BOX 44","WAYLAND","MA","01778"
"34 FORD CABRIOLET", "35 BUICK RUMBLE SEAT COUPE", "39 PACKARD SEDAN"
"46 CHRYSLER TOWN & COUNTRY", "52 MG TD ROADSTER", " "
```

DG-25246

Figure 7-1. File MEMBERS.DATA

Example 1 — Printing Labels

Figure 7-2 contains a listing of program PRINT_LABELS. Note that one form-feed character will precede the characters for each label. The only channel skipping the printer will do while working with the labels is to channel 1 — precisely the effect of the form-feed character. The labels are 15/16 inches high by 3.5 inches wide, which is a standard size.

```
      PROGRAM   PRINT_LABELS   ! TO PREPARE FILE <MEMBERS.LABELS>
C                                 FOR PRINTING LABELS

      CHARACTER*25 NAME, ADDRESS_1, ADDRESS_2
      CHARACTER*15 CITY
      CHARACTER*2 STATE
      CHARACTER*5 ZIP
      CHARACTER*26 CARS_OWNED(6)
      INTEGER COUNT  / 0 /     ! COUNT OF LABELS PRINTED

      OPEN (2, FILE='MEMBERS.DATA', STATUS='OLD', IOINTENT='INPUT')
      OPEN (3, FILE='MEMBERS.LABELS', STATUS='FRESH', IOINTENT='OUTPUT')

 10   READ (2, *, END=60) NAME, ADDRESS_1, ADDRESS_2, CITY, STATE, ZIP
      READ (2, *) ! (CARS_OWNED(I), I = 1, 3)  READ THESE RECORDS, AND
      READ (2, *) ! (CARS_OWNED(I), I = 4, 6)     THEN IGNORE THEM.

      WRITE (3, 20) NAME
 20   FORMAT ('<FF>', A)          ! <NAME> GOES ON A NEW LABEL.
      WRITE (3, 30) ADDRESS_1
 30   FORMAT (A)
      IF ( ADDRESS_2 .NE. " " ) WRITE (3, 30) ADDRESS_2
      WRITE (3, 40) CITY, STATE
 40   FORMAT (A, 2X, A)
      WRITE (3, 50) ZIP
 50   FORMAT (10X, A)             ! INDENT ZIP CODE FOR THE POSTAL SERVICE.
      COUNT = COUNT + 1
      GO TO 10

 60   WRITE (3, 70) COUNT         ! END THE LABELS EXPLICITLY.
 70   FORMAT ('<FF>', '*** ', I4, ' LABELS PRINTED ***' )

      CLOSE (2)
      CLOSE (3)

      PRINT *, 'FILE   MEMBERS.LABELS   IS READY FOR PRINTING'

      STOP
      END
```

DG-25247

Figure 7-2. Program PRINT_LABELS

Marll executes PRINT_LABELS.PR to create MEMBERS.LABELS. He also has to execute FCU.PR to create the VFU specifications file for MEMBERS.LABELS. This file is in the User Data Area (UDA) assigned to MEMBERS.LABELS. The dialog between Marll and FCU.PR appears next.

```
) XEQ FCU ↵

AOS Forms Control Utility    Revision ...

Type 'Help' for instructions

Command ?   C ↵
Pathname ?   MEMBERS.LABELS ↵
Characters Per Line (16-255)
        [80] ?   35 ↵
Tab Stops (2-79, OR STANDARD)
        [8,16,24,32,40,48,56,64,72]
?  ↵
Form length in Lines Per Page (6-144)
        [66] ?   6 ↵
Top of Form (Channel 1) Line Number (1-6)
        [1] ?   ↵
Bottom of Form (Channel 12) Line number (1-6)
        [6] ?   ↵
VFU Tape (Line numbers 1-6, Channels 2-11, OR STANDARD)
        []
?  ↵
Output to Pathname
        [:UDD:F77:MARLL:MEMBERS.LABELS] ?    ↵

Command ?   BYE ↵

FCU terminating ...
```

Marll verifies that the VFU specifications file exists with the CLI command

```
FILESTATUS/UDA MEMBERS.LABELS
```

AOS/VS responds with

MEMBERS.LABELS   UDA

Marll and his system's operator, John, go to the operator's console (username OP) and to the printer. They perform the following steps.

1. They determine that the current LPP, CPL, and HEADERS values are 66, 80, and 1, respectively.
2. They wait for the current print queue to LPT (devicename @LPB) to complete.
3. John gives these commands to the CLI.

   ```
   CONTROL @EXEC PAUSE @LPB
   CONTROL @EXEC LPP @LPB 6
   CONTROL @EXEC CPL @LPB 35
   CONTROL @EXEC HEADERS @LPB 0
   ```

4. They insert and align the labels in their Model 4216 printer.
5. John gives these commands to the CLI.

   ```
   CONTROL @EXEC CONTINUE @LPB
   QPRINT :UDD:F77:MARLL:MEMBERS.LABELS
   CONTROL @EXEC PAUSE @LPB
   CONTROL @EXEC LPP @LPB 66
   CONTROL @EXEC CPL @LPB 80
   CONTROL @EXEC HEADERS @LPB 1
   ```

6. They remove the labels and reinsert standard 11-inch high paper.
7. John gives the command

   ```
   CONTROL @EXEC CONTINUE @LPB
   ```

   to finish the restoration of the printer to its previous settings.

Example 2 — Printing Index Cards

Figure 7-3 contains a printed index card. Specifically

- Its height is 3 inches (= 18 lines) and its width is 5 inches (= 50 characters).
- Marll wants printing to begin on the second line of the form.
- Marll wants the printer to advance each card as quickly as possible from the name/address area of the form to line 10 before printing the cars a member owns. He arbitrarily chooses channel 4 of the electronic carriage control tape to correspond to line 10.

```
    --------------------------------------------------
 1 |                                                  |
 2 |  GORDON CLIFFORD                                 |
 3 |  501 BELKNAP ROAD                                |
 4 |  BOX 44                                          |
 5 |  WAYLAND      MA 01778                           |
 6 |                                                  |
 7 |                                                  |
 8 |                                                  |
 9 |                                                  |
10 |  34 FORD CABRIOLET                               |
11 |  35 BUICK RUMBLE SEAT COUPE                      |
12 |  39 PACKARD SEDAN                                |
13 |  46 CHRYSLER TOWN & COUNTRY                      |
14 |  52 MG TD ROADSTER                               |
15 |                                                  |
16 |                                                  |
17 |                                                  |
18 |                                                  |
    --------------------------------------------------
```

DG-00114

Figure 7-3. A Typical Index Card

Figure 7-4 contains a listing of program PRINT_CARDS. Note that one form-feed character will precede the characters for each card. The printer must skip to channel 1 while working with the cards; the form-feed characters in FORMAT statements 20 and 80 accomplish this. The 2 bytes <022><103> in FORMAT statement 50, along with the proper execution of FCU.PR, cause the printer to advance a card to its line 10. The "$" character is in statement 50 to prevent the issuance of a NEWLINE character (<12>) and the resulting advance of an index card to line 11 for the printing of the first antique auto's information.

```
        PROGRAM   PRINT_CARDS    ! TO PREPARE FILE <MEMBERS.CARDS>
C                                  FOR PRINTING OF INDEX CARDS

        CHARACTER*25 NAME, ADDRESS_1, ADDRESS_2
        CHARACTER*15 CITY
        CHARACTER*2 STATE
        CHARACTER*5 ZIP
        CHARACTER*26 CARS_OWNED(6)
        INTEGER COUNT  / 0 /     ! COUNT OF CARDS PRINTED

        OPEN (2, FILE='MEMBERS.DATA', STATUS='OLD', IOINTENT='INPUT')
        OPEN (3, FILE='MEMBERS.CARDS', STATUS='FRESH', IOINTENT='OUTPUT')

   10   READ (2, *, END=70) NAME, ADDRESS_1, ADDRESS_2, CITY, STATE, ZIP
        READ (2, *) (CARS_OWNED(I), I = 1, 3)
        READ (2, *) (CARS_OWNED(I), I = 4, 6)

        WRITE (3, 20) NAME
   20   FORMAT ('<FF>', A)         ! <NAME> GOES ON A NEW LABEL.
        WRITE (3, 30) ADDRESS_1
   30   FORMAT (A)
        IF ( ADDRESS_2 .NE. "  " ) WRITE (3, 30) ADDRESS_2
        WRITE (3, 40) CITY, STATE, ZIP
   40   FORMAT (A, 2X, A, 2X, A)

C       SKIP TO LINE 10 (THAT IS, CHANNEL 4 OF THE VFU "TAPE") ...
        WRITE (3, 50)
   50   FORMAT ('<022><103>', $)

C       ... AND PRINT THE CARS THE MEMBER OWNS.
        DO 60 I = 1, 6
                IF ( CARS_OWNED(I) .NE. " " ) WRITE (3, 30) CARS_OWNED(I)
   60   CONTINUE
        COUNT = COUNT + 1
        GO TO 10

   70   WRITE (3, 80) COUNT        ! END THE CARDS EXPLICITLY.
   80   FORMAT ('<FF>', '*** ', I4, ' CARDS PRINTED ***' )

        CLOSE (2)
        CLOSE (3)

        PRINT *, 'FILE   MEMBERS.CARDS   IS READY FOR PRINTING'

        STOP
        END
```

DG-25248

Figure 7-4. Program PRINT_CARDS

Marll executes PRINT_CARDS.PR to create MEMBERS.CARDS. He also has to execute FCU.PR to create the VFU specifications file for MEMBERS.CARDS. The dialog between Marll and FCU.PR appears next.

```
) XEQ FCU ↵

AOS Forms Control Utility    Revision ...

Type 'Help' for instructions

Command ?  C ↵
Pathname ?  MEMBERS.CARDS ↵
Characters Per Line (16-255)
        [80] ?  50 ↵
Tab Stops (2-79, OR STANDARD)
        [8,16,24,32,40,48,56,64,72]
?  ↵
Form length in Lines Per Page (6-144)
        [66] ?  18 ↵
Top of Form (Channel 1) Line Number (1-18)
        [4] ?  2 ↵
Bottom of Form (Channel 12) Line number (2-18)
        [18] ?  ↵
VFU Tape (Line numbers 2-18, Channels 2-11, OR STANDARD)
        []
?  4-10 ↵
?  ↵
Output to Pathname
        [:UDD:F77:MARLL:MEMBERS.CARDS] ?  ↵

Command ?  BYE ↵

FCU terminating ...
```

Marll and his system's operator, John, go to the operator's console (username OP) and to the printer. They perform the following steps.

1. They determine that the current LPP, CPL, and HEADERS values are 66, 80, and 1, respectively.
2. They wait for the current print queue to LPT (devicename @LPB) to complete.
3. John gives these commands to the CLI.

```
CONTROL @EXEC PAUSE @LPB
CONTROL @EXEC LPP @LPB 18
CONTROL @EXEC CPL @LPB 50
CONTROL @EXEC HEADERS @LPB 0
```

4. They insert and align the cards on their Model 4216 printer.
5. John gives these commands to the CLI.

```
CONTROL @EXEC CONTINUE @LPB
QPRINT :UDD:F77:MARLL:MEMBERS.CARDS
CONTROL @EXEC PAUSE @LPB
CONTROL @EXEC LPP @LPB 66
CONTROL @EXEC LPP @CPL 80
CONTROL @EXEC HEADERS @LPB 1
```

6. They remove the cards and reinsert standard 11-inch high paper.
7. John gives the command

   CONTROL @EXEC CONTINUE @LPB

   to finish the restoration of the printer to its previous settings.

The most important point in this section is that you must place special characters (VFU codes) in an output file so that when it prints, the paper advances properly. The *AOS/VS Operator's Guide* explains how the FORMS command can help to eliminate the need for giving many specific instructions each time you need to print an F77-created output file on special forms.

End of Chapter

Chapter 8
Introduction to DG/DBMS

Overview

The Data General Database Management System (DG/DBMS) is a CODASYL-compliant, network-structured, database management system.

In general, a database is a collection of interrelated data. This data is stored in a way that controls redundancy, while allowing many distinct applications to use the information in different ways. The database system controls access to the data, and stores the data independent of the programs that use it.

DG/DBMS is a system that manages and coordinates access to all the data in the database. With it, you can set up your applications so that information from different sources exists in the same database. Many programs can then access and modify the data concurrently. In this way, the system need not store data more than once, and all programs can access the most recent information about any subject.

DG/DBMS protects data by allowing only certain programs to access certain data. Some programs can modify data, others can only read data. DG/DBMS prevents more than one program from modifying the same piece of data simultaneously. Furthermore, it prevents a program from accessing information that might be inconsistent because another program is in the process of modifying the same information.

This chapter and the next seven describe the statements and clauses you must add to your FORTRAN 77 program in order to use DG/DBMS, and how to link the DG/DBMS library routines with your FORTRAN 77 program. Because we are assuming your familiarity with DG/DBMS, we have *not* included a full discussion of DG/DBMS here. You should read the *Data General/Database Management System (DG/DBMS) Reference Manual* before you attempt to use the FORTRAN 77 preprocessor.

DG/DBMS Description

DG/DBMS is a CODASYL-compliant DBMS. The database administrator (DBA) in your organization uses the interactive Data Definition Facility (DDF) to enter the description of the database structure. This description is called the *Schema* and is written in the data definition language (DDL). There is one schema per database, which names and describes the contents of every data item in the database. It also names and describes the grouping of data items into records, and the structural relationships between records (called *sets*).

The DBA also uses the DDF to create one or more *subschemas*, each of which describes one "view" of the database. A subschema contains: a subset of the data items, records, and sets, all of which were defined in the schema. Data items can be reordered within a record and defined to be of a different data type and length than that used in the schema. While the schema is a language-independent description of the database, the subschemas are language-dependent. This means that a COBOL program, a FORTRAN 5 program, and a FORTRAN 77 program that "see" the same data will each use a different subschema. This is because the internal representation of some data types is different for each language. Figure 8-1 illustrates the schema-to-subschema-to-language relationships.

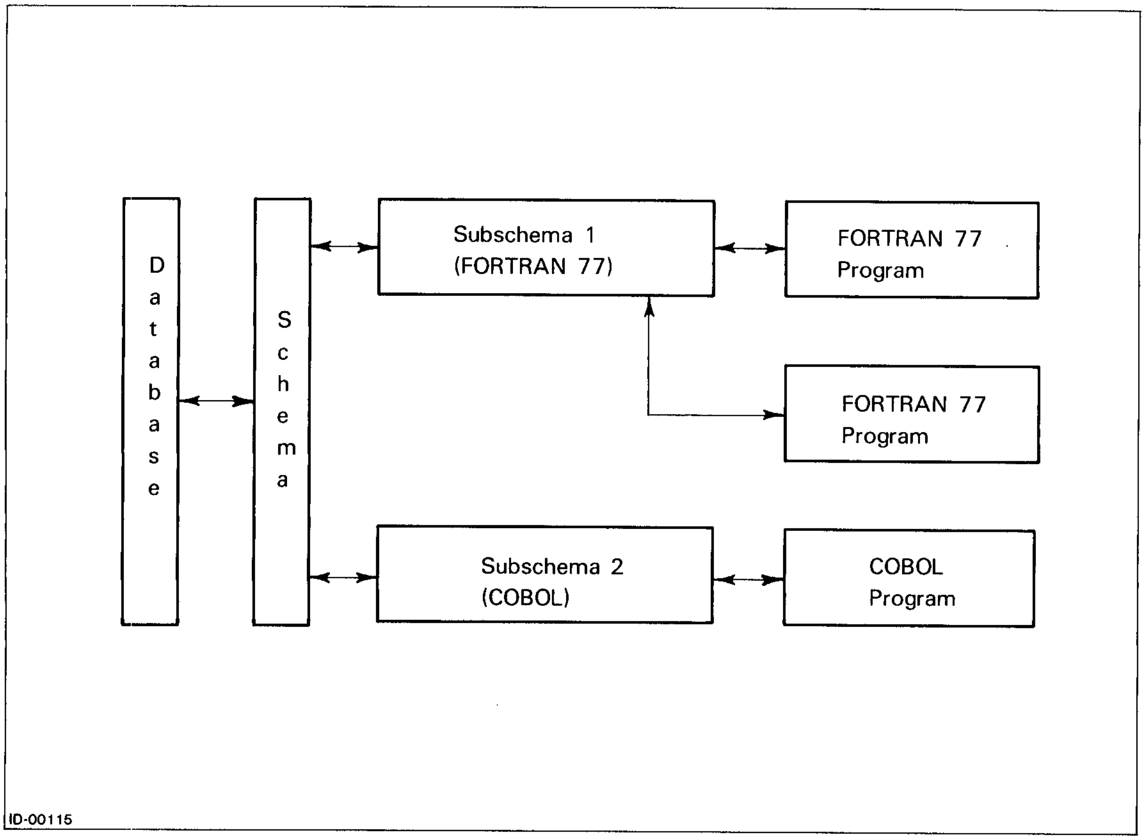


Figure 8-1. Schema-Subschema-Language Relationships

Notice that all of the relationships are two-way. That is, DG/DBMS performs the correct transformations from schema to subschema and from subschema to schema.

Your application program cannot change the defined structure of the database or create a new one. Only your DBA can do this. However, the DBA can allow you to add, delete, or change the values of entries and to change the relationships between entries in the database.

The FORTRAN 77 Preprocessor Interface

The FORTRAN 77 language interface is a preprocessor-runtime interface to DG/DBMS.

The FORTRAN 77 preprocessor analyzes a data manipulation language (DML) similar to the specifications in the "CODASYL FORTRAN DATA BASE FACILITY" (January 1980). The preprocessor translates each DML statement in your application program into one or more executable FORTRAN 77 statements. These statements set up the necessary parameters for a CALL statement to a runtime interface routine. You must give your program to the preprocessor immediately before the actual FORTRAN 77 compilation (of the preprocessor's output from your program).

In order to understand what the FORTRAN 77 preprocessor is doing, you must know a little of how DG/DBMS works. Every data item, record, and set in the schema and subschema is assigned an internal identifier by the DDF. The association of symbolic names to internal identifiers must be made by the preprocessor or runtime routines before the DBMS can process a command. For example, in a DML statement that refers to a record by name (e.g., "EMPLOYEE"), the preprocessor or runtime routines must convert the name to the record's identifier (e.g., 3). In order to simplify the runtime interface, as well as for performance considerations, this association is done at compile time. The

FORTRAN 77 preprocessor performs the translation from FORTRAN-like DML statements that use symbolic names for data items, records, and sets to CALLs to the DBMS runtime routines that have the correct internal identifiers.

Figure 8-2 shows the relationships between the various DG/DBMS components, the FORTRAN 77 preprocessor, and your program. It is important to understand these relationships. The figure shows the following:

- The DDF uses the DBA's description of the schema and subschema to produce a database, called the *metadata database*. This database describes the schema and all of the subschemas. The FORTRAN 77 subschema source code generator within the DDF produces a file with a FORTRAN 77 format specification of all sets, records and items in the subschema. The *metadata binder* produces a compact representation of the metadata database; this is the packed metadata (PMD). The PMD includes a set of files that can be used to compute an internal identifier given a symbolic name. The subschema source code file and the metadata files are left in the database directory for use by the preprocessor in translating your programs.
- The FORTRAN 77 preprocessor reads a source file that contains your program or subprogram.

  A DG/DBMS FORTRAN 77 program contains an INVOKE statement that names the subschema to be used. The correct subschema source code file is inserted into your program and all DML statements in the program are converted to CALLs to the appropriate DBMS runtime routine, replacing symbolic names by their internal identifiers. All non-DBMS statements are left unchanged. Note that the only difference between compiling a program that contains DML statements and compiling one that does not, is the preprocessor step. The CLI macro DB.F77.CLI has made this step nearly invisible. The preprocessor also correctly processes non-DBMS source files that contain only one run unit.
- The output of the preprocessor is a FORTRAN 77 source program that is input to the FORTRAN 77 compiler. After compilation you use AOS/VS Link to link your compiled program with the DG/DBMS runtime routines to produce an executable program file.

Figure 8-2 summarizes these points.

How to use the Interface

To use DG/DBMS statements in a FORTRAN 77 program, you must include a description of the database in your program. Your DBA will have built this description with the DDF.

First, the DBA uses the DDF to describe and create the schema of the entire database. Then, the DBA creates subsets of the schema (called subschemas), which are compatible with FORTRAN 77 syntax. The preprocessor INVOKE statement automatically copies the subschema source code file into your F77 program.

The DDF automatically produces the FORTRAN 77 source file when the DBA defines a subschema. If the DBA modifies a subschema, you must recompile your F77 program with the new version of the subschema.

For example, for the database with the name, TREATMENT_DATABASE, using the subschema name, PATIENT_SEARCH, located in directory, :UDD:BRUCE, you would write the statement

```
D      INVOKE(SUBSCHEMA="PATIENT_SEARCH",
          SCHEMA=":UDD:BRUCE:TREATMENT_DATABASE")
```

in an F77 source program.

Subschemas have special access controls that determine if a user has compile, retrieve, and update access to a subschema. The *DG/DBMS Reference Manual* contains complete information about subschema access rights.

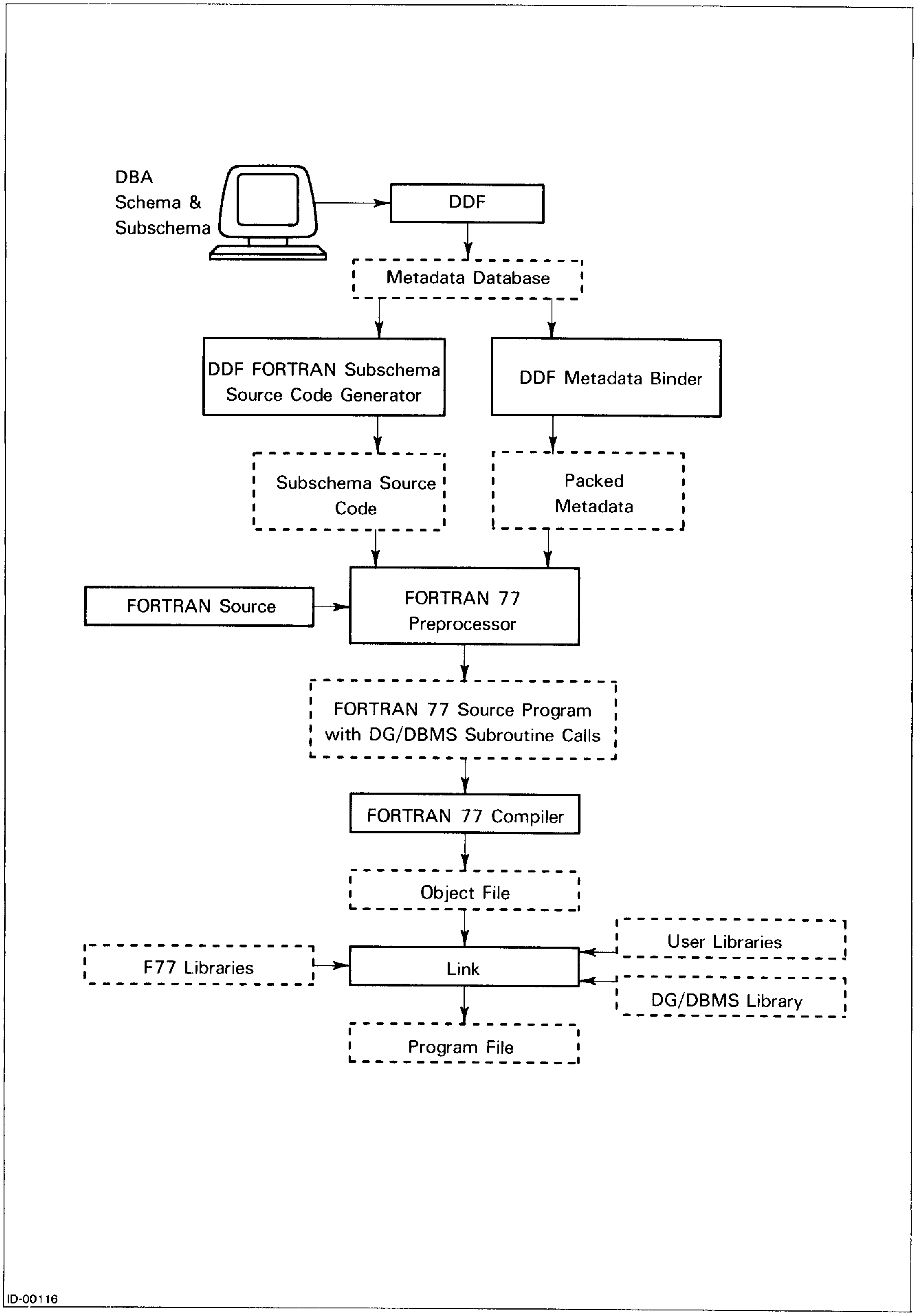


Figure 8-2. Progression from Data Definition Through Executable Code

The schema pathname is the AOS/VS directory pathname for the database directory. The subschema name is the name of the subschema source code file *without* the ".F77" extension.

Figure 8-3 shows an example of subschema source code illustrating the use of DG/DBMS in F77. Figure 8-4 shows the structure of the data.

```
C  SUBSCHEMA NAME IS "PATIENT_SEARCH"
C     ALLOWS ERASE GET MODIFY STORE


C  SET DEFINITION SECTION.

C     SET = DOCTORS_BY_NAME
C           ALLOWS RECONNECT
C           OWNER IS SYSTEM
C           MEMBER IS DOCTOR
C           AUTOMATIC MANDATORY
C           ORDER IS SORTED BY KEY ASCENDING
C              KEYS ARE:
C                       DOCTOR_LAST_NAME
C                       DOCTOR_FIRST_NAME
C              DUPLICATES ALLOWED
C           MEMBER LIMIT IS NONE

C     SET = PATIENTS_BY_NAME
C           ALLOWS RECONNECT
C           OWNER IS SYSTEM
C           MEMBER IS PATIENT
C           AUTOMATIC MANDATORY
C           ORDER IS SORTED BY KEY ASCENDING
C              KEYS ARE:
C                       PATIENT_LAST_NAME
C                       PATIENT_FIRST_NAME
C              DUPLICATES ALLOWED
C           MEMBER LIMIT IS NONE

C     SET = PATIENT_TREATMENTS
C           ALLOWS RECONNECT
C           OWNER IS PATIENT
C           MEMBER IS TREATMENTS
C           AUTOMATIC MANDATORY
C           ORDER IS NEXT
C           MEMBER LIMIT IS NONE

C     SET = DOCTOR_TREATMENTS
C           ALLOWS RECONNECT
C           OWNER IS DOCTOR
C           MEMBER IS TREATMENTS
C           AUTOMATIC MANDATORY
C           ORDER IS NEXT
C           MEMBER LIMIT IS NONE
```

DG-25249

Figure 8-3. Subschema Example (continues)

```
C  RECORD DEFINITION SECTION.

C  RECORD = DOCTOR     ALLOWS ERASE GET MODIFY STORE

      CHARACTER*1 DOCTOR
      COMMON/DOCTOR/DOCTOR_LAST_NAME
            CHARACTER*25 DOCTOR_LAST_NAME
C               CONTENTS: CHAR*25L     ALLOWS GET MODIFY

      COMMON/DOCTOR/DOCTOR_FIRST_NAME
            CHARACTER*20 DOCTOR_FIRST_NAME
C               CONTENTS: CHAR*20L     ALLOWS GET MODIFY

      COMMON/DOCTOR/SPECIALTY
            CHARACTER*15 SPECIALTY
C               CONTENTS: CHAR*15L     ALLOWS GET MODIFY

      COMMON/DOCTOR/INFO
            CHARACTER*40 INFO
C               CONTENTS: CHAR*40L     ALLOWS GET MODIFY

      COMMON/DOCTOR/BEEPER
            INTEGER*2 BEEPER
C               CONTENTS: NUMERIC      ALLOWS GET MODIFY
C               RANGE:    -9999 TO +9999

      EQUIVALENCE (DOCTOR,DOCTOR_LAST_NAME)


C  RECORD = PATIENT     ALLOWS ERASE GET MODIFY STORE

      CHARACTER*1 PATIENT
      COMMON/PATIENT/PATIENT_LAST_NAME
            CHARACTER*20 PATIENT_LAST_NAME
C               CONTENTS: CHAR*20L     ALLOWS GET MODIFY

      COMMON/PATIENT/PATIENT_FIRST_NAME
            CHARACTER*16 PATIENT_FIRST_NAME
C               CONTENTS: CHAR*15L     ALLOWS GET MODIFY

      COMMON/PATIENT/WARD
            CHARACTER*4 WARD
C               CONTENTS: CHAR*4L      ALLOWS GET MODIFY

      COMMON/PATIENT/ROOM
            INTEGER*2 ROOM
C               CONTENTS: NUMERIC      ALLOWS GET MODIFY
C               RANGE:    +0 TO +999

      EQUIVALENCE (PATIENT,PATIENT_LAST_NAME)
```

DG-25249

Figure 8-3. Subschema Example (continued)

```
C  RECORD = TREATMENTS     ALLOWS ERASE GET MODIFY STORE

      CHARACTER*1 TREATMENTS
      COMMON/TREATMENTS/DISEASE
             CHARACTER*100 DISEASE
C                CONTENTS: CHAR*100L    ALLOWS GET MODIFY

      COMMON/TREATMENTS/MEDICATION
             CHARACTER*25 MEDICATION (5)
C                CONTENTS: CHAR*25L     ALLOWS GET MODIFY

      COMMON/TREATMENTS/DIET
             CHARACTER*200 DIET
C                CONTENTS: CHAR*200L    ALLOWS GET MODIFY

      COMMON/TREATMENTS/SPECIAL_INSTRUCTIONS
             CHARACTER*40 SPECIAL_INSTRUCTIONS (5)
C                CONTENTS: CHAR*40L     ALLOWS GET MODIFY

      EQUIVALENCE (TREATMENTS,DISEASE)


C  END OF FORTRAN 77 "PATIENT_SEARCH" SUBSCHEMA.
```

DG-25249

Figure 8-3. Subschema Example (concluded)

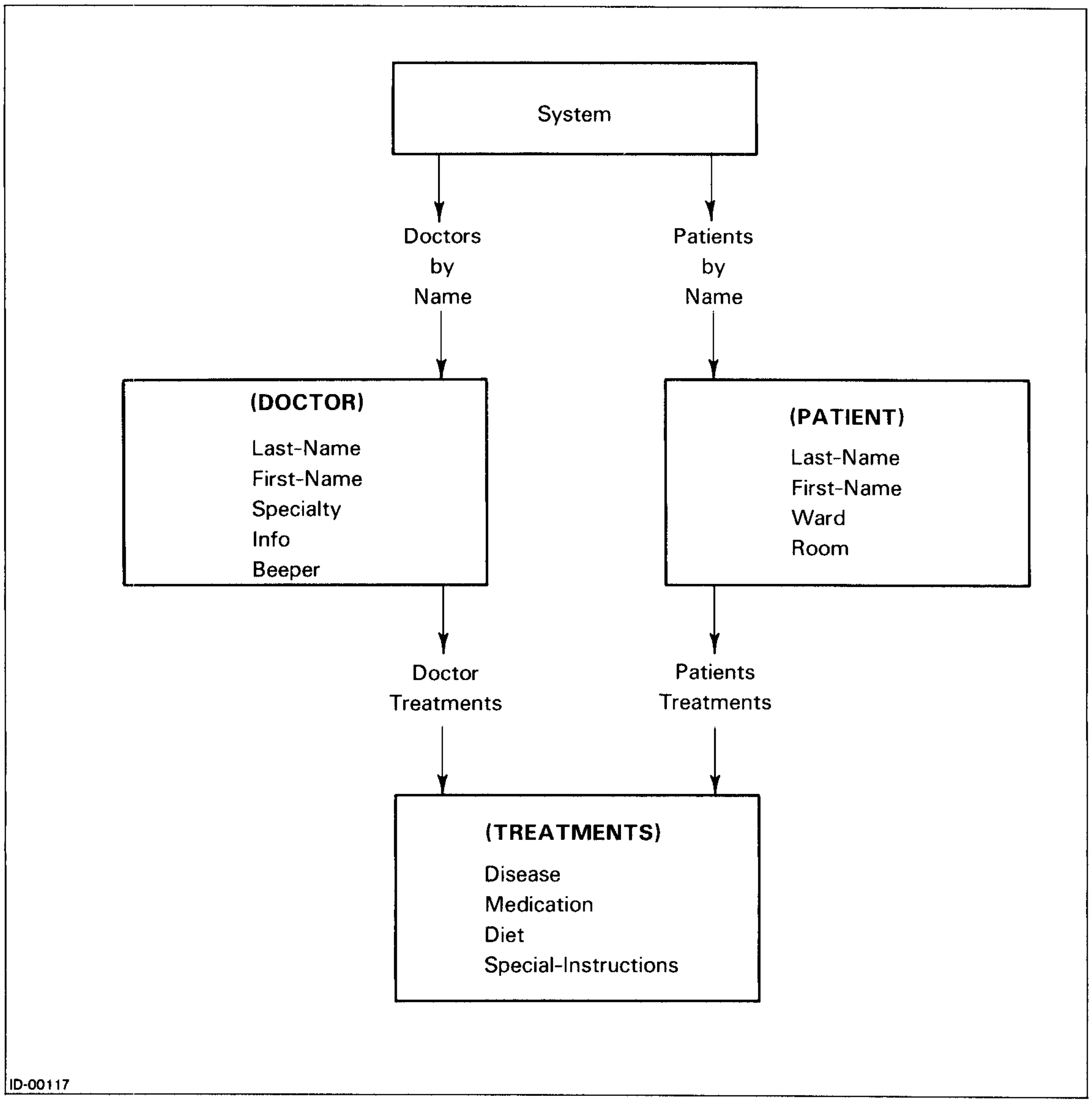


Figure 8-4. Structure of Data in the Subschema Example

Database Records

Subschema source code files have two parts. The first part, a series of comment lines, describes interrecord relationships that are called *sets*. The second part, a series of declarations of named COMMON areas, describes the various record types for the database.

A *record type* is a logical unit of data, composed of elementary items. Many records of the same record type can exist in the database; we call these *occurrences* of the record type.

Figure 8-5 shows the subschema's record type descriptions of our database. This part of the subschema defines three record types with a total of 13 data items. The data items in this structure can be manipulated with ordinary FORTRAN 77 statements (IF, assignment, etc.). But, if you use a special collection of statements, called data manipulation language (DML) statements on those data items, the data structure becomes a window into the DG/DBMS database.

When you INVOKE the subschema in your program, a storage area called the User Work Area (UWA) is defined in named COMMON blocks. You use ordinary FORTRAN 77 statements to use data in the UWA in your program. You use DML statements to transfer information between the UWA and the database. The record definitions in the subschema both *declare* the data that your program can access in the database, and *define* the UWA of the program.

Each record description includes a comment on user access rights. DG/DBMS enforces these restrictions on any program using the subschema to access the database file. The preprocessor also enforces these restrictions at compile time.

```
C  RECORD DEFINITION SECTION.


C  RECORD = DOCTOR     ALLOWS ERASE GET MODIFY STORE

      CHARACTER*1 DOCTOR
      COMMON/DOCTOR/DOCTOR_LAST_NAME
             CHARACTER*25 DOCTOR_LAST_NAME
C                CONTENTS: CHAR*25L     ALLOWS GET MODIFY

      COMMON/DOCTOR/DOCTOR_FIRST_NAME
             CHARACTER*20 DOCTOR_FIRST_NAME
C                CONTENTS: CHAR*20L     ALLOWS GET MODIFY

      COMMON/DOCTOR/SPECIALTY
             CHARACTER*15 SPECIALTY
C                CONTENTS: CHAR*15L     ALLOWS GET MODIFY

      COMMON/DOCTOR/INFO
             CHARACTER*40 INFO
C                CONTENTS: CHAR*40L     ALLOWS GET MODIFY

      COMMON/DOCTOR/BEEPER
             INTEGER*2 BEEPER
C                CONTENTS: NUMERIC      ALLOWS GET MODIFY
C                RANGE:    -9999 TO +9999

      EQUIVALENCE (DOCTOR,DOCTOR_LAST_NAME)


C  RECORD = PATIENT     ALLOWS ERASE GET MODIFY STORE

      CHARACTER*1 PATIENT
      COMMON/PATIENT/PATIENT_LAST_NAME
             CHARACTER*20 PATIENT_LAST_NAME
C                CONTENTS: CHAR*20L     ALLOWS GET MODIFY

      COMMON/PATIENT/PATIENT_FIRST_NAME
             CHARACTER*16 PATIENT_FIRST_NAME
C                CONTENTS: CHAR*15L     ALLOWS GET MODIFY

      COMMON/PATIENT/WARD
             CHARACTER*4 WARD
C                CONTENTS: CHAR*4L      ALLOWS GET MODIFY
```

DG-25250

Figure 8-5. Example Subschema Record Type Description (continues)

```
      COMMON/PATIENT/ROOM
           INTEGER*2 ROOM
C                   CONTENTS: NUMERIC      ALLOWS GET MODIFY
C                   RANGE:    +0 TO +999

      EQUIVALENCE (PATIENT,PATIENT_LAST_NAME)


C  RECORD = TREATMENTS     ALLOWS ERASE GET MODIFY STORE

      CHARACTER*1 TREATMENTS
      COMMON/TREATMENTS/DISEASE
           CHARACTER*100 DISEASE
C                   CONTENTS: CHAR*100L    ALLOWS GET MODIFY

      COMMON/TREATMENTS/MEDICATION
           CHARACTER*25 MEDICATION (5)
C                   CONTENTS: CHAR*25L     ALLOWS GET MODIFY

      COMMON/TREATMENTS/DIET
           CHARACTER*200 DIET
C                   CONTENTS: CHAR*200L    ALLOWS GET MODIFY

      COMMON/TREATMENTS/SPECIAL_INSTRUCTIONS
           CHARACTER*40 SPECIAL_INSTRUCTIONS (5)
C                   CONTENTS: CHAR*40L     ALLOWS GET MODIFY

      EQUIVALENCE (TREATMENTS,DISEASE)


C  END OF FORTRAN 77 "PATIENT_SEARCH" SUBSCHEMA.
```

DG-25250

Figure 8-5. Example Subschema Record Type Description (concluded)

Database Navigation

SYSTEM Sets

All schemas and subschemas contain a special record type, named SYSTEM. The SYSTEM record serves as an initial entry point into the database. The SYSTEM record is system maintained and there is only one occurrence of it, therefore, it can always be located. Set types that have the SYSTEM record as their owner are called system sets. (The SYSTEM record can only be an owner, never a member.) All programs that access a database start out by navigating in a system set because system sets are the only set occurrences that can be directly located when a program starts.

Set Types

Relationships between different record types are defined by set types. The first part of a subschema contains the definitions of the subschema's sets. Figure 8-6 shows the set types in our example subschema.

Each set type consists of the following:

- The set's name.
- A clause listing the allowed connection statements.
- An owner record type specification.
- A member record type specification.
- Insertion/retention criteria for member records.
- The ordering criteria of the member records within a set occurrence. If the set is sorted, this includes the list of the sort key item(s) and a clause specifying whether or not duplicate sort key values are allowed.
- The maximum number of member occurrences within a set occurrence (there could be no limit).

```
C  SET DEFINITION SECTION.

C     SET = DOCTORS_BY_NAME
C           ALLOWS RECONNECT
C           OWNER IS SYSTEM
C           MEMBER IS DOCTOR
C           AUTOMATIC MANDATORY
C           ORDER IS SORTED BY KEY ASCENDING
C              KEYS ARE:
C                       DOCTOR_LAST_NAME
C                       DOCTOR_FIRST_NAME
C              DUPLICATES ALLOWED
C           MEMBER LIMIT IS NONE

C     SET = PATIENTS_BY_NAME
C           ALLOWS RECONNECT
C           OWNER IS SYSTEM
C           MEMBER IS PATIENT
C           AUTOMATIC MANDATORY
C           ORDER IS SORTED BY KEY ASCENDING
C              KEYS ARE:
C                       PATIENT_LAST_NAME
C                       PATIENT_FIRST_NAME
C              DUPLICATES ALLOWED
C           MEMBER LIMIT IS NONE

C     SET = PATIENT_TREATMENTS
C           ALLOWS RECONNECT
C           OWNER IS PATIENT
C           MEMBER IS TREATMENTS
C           AUTOMATIC MANDATORY
C           ORDER IS NEXT
C           MEMBER LIMIT IS NONE

C     SET = DOCTOR_TREATMENTS
C           ALLOWS RECONNECT
C           OWNER IS DOCTOR
C           MEMBER IS TREATMENTS
C           AUTOMATIC MANDATORY
C           ORDER IS NEXT
C           MEMBER LIMIT IS NONE
```

DG-25251

Figure 8-6. Set Types in Subschema Example

Set Occurrences

A *set occurrence* consists of a least one occurrence of the owner record type, and zero or more member record occurrences. The set type specification defines the owner/member relationships in the set occurrence.

The set type specifications and owner/member relationships are described in detail in the *DG/DBMS Reference Manual.*

CODASYL databases are *network* databases. You can consider the set occurrences as the pathways that you travel, and the record occurrences as the destinations to which you go. Figure 8-7 illustrates a set occurrence.

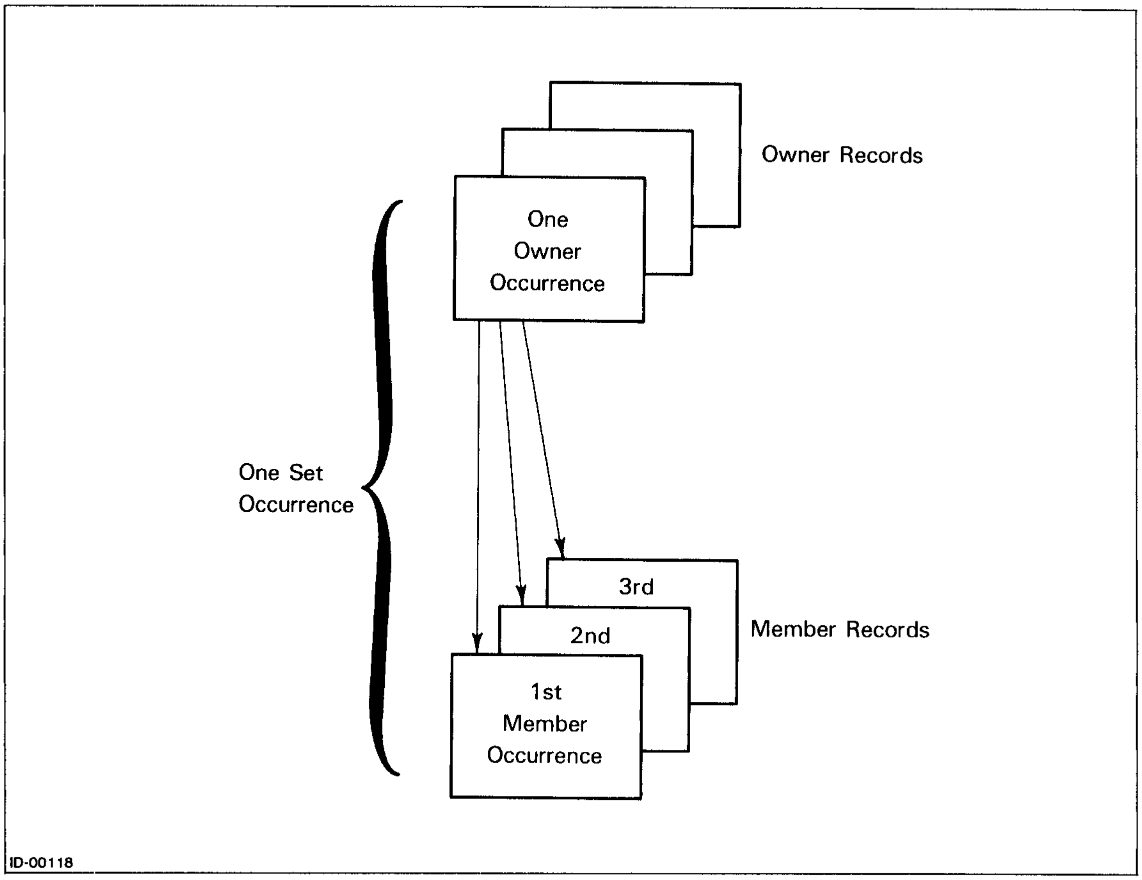


Figure 8-7. A Set Occurrence

End of Chapter

Chapter 9
DG/DBMS Subschema Data Definition

Overview

The DBA defines a subschema using the Data Definition Facility. The description of the subschema and record specifications for the FORTRAN 77 program are contained in a source code file, which is produced by the FORTRAN 77 subschema source code generator, invoked by DDF upon request by the DBA.

The subschema defines your view of the database. Each set is described by a group of comments. All records and associated items are allocated in FORTRAN as named COMMON blocks.

Every FORTRAN 77 application program (or subprogram) must identify the subschema it is using with an INVOKE statement before any other DML statement in the program. The INVOKE statement has the form

```
D     INVOKE (SUBSCHEMA="<ss_name>",SCHEMA="<s_name>")
```

where "D" must be in column 1, and where "INVOKE" must begin in column 7 or beyond. The parameter <ss_name> is the subschema name and <s_name> is the absolute or relative pathname of the database. The INVOKE statement identifies the subschema that the preprocessor will use for the data declarations it will make in your program.

Schema to Subschema Transformation

Data is transformed from DG/DBMS internal format to FORTRAN 77 format when it is read from the database, and from FORTRAN 77 format to DG/DBMS internal format when it is stored. These transformations are automatically performed by DG/DBMS. The DBA's definition of the schema and subschema determines the transformations to be performed.

Schema Data Formats

DG/DBMS stores all data in one of four internal formats and automatically converts data between schema and subschema formats. The schema formats are:

- Multiple-precision signed integer (two's complement binary) with implied decimal point.
- Floating point (single- and double-precision).
- Character.
- Bit.

FORTRAN 77 Subschema Data Formats

DG/DBMS supports three basic types of data formats in a FORTRAN 77 subschema. These are numeric, character, and bit. The following sections describe the FORTRAN 77 data types that can be used to represent these formats.

Numeric Data

Numeric data can be represented in a FORTRAN 77 subschema with any of the following data types:

- INTEGER*2 — Single-word signed integer (implied trailing decimal point).
- INTEGER*4 — Double-word signed integer (implied trailing decimal point).
- REAL*4 — Single-precision floating point.
- REAL*8 — Double-precision floating point.
- DOUBLE PRECISION — Double precision floating point

DG/DBMS can convert data items declared as numeric in the schema (either FIXED or FLOAT) to any of the above data types. If the conversion results in the number being truncated, DG/DBMS will issue a warning.

Character Data

Character data is represented in a FORTRAN 77 subschema with the CHARACTER*n data type. DG/DBMS does not convert character data to any other formats; hence, the corresponding schema data item must be declared as CHARACTER. Subschema data items can be longer or shorter than the schema definition, and can have a different justification. Strings will be truncated to fit into shorter data items. Strings moved into longer data items will be padded with blanks. Character strings must contain valid ASCII characters and must not contain any nulls (i.e., '<000>'); otherwise, a data conversion error will result.

Bit Data

Since there is no bit data type in FORTRAN 77, the subschema allows the usual FORTRAN programming practice of storing the data in variables and arrays of other FORTRAN 77 data types. The following data types can be used:

- INTEGER*2, LOGICAL*2 (16 bits per element)
- INTEGER*4, LOGICAL*4, REAL*4 (32 bits per element)
- REAL*8, DOUBLE PRECISION (64 bits per element)

DDF will create an array of elements large enough to contain the bit string; e.g., a bit string of 31 bits could be contained in a 2-occurrence array of INTEGER*2 or in a 1-occurrence array of INTEGER*4.

The type of data conversion used for items declared as BIT in the schema depends on the contents declaration of the subschema. The contents declaration defines the usage the item will have in the subschema. There are two ways to convert BIT items.

First, if the usage of the item in the subschema is declared as BIT, the destination data item will be treated as a (left-justified) bit string. Starting with the leftmost bit, each bit is moved one at a time, without any content checking. If the destination string is longer than the source, the excess is padded with zero bits. If the destination string is shorter than the source, the excess bits are truncated on the right. If any nonzero bits are truncated, a warning is returned. See the "Character and Bit Strings" section in Chapter 14 for additional information.

The second way of handling bit strings is to define the subschema usage as NUMERIC. In this case, the destination is treated as a right-justified bit string whose length is the length of the data type (e.g., 16 bits for INTEGER*2). Starting with the rightmost bit, the data is moved into the low-order bits of the destination. If necessary, the string is truncated or padded on the left with zero bits.

Supported Subschema Data Types and Conversion Rules

Table 9-1 summarizes all the legal data type declarations in a FORTRAN 77 subschema.

Table 9-1. Supported FORTRAN 77 Subschema Data Types

Data Contents	Data Declaration	Size (bytes)	Aln	Dimension
CHARACTER	CHAR*n	n	B	Number of times the item occurs in the schema.
BIT	INTEGER*2 LOGICAL*2 INTEGER*4 LOGICAL*4 REAL*4 REAL*8 DOUBLE PRECISION	2 2 4 4 4 8 8	W W W W W W W	First dimension is the minimum number of times the data type must occur in order to contain the data item. Second dimension is the number of times the item occurs in the schema.
NUMERIC	INTEGER*2 INTEGER*4 REAL*4 REAL*8 DOUBLE PRECISION	2 4 4 8 8	W W W W W	Number of times the item occurs in the schema.

where:

Data Contents	defines the basic data type of the item.
Data Declaration	defines the FORTRAN 77 data declarations that are allowed for the given contents.
Size (bytes)	defines the number of bytes of storage a single occurrence of the given type occupies. To compute the total amount of space a type occupies, multiply the byte size of the item by all its occurs clauses.
Aln	defines the alignment required for the declaration; B means byte alignment, W means word.
Dimension	defines the dimension declarations for the given type.

Table 9-2 summarizes the allowed schema-to-subschema data type mappings and the action DG/DBMS takes.

Table 9-2. Schema to Subschema Data Type Mappings

Subschema Type Specification	Schema Type Specification			
	Fixed Numeric	Float Numeric	Character	Bit
CHARACTER*N	4	4	2	3
INTEGER*2	1	1	4	3
INTEGER*4	1	1	4	3
LOGICAL*2	4	4	4	3
LOGICAL*4	4	4	4	3
REAL*4	1	1	4	3
REAL*8	1	1	4	3
DOUBLE PRECISION	1	1	4	3

The following four notes apply to Table 9-2.

1. Numeric Move

   After converting the data to the destination data type, DG/DBMS will:

   - Align the digits on the decimal point (or implied decimal point).
   - Truncate/pad to fit the source number into the destination number.
   - Perform a range check, if a range was specified.
   - Send a truncation warning if it truncates a number.

2. Character Move

   DG/DBMS verifies the string for valid characters.

   Characters in the source string are moved to the destination, as follows:

   - If the destination string is left-justified, then the leftmost character of the source string is moved to the leftmost character of the destination string, followed by the next leftmost character, etc.
   - If the destination string is right-justified, the system starts with the rightmost character of the source string and moves it to the rightmost character position of the destination, followed by the next rightmost character, etc.

   If the destination string is longer than the source, DG/DBMS will pad it with blanks.

   If the source string is longer than the destination, then the system truncates it to the size of the destination. If any of the truncated characters were not blanks, you will receive a truncation warning.

3. Bit Move

   Basically, this is the same as a character move, except that DG/DBMS:

   - Performs no verification.
   - Moves strings on a bit-by-bit basis, not by character.
   - Pads with zero bits, not blanks.
   - Will send a truncation warning if it truncates any nonzero bits (as opposed to blanks).

   Bit strings with BIT as subschema content type are left-justified. Bit strings with NUMERIC as subschema content type are right-justified.

4. Not allowed.

FORTRAN 77 Subschema Data Definition

Figure 9-1 is an example of the DDF screen used to define FORTRAN 77 subschema data types.

```
FORTRAN 77 SUBSCHEMA: DATA ITEMS: Record Name: EMPLOYEE
--------------------------------------------------------------------------
                                                Fortran 77    Access
No. I  Data Item:  subschema name     Contents  Specification Rights
    E              schema name                  TYPE    DIM    G M
--- -  ------------------------------  -------- ----  -------  ------

 1  I  LAST_NAME                       CHAR*20L  C*10            Y Y
         LASTNAME
 2  I  MID_INIT                        CHAR*1L   C*1             Y Y
         MIDINIT
 3  I  FIRST_NAME                      CHAR*10L  C*10            Y Y
         FIRSTNAME
 4  I  EMPNO                           NUMERIC   I*2             Y Y
         EMPNO


I/E:       I = Include,  E = Exclude       Access rights: G = Get, M = Modify
Contents: NUMERIC, CHAR*nj (n = # of chars, j = L or R), BIT*n (n = # of bits)
Types:     I*n=Integer, L*n=Logical, R*n=Real, C*n=Character, D=Double Precision

-------------------------------------------------------------------------------
```

DG-25252

Figure 9-1. Sample FORTRAN 77 Data Item Screen

The following notes apply to Figure 9-1.

1. I
   E

   The I/E column defines whether the data item is included (I) or excluded (E) from the subschema. Excluded items are displayed dimly, and included items at normal intensity.

2. Data Item

   The Data Item name column defines the name the item will have in the subschema. The name the item has in the schema is displayed on the second line.

3. Contents

   The Contents column defines the usage the data item will have in the subschema. FORTRAN 77 data items can contain CHARACTER data, BIT data, or NUMERIC data.

   For data declared as CHARACTER in the schema, the Contents entry will be CHAR*nj, where *n* is the number of characters in the string and *j* is the justification of the string (left or right). Both the length and the justification of the item can be changed.

   For data declared as BIT in the schema, this field determines whether the item is to be treated as a bit string or as a number. Declaring the field as BIT*n causes the data item to be treated as a left-justified bit string, where *n* is the number of bits in the string. Declaring the field as NUMERIC causes the data item to be treated as a number (a right-justified string) whose length is the maximum number of bits that can be contained in the data type declared.

   For data declared as NUMERIC in the schema, this column will have NUMERIC in it and serves only as a comment.

4. TYPE

   The TYPE column defines the FORTRAN 77 data type declaration for the item. The supported FORTRAN 77 types are: CHARACTER, INTEGER*2, INTEGER*4, LOGICAL*2, LOGICAL*4, REAL*4, REAL*8, and DOUBLE PRECISION. Which data types are allowed depends on the Contents entry. Table 9-1, earlier, defines all possible legal combinations of Contents and TYPE.

5. DIM

   The DIM (for "dimension") column defines the array dimensions (if any) of the data item. The numbers for this entry are computed and displayed by DDF and cannot be changed on this screen. The values in the DIM column depend upon the item's Contents entry and its schema occurs value. For NUMERIC and CHARACTER data, the entry contains the occurs value of the item in the schema. For BIT data, the field contains at most two dimensions. The first dimension (calculated by DDF) reflects the amount of physical storage necessary to accommodate the data item; the second dimension contains the occurs value of the item in the schema. If any array dimension is computed to have a value of one, the dimension is omitted (its value is implicitly one).

6. Access Rights

   Finally, the last two columns on the screen are used to define the access rights for the item. The item can have Get (G) and/or Modify (M) access, or neither.

Default Subschema Data Types

When the DDF first generates a FORTRAN 77 subschema, it automatically translates the schema data item definitions into the most appropriate FORTRAN 77 data item specifications. These specifications of data type can be changed by the DBA before the subschema is bound. For additional information about subschema binding consult the *DG/DBMS Reference Manual.*

Table 9-3 shows the FORTRAN 77 specifications that DDF will generate as the default.

Table 9-3. Default FORTRAN 77 Subschema Data Types

Schema	Contents	FORTRAN Specification
CHARACTER		
X(n)L	CHAR*nL	CHARACTER*n [(OCCURS)]
X(n)R	CHAR*nR	CHARACTER*n [(OCCURS)]
BIT		
B(n)	BIT*n	INTEGER*2 [([(n+15)/16] [,OCCURS])]
FIXED NUMERIC		
9(n) n < 5	NUMERIC	INTEGER*2 [(OCCURS)]
9(n) 4 < n < 10	NUMERIC	INTEGER*4 [(OCCURS)]
9(n) 9 < n	NUMERIC	DOUBLE PRECISION [(OCCURS)]
9(n).9(m)	NUMERIC	DOUBLE PRECISION [(OCCURS)]
FLOAT NUMERIC		
P(1)	NUMERIC	REAL*4 [(OCCURS)]
P(2)	NUMERIC	DOUBLE PRECISION [(OCCURS)]

End of Chapter

Chapter 10
Data Manipulation Statements for DG/DBMS

FORTRAN 77 Data Manipulation Statements

Overview

DG/DBMS databases are accessed by DML statements, which you include directly in your F77 application program. Each DML statement must contain a "D" in column 1. The "D" is optional on continuation lines. Since one DML statement might produce more than one FORTRAN statement, labels are not allowed on DML statements.

The DG/DBMS FORTRAN 77 DML syntax is compatible with that of FORTRAN 77. The DML statements take the form of a command verb, generally followed by a list of keyword parameters enclosed in parentheses. The parameters can be entered in any order.

Using Free Cursors

The use of free cursors allows your application programs to set and update their own database position markers. For a complete description of free cursors and their uses, see the *DG/DBMS Reference Manual* or the *Guide to Using DG/DBMS*.

To use free cursors in your FORTRAN program, you must include a declaration for each free cursor. The format of this declaration is the following.

```
D     FREE CURSOR fc_name [(n)], RECORD=rec_name
```

The D must be in column 1, and the keyword FREE CURSOR must begin in column 7 or beyond.

The FORTRAN program unit that contains the READY DML statement must contain a declaration for every free cursor. In all other subprograms, only the free cursors that are used in that subprogram have to be declared. Free cursors can also be one-dimensional arrays.

When any module is preprocessed, a FORTRAN 77 named COMMON statement and an INTEGER*2 declaration statement will be generated for each free cursor declared in that module.

It is very important that you use free cursors only in DML statements. Altering their values with FORTRAN 77 statements will lead to unexpected results.

Note that FREE CURSOR statements are nonexecutable. They are used to declare internal data structures and to make the association of fc_name with record type rec_name. As nonexecutable statements that cause the declarations of COMMON blocks, FREE CURSOR statements must be placed near the beginning of a module (but after the INVOKE statement).

The DML Statements

You use the DML statements to:

- Open and close a database.
- Handle transactions.
- Locate record occurrences.

- Read and update record occurrences.
- Modify the set participation of record occurrences.
- Assign and check cursors.

The statements are organized according to their functionality. The seven DML statement groups are defined briefly in the following sections:

- Subschema Statements
- Transaction Statements
- Connection Statements
- Find Statements
- Record Statements
- Fetch Statements
- Utility Statements

Refer to the *DG/DBMS Reference Manual* for a complete description of the actions of these statements.

Subschema Statements

The subschema statements are INVOKE, READY, and FINISH.

INVOKE names the database and subschema to be used by the FORTRAN program. It should not be confused with READY, which actually opens the database. INVOKE is a nonexecutable statement that identifies the database to be used and sets up the declarations of named COMMON areas for the data items and record types in the subschema.

READY opens the database for use through the INVOKEd subschema. You can indicate the usage (EXCLUSIVE or CONCURRENT) and the mode (UPDATE or RETRIEVAL) with which the database is to be opened.

FINISH terminates access to the database via the currently READY subschema.

Transaction Statements

The transaction statements are INITIATE, COMMIT, ROLLBACK, and CHECK.

INITIATE signals the start of a database transaction in either UPDATE or RETRIEVAL mode. The transaction ID assigned by DG/DBMS is returned as a parameter. Note that transactions cannot be nested.

COMMIT signals the end of the outstanding transaction.

ROLLBACK voids the outstanding transaction, undoing any updates that you might have made.

CHECK returns the status of any transaction. The transaction id is represented as a double-precision floating-point number. This statement returns an INTEGER*2 variable whose possible values are as follows.

Code	Meaning
0	Unknown Transaction
1	Transaction currently active
2	Transaction successfully completed
3	Transaction backed out (ROLLBACK completed)
4	System error; DG/DBMS cannot determine the status

Connection Statements

The connection statements are CONNECT, DISCONNECT, and RECONNECT.

CONNECT inserts an existing member record into the set identified by the current-of-set. The record can be identified through either a free cursor or by the current-of-record.

DISCONNECT removes a member record occurrence from the specified set. The record can be identified by either a free cursor or by the current-of-record.

RECONNECT disconnects the given member record from its present set position and then reconnects it into the position indicated by the specified current-of-set. The record can be identified only by a free cursor.

Find Statements

The find statements are FIND CURRENT, FIND OWNER, FIND positional, and FIND keyed.

The ASSIGN free cursor clause is optional with all of the find DML statements. There is also an independent ASSIGN statement.

FIND positional will return an end-of-set error if the search fails. FIND Keyed will return an unsuccessful-keyed-search error if the search fails. An END clause can be included in the these FIND statements to test for these conditions.

In all FIND statements, except for FIND CURRENT, the record clause is included for member validation. It is also a source of data in FIND keyed.

FIND CURRENT has two forms. In the first form, the record to be located is that designated by a free cursor or by the current-of-record. In the second form, the record to be located is that designated by the specified current-of-set, if the record occurrence indicated by the current-of-set matches the given record type.

FIND OWNER locates the owner record occurrence of the specified current-of-set.

FIND positional locates a specific member record occurrence within the given set occurrence. The search is performed using a direction relative to the record that is current-of-set (FIRST, LAST, NEXT, or PRIOR). Alternatively, the direction can be specified as a nonzero variable. If the variable is positive, the nth record from the first record occurrence will be located. If the variable is negative, then the nth record from the last record occurrence will be located.

FIND keyed can take several forms depending on the fields being compared, the source of the comparison values, the relationship being tested and the direction of the search. The fields being compared are either a SORT KEY (the key of a sorted set) or a SEARCH KEY (an arbitrary list of data items in the record). The values used for comparison can be taken from either the user work area (UWA) or the from the fields in the current record of the set occurrence (CUR_SET). The FORTRAN relational operators .EQ., .GT., .LT., .GE., and .LE. can be used. However, not all operators are allowed with all combinations of fields, value source (UWA or CUR_SET), and direction of search (FIRST, LAST, NEXT or PRIOR). For the legal combinations, the FIND statement locates the FIRST/LAST/NEXT/PRIOR member whose fields are in the specified relationship to the values selected for the fields specified in the statement.

Record Statements

The record statements are GET, MODIFY, STORE, and ERASE.

GET retrieves either an entire record occurrence or a specified list of items within a record occurrence. The record can be identified either with a free cursor or by the current-of-record. GET statements can specify up to 15 items to be retrieved, or they can retrieve an entire record.

MODIFY updates all or some of the items in the specified record. The record can be identified either with a free cursor or by the current-of-record. MODIFY statements can specify up to 15 items to be updated, or they can update an entire record.

ERASE deletes a record occurrence, disconnecting it from all sets in which it is connected as a member. The record can be identified either with a free cursor or by the current-of-record.

STORE creates a new record occurrence of the given type, connecting it into all sets in which it is an automatic member. The ASSIGN free cursor clause can be used with this statement.

Fetch Statements

A FETCH statement is a combination of a FIND and GET. For each FIND statement defined in the previous section "Find Statements" there is a corresponding FETCH statement. The syntax for a FETCH is identical to that of a FIND. Each FETCH statement performs the search indicated cated by its corresponding FIND (including the ASSIGN and END clauses). If the FIND succeeds, a GET is done on the record and all the related cursors are updated. If the FETCH fails, the cursors are left unchanged and no record is retrieved.

Utility Statements

The utility statements are ASSIGN, CONNECTED, OWNER, MEMBER, NULL, and EMPTY. Each of these statements, except ASSIGN, acts as a logical function. They can be used either in a logical assignment statement or as the logical expression in an IF construct.

When a free cursor is declared, it is associated with a specific record type. The ASSIGN free cursor statement sets the named free cursor to the current-of-record of the record type with which it is associated.

The CONNECTED function returns the value .TRUE., if the given record is a connected member of the specified set; and it returns the value .FALSE., if it is not. The record can be identified either with a free cursor or by the current-of-record.

The OWNER function returns the value .TRUE., if the current-of-set is on an owner record type of the specified set; and it returns the value .FALSE., if it is not.

The MEMBER function returns the value .TRUE., if the current-of-set is on a member record type of the specified set; and it returns the value .FALSE., if it is not.

The NULL function returns the value .TRUE., if the specified free cursor, record cursor, or set cursor does not distinguish a record; and it returns the value .FALSE., if it does.

The EMPTY function returns the value .TRUE., if the specified set is empty; and it returns the value .FALSE., if it is not.

Error Handling

The preprocessor declares DBSTATUS as an INTEGER*2 variable in named COMMON. This integer will be set by every DML statement to the resulting error code. A successful statement will return the value zero. Unsuccessful statements will return either an AOS/VS error code or a DBMS error code of the form 017xxxK. The *DG/DBMS Reference Manual* has a full description of all DBMS error codes (including a description of the effect an error has on the database and on your cursors). You can explicitly test the value of DBSTATUS at any time in your program or with your own default error handler.

Several types of error handling options are available for the various DML statements. In all statements (except the logical functions), an error clause (ERR=<err_opt>) can be included to specify an action to be taken on any error returned from the call. The FIND and FETCH DML statements can also contain an end clause (END=<end_opt>), which specifies an action to be taken when an unsuccessful-keyed-search or an end-of-set error occurs. The format of the two clauses is as follows:

```
ERR =  { label          }
       { subr [(args)]  }

END =  { label          }
       { subr [(args)]  }
```

Depending on the error clauses that you include in a DML statement, the preprocessor will either generate error handling code in your application program, or errors will be handled by calling a default error subroutine. In any case, DBSTATUS always contains the error code returned by the most recent DML statement.

If the ERR clause is included, but not the END clause, any error that occurs will be returned. The preprocessor will generate one of the following two statements.

```
IF (DBSTATUS .NE. 0) GO TO label
IF (DBSTATUS .NE. 0) CALL subr [(args)]
```

If the ERR clause is not included, but the END clause is, the default error routine will be called if any error other than end-of-set or unsuccessful-keyed-search occurs. The preprocessor will generate one of the following two statements:

```
IF (DBSTATUS .EQ. ercode) GO TO label
IF (DBSTATUS .EQ. ercode) CALL subr [(args)]
```

The ercode symbol represents a constant for either the end-of-set error code or the unsuccessful-keyed-search error code. For each FIND (or FETCH), only one of the two ercodes is a possible result. For sort key or search key FINDs, unsuccessful-keyed-search could be returned. For positional FINDs, end-of-set could be returned. The preprocessor will only generate code to check for the possible ercode.

If both the ERR clause and the END clause are included, any error that occurs will be returned. The preprocessor will generate the following two statements in the order listed.

```
IF ( DBSTATUS .EQ. ercode )  { GO TO label         }
                             { CALL subr [(args)]  }

IF ( (DBSTATUS .NE. 0) .AND. (DBSTATUSNE. ercode))
                             { GO TO label         }
                             { CALL subr [(args)]  }
```

If DBMS returns an error for which you have not specified an error handling clause, the default error handling module, DBERROR, is called. This module performs two actions. First, it calls the FORTRAN 77 runtime ERRCODE with the value of DBSTATUS as input. ERRCODE will display the error and an error traceback. Then, it transfers control to the father process. (See Chapter 2 for a further explanation of ERRCODE).

Alternatively, you can write your own DBERROR routine and place it before the interface library in your link line. This will cause your DBERROR routine to be used in place of the supplied default DBERROR. The following information will be helpful in writing a DBERROR replacement:

- DBERROR must be defined as a SUBROUTINE with no arguments. The invoking statement is CALL DBERROR.
- The subroutine can either terminate execution or RETURN. In the latter case, execution will continue with the statement following the DML statement that resulted in the error.
- If your DBERROR subroutine makes use of DBSTATUS, it must either contain an INVOKE statement or a declaration of DBSTATUS in named COMMON as follows.

  ```
  COMMON/DBSTATUS/DBSTATUS
  INTEGER*2 DBSTATUS
  ```

- The DBERROR subroutine can do DML. However, you must be aware that any DML statement will reset DBSTATUS, since it is a global (COMMON) variable. Also, you must ensure that any DML in DBERROR does not result in an error call to DBERROR, which executes a DML statement that causes an error, etc.

End of Chapter

Chapter 11
Data Manipulation Language Syntax for DBMS

Syntax Overview

The syntax used in this chapter is the syntax used in the CODASYL FORTRAN Database Facility. The syntax is a modified Backus-Normal Form (BNF), with the following rules in addition to those of the usual BNF:

- The rules for comments, continuation, statements, blank characters, and identifiers are as defined in FORTRAN 77. DML statements cannot be labeled. A "D" is required in column 1 on the first line of a DML statement, but is optional on any continuation statements. We use a "." in column 6 on second and subsequent lines to indicate the continuation of a DML statement. You can use any legal continuation character.
- In statements that contain a parenthesized list of parameters, a single comma (only) is required between adjacent parameters. Also, no comma can follow a left parenthesis. For example, the illegal statement

  ```
  D     READY (,UPDATE,,ERR=100)
  ```

  should become the legal statement

  ```
  D     READY (UPDATE,ERR=100)
  ```

- The parentheses enclosing a list in a WHERE clause, can be deleted if the list consists of one item. For example

  ```
  D     FIND(FIRST,RECORD=EMP,SET=EMPS,WHERE((NAME).EQ.UWA))
  ```

  can become

  ```
  D     FIND(FIRST,RECORD=EMP,SET=EMPS,WHERE(NAME.EQ.UWA))
  ```

- In a statement that has an empty parenthesized list, the parentheses must be deleted. For example, change

  ```
  READY()
  ```

  to

  ```
  READY
  ```

- The clauses within a parenthesized list can be in any order. For example

  ```
  FIND (FIRST,RECORD=EMP,SET=PAY)
  ```

  is equivalent to

  ```
  FIND (SET=PAY,RECORD=EMP,FIRST)
  ```

Syntax Meta-Symbols

The following meta-symbols are used in this chapter. Note that a single vertical bar (|) represents "or".

args	List of valid FORTRAN 77 subroutine parameters. Arithmetic expressions are not allowed.
end_opt	{ label | subr *[args]* }
err_opt	{ label | subr *[args]* }
fc_name	Any valid free cursor name that you declare; maximum length is 32 characters. The names of all free cursors must be unique in the first eight characters.
item	Any valid item name that is declared in your subschema; maximum length is 32 characters.
itemlist	item *[,item]*. There is a maximum of 15 items in an item list.
label	Any valid FORTRAN 77 statement label.
logical	Any valid FORTRAN 77 LOGICAL variable.
path_name	Any valid AOS/VS pathname for a file.
posit	DOUBLE PRECISION variable that can be used in the FIND positional statement to indicate the search direction.
rec_name	Any valid record name that is declared in your subschema; maximum length is 32 characters. Record names must be unique within the first eight characters.
set_name	Any valid set name that is declared in your subschema; maximum length is 32 characters.
ss_name	The name of the subschema being used in a (sub)program. Maximum length is 27 characters.
s_name	The name of the schema (database) being used in a (sub)program.
status	INTEGER variable that you declare to receive the transaction status from CHECK transaction.
stmt	Any valid FORTRAN 77 statement except a DO statement; it cannot be a DML statement.
subr	Any valid FORTRAN 77 subroutine name.
tx_id	DOUBLE PRECISION variable that you declare to receive the transaction id from INITIATE.
usinglist	(item_list)

The remaining pages in this chapter present the details of each DML statement. The statements appear alphabetically.

ASSIGN Statement

Format

D ASSIGN (FREE CURSOR=fc_name *[,ERR=err_opt]*)

This statement assigns the free cursor **fc_name** to the record occurrence, which is current-of-record. Free cursor **fc_name** is associated with a specific record type as a result of the FREE CURSOR statement in which it was declared.

If you include the *ERR=err_opt* clause and DG/DBMS returns an error, control passes to *err_opt*.

CHECK Transaction Status Statement

Format

D CHECK (ID=tx_id *[,ERR=err_opt]*) status

When you issue the CHECK Transaction Status command, DG/DBMS returns an integer describing the status of the transaction into **status**. The following is a list of these codes and their meanings.

Code	Meaning
0	Unknown transaction
1	Transaction currently active
2	Transaction successfully completed
3	Transaction backed out (ROLLBACK completed)
4	System error; DG/DBMS cannot determine the status

If you include the *ERR=err_opt* clause and DG/DBMS returns an error, control passes to *err_opt*.

The CHECK Transaction Status command never affects any cursors.

COMMIT Statement

Format

D   COMMIT *[(ERR=err_opt)]*

The COMMIT statement makes the updates you have made to the database permanent. Unless you COMMIT your transaction, all changes you have made to the database will be rolled back.

After you COMMIT a transaction, any future transaction can see the modifications you have made to the database. Before you COMMIT your transaction, your modifications are not visible to other users. After the COMMIT, you need a new INITIATE command to continue to access the database.

If you include the *ERR=err_opt* clause and DG/DBMS returns an error, control passes to *err_opt*.

The COMMIT statement has no effect on any cursors.

Note that COMMIT is a relatively "inexpensive" command. DG/DBMS made all the database modifications during the program transactions; COMMIT simply makes these modifications visible to other users.

To end a transaction and abort all changes made during it, use the ROLLBACK statement.

CONNECT Statement

Format

D   CONNECT   ({RECORD=rec_name | FREE CURSOR=fc_name},
•   SET=set_name *[,ERR=err_opt]*)

DG/DBMS connects the record indicated by current-of-record or by a free cursor to the owner record within the current-of-set defined by set_name. You must be positioned on an occurrence of set_name and have either a record cursor or a free cursor on the potential member record.

If you include the *ERR=err_opt* clause and DG/DBMS returns an error, control passes to *err_opt*.

CONNECT has no effect on free cursors. It sets the current-of-record to the connected record occurrence. It sets the current-of-set for set_name to the connected record occurrence.

CONNECT statements are used to associate owner and member record occurrences in MANUAL or OPTIONAL set types. (See the *DG/DBMS Reference Manual* for a full discussion of sets and set types).

CONNECTED Function

Format

```
D        logical = [.NOT.] CONNECTED
  •          ({RECORD=rec_name | FREE CURSOR=fc_name},
  •            SET=set_name)

D        IF ([.NOT.] CONNECTED
  •          ({RECORD=rec_name | FREE CURSOR=fc_name},
  •            SET=set_name)) stmt
```

This function tests to see whether or not the record indicated by current-of-record or a free cursor is connected in the set type set_name.

DML logical functions must not contain any additional logical tests.

stmt cannot be a DML statement or a DO statement.

DISCONNECT Statement

Format

```
D     DISCONNECT      ({RECORD=rec_name | FREE CURSOR=fc_name},
  •                    SET=set_name [,ERR=err_opt])
```

This statement disconnects the member record on which you are currently positioned, or which you have marked by a free cursor from its owner in set_name.

You cannot disconnect an occurrence in a MANDATORY set type. (However, you can ERASE the member record occurrences.) See the *DG/DBMS Reference Manual* for a full discussion of sets and set types.

If you include the *ERR=err_opt* clause and DG/DBMS returns an error, control passes to *err_opt*.

DISCONNECT has no effect on free cursors. It sets the current-of-record of the member record type to the disconnected record. It leaves current-of-set in the disconnected set on the "hole" the record used to occupy.

EMPTY Function

Format

D logical = *[.NOT.]* EMPTY (SET=set_name)

D IF (*[.NOT.]* EMPTY (SET=set_name)) stmt

This function tests to see whether or not the set occurrence identified by the current-of-set has any member occurrences.

DML logical functions must not contain any additional logical tests.

stmt cannot be a DML statement or a DO statement.

ERASE Statement

Format

D ERASE ({RECORD=rec_name | FREE CURSOR=fc_name}
• *[,ERR=err_opt]*)

ERASE deletes from the database the record occurrence indicated by current-of-record or free cursor fc_name. You cannot ERASE a record occurrence if it is the owner of another record occurrence in any set; you must ERASE or DISCONNECT all of its member records first.

If you include the *ERR=err_opt* clause and DG/DBMS returns an error, control passes to *err_opt*.

ERASE sets the following to null:

- Free cursors that were pointing to the erased occurrence.
- Current-of-record for this record type.
- Current-of-set for all sets that the erased record owned.

ERASE sets the current-of-set for all sets in which the erased record was a connected member to the "hole" the record left in the set occurrence.

Note that a DG/DBMS ERASE always causes an immediate, physical deletion. The only way to undo an ERASE command is to ROLLBACK the transaction instead of COMMITting it.

FETCH CURRENT Statement

Format

```
D     FETCH     (CURRENT, {RECORD=rec_name |
   •            FREE CURSOR=fc_name},
   •            [,ASSIGN=fc_name]
   •            [,ERR=err_opt])

D     FETCH     (CURRENT, RECORD=rec_name, SET=set_name
   •            [,ASSIGN=fc_name] [,ERR=err_opt])
```

The FETCH CURRENT statement is the only FETCH that does not locate a new record occurrence. The statement resets all the system cursors associated with a particular record type to point to the same, previously known record occurrence. Current-of-record, current-of-set, or a free cursor indicate the known record occurrence. If a record is successfully found, FETCH then gets the record.

If you specify just a record name or free cursor name, DG/DBMS will locate the appropriate occurrence (provided the cursor is not null.) If you specify a record and set type for the record you want to locate, the system will compare that record type to the one identified by current-of-set. If they match and current-of-set is not null, DG/DBMS will perform the find as indicated, otherwise an error message will be generated and all cursors will remain unchanged.

FETCH CURRENT has no effect on free cursors unless the optional ASSIGN clause is present, in which case it will assign a free cursor to the found record.

FETCH CURRENT sets current-of-set to the located record in all sets in which the record is an owner or a connected member, and sets current-of-record to the located record occurrence.

If you include the *ERR=err_opt* clause and DG/DBMS returns an error, control passes to *err_opt*.

FETCH OWNER Statement

Format

```
D     FETCH     (OWNER, RECORD=rec_name, SET=set_name
   •            [,ASSIGN=fc_name] [,ERR=err_opt])
```

FETCH OWNER locates the owner record occurrence of the set indicated by the current-of-set for **set_name**. If a record is successfully found, FETCH then gets the record.

The ASSIGN clause assigns a free cursor to the located record.

FETCH OWNER has no effect on free cursors (unless the ASSIGN clause is used). It sets current-of-set to the located record in all set types in which the record is an owner or a connected member. It sets current-of-record to the located record occurrence.

If you include the *ERR=err_opt* clause and DG/DBMS returns an error, control passes to *err_opt*.

FETCH Positional Statement

Format

```
D    FETCH    ({FIRST | LAST | NEXT | PRIOR | OFFSET=posit},
•             RECORD=rec_name, SET=set_name
•             [,ASSIGN=fc_name] [,END=end_opt] [,ERR=err_opt])
```

FETCH positional moves you through occurrences of a record type within a given set occurrence. If a record is successfully found, then FETCH gets the record.

FIRST locates the first occurrence of the record type in the current set occurrence.

LAST locates the last occurrence of the record type in the current set occurrence.

NEXT locates the next occurrence of the record type (relative to the current-of-set) within the set occurrence.

PRIOR locates the immediately previous occurrence of the record type (relative to the current-of-set) within the set occurrence.

If you specify a positive posit, DG/DBMS locates the record that is posit occurrences from the beginning of the set. If you specify a negative posit, DG/DBMS locates the record that is posit occurrences from the end of the set.

The ASSIGN clause assigns a free cursor to the located record occurrence.

FETCH positional has no effect on free cursors (unless the ASSIGN clause is used). It sets current-of-set to the located record in all set types in which the record is an owner or a connected member. It sets the current-of-record to the located record occurrence.

If you include the *ERR=err_opt* clause and DG/DBMS returns an error, control passes to *err_opt*.

If you include the *END=end_opt* clause and DG/DBMS encounters end-of-set, control passes to *end_opt*.

FETCH Keyed (SEARCH KEY) Statement

Format

```
D     FETCH     ({FIRST | LAST | NEXT | PRIOR},
  •             RECORD=rec_name, SET=set_name,
  •             WHERE(usinglist {.EQ.|.GT.|.LT.|.GE.|.LE.} UWA)
  •             [,ASSIGN=fc_name] [,END=end_opt] [,ERR=err_opt])

D     FETCH     ({NEXT | PRIOR}, RECORD=rec_name,
  •             SET=set_name, WHERE(usinglist .EQ. CUR_SET)
  •             [,ASSIGN=fc_name] [,END=end_opt] [,ERR=err_opt])
```

Use this statement to find a record occurrence for which you know the contents of one or more specific fields. If a record is successfully found, FETCH then gets the record.

If you specify UWA, the search will be conducted based on the values in the User Work Area, for the field(s) specified in usinglist.

If you specify CUR_SET, the search will be conducted based on the values in the record identified by current-of-set, for the field(s) specified in usinglist.

FIRST locates the first occurrence of the record type in the current set occurrence for which the search criteria are satisfied.

LAST locates the last occurrence of the record type in the current set occurrence for which the search criteria are satisfied.

NEXT locates the next occurrence (relative to the current-of-set) of the record type within the set occurrence, for which the search criteria are satisfied.

PRIOR locates the immediately previous occurrence (relative to the current-of-set) of the record type within the set occurrence, for which the search criteria are satisfied.

The ASSIGN clause assigns a free cursor to the located record.

This FETCH statement has no effect on free cursors (unless the ASSIGN clause is used). It sets current-of-set to the located record in all set types in which the record is an owner or a connected member. It sets current-of-record to the located record occurrence.

If you include the *ERR=err_opt* clause and DG/DBMS returns an error, control passes to *err_opt*.

If you include the *END=end_opt* clause and DG/DBMS returns unsuccessful-keyed-search, control passes to *end_opt*.

FETCH Keyed (SORT KEY) Statement

Format

```
D    FETCH    (FIRST, RECORD=rec_name, SET=set_name,
 •            WHERE(SORT KEY {.EQ.|.GE.} UWA)
 •            [,ASSIGN=fc_name] [,END=end_opt] [,ERR=err_opt])

D    FETCH    (LAST, RECORD=rec_name, SET=set_name,
 •            WHERE(SORT KEY {.EQ.|.LE.} UWA)
 •            [,ASSIGN=fc_name] [,END=end_opt] [,ERR=err_opt])

D    FETCH    ({NEXT | PRIOR}, RECORD=rec_name,
 •            SET=set_name, WHERE(SORT KEY {.EQ.|.NE} CUR_SET)
 •            [,ASSIGN=fc_name] [,END=end_opt] [,ERR=err_opt])
```

Use this statement to find a record occurrence for which you know the contents of one or more specific field(s) on which the set is sorted. If a record is successfully found, FETCH then gets the record.

If you specify UWA, the search will be conducted based on the values in the User Work Area, for the field(s) on which the set is sorted.

If you specify CUR_SET, the search will be conducted based on the values in the record occurrence indicated by current-of-set, for the field(s) on which the set is sorted.

FIRST locates the first occurrence of the record type in the current set occurrence, for which the search criteria are satisfied.

LAST locates the last occurrence of the record type in the current set occurrence, for which the search criteria are satisfied.

NEXT locates the next occurrence (relative to the current-of-set) of the record type within the set occurrence, for which the search criteria are satisfied.

PRIOR locates the immediately previous occurrence (relative to the current-of-set) of the record type within the set occurrence, for which the search criteria are satisfied.

The ASSIGN clause assigns a free cursor to the located record.

FETCH has no effect on free cursors (unless the ASSIGN clause is used). It sets current-of-set to the located record in all set types in which the record is an owner or a connected member. It sets the current-of-record to the located record occurrence.

If you include the *ERR=err_opt* clause and DG/DBMS returns an error, control passes to *err_opt*.

If you include the *END=end_opt* clause and DG/DBMS returns unsuccessful-keyed-search, control passes to *end_opt*.

FIND CURRENT Statement

Format

```
D     FIND        (CURRENT, {RECORD=rec_name |
  •               FREE CURSOR=fc_name}, [,ASSIGN=fc_name]
  •               [,ERR=err_opt])

D     FIND        (CURRENT, RECORD=rec_name, SET=set_name
  •               [,ASSIGN=fc_name] [,ERR=err_opt])
```

The FIND CURRENT statement is the only FIND that does not locate a new record occurrence. The statement resets all the system cursors associated with a particular record type to point to the same, previously known record occurrence. Current-of-record, current-of-set, or a free cursor indicate the known record occurrence.

If you specify just a record name or free cursor name, DG/DBMS will locate the appropriate occurrence (provided the cursor is not null.) If you specify a record and set type for the record you want to locate, the system will compare that record type to the one identified by current-of-set. If they match and current-of-set is not null, DG/DBMS will perform the find as indicated, otherwise an error message will be generated and all cursors will remain unchanged.

FIND CURRENT has no effect on free cursors unless the optional ASSIGN clause is present. In this case, it will assign a free cursor to the found record.

FIND CURRENT sets current-of-set to the located record in all sets in which the record is an owner or a connected member. It also sets current-of-record to the located record occurrence.

If you include the *ERR=err_opt* clause and DG/DBMS returns an error, control passes to *err_opt*.

FIND OWNER Statement

Format

```
D     FIND        (OWNER, RECORD=rec_name, SET=set_name
  •               [,ASSIGN=fc_name] [,ERR=err_opt])
```

FIND OWNER locates the owner record occurrence of the set indicated by the current-of-set for set_name.

The ASSIGN clause assigns a free cursor to the located record.

FIND OWNER has no effect on free cursors (unless the ASSIGN clause is used). It sets current-of-set to the located record in all set types in which the record is an owner or a connected member. It sets current-of-record to the located record occurrence.

If you include the *ERR=err_opt* clause and DG/DBMS returns an error, control passes to *err_opt*.

FIND Positional Statement

Format

```
D    FIND    ({FIRST | LAST | NEXT | PRIOR | OFFSET=posit},
  •          RECORD=rec_name, SET=set_name
  •          [,ASSIGN=fc_name] [,END=end_opt] [,ERR=err_opt])
```

FIND positional moves you through occurrences of a record type within a given set occurrence.

FIRST locates the first occurrence of the record type in the current set occurrence.

LAST locates the last occurrence of the record type in the current set occurrence.

NEXT locates the next occurrence of the record type (relative to the current-of-set) within the set occurrence.

PRIOR locates the immediately previous occurrence of the record type (relative to the current-of-set) within the set occurrence.

If you specify a positive posit, DG/DBMS locates the record that is posit occurrences from the beginning of the set. If you specify a negative posit, DG/DBMS locates the record that is posit occurrences from the end of the set.

The ASSIGN clause assigns a free cursor to the located record occurrence.

FIND positional has no effect on free cursors (unless the ASSIGN clause is used). It sets current-of-set to the located record in all set types in which the record is an owner or a connected member. It sets the current-of-record to the located record occurrence.

If you include the *ERR=err_opt* clause and DG/DBMS returns an error, control passes to *err_opt*.

If you include the *END=end_opt* clause and DG/DBMS encounters end-of-set, control passes to *end_opt*.

FIND Keyed (SEARCH KEY) Statement

Format

```
D     FIND      ({FIRST | LAST | NEXT | PRIOR},
  •             RECORD=rec_name, SET=set_name,
  •             WHERE(usinglist {.EQ.|.GT.|.LT.|.GE.|.LE.} UWA)
  •             [,ASSIGN=fc_name] [,END=end_opt] [,ERR=err_opt])

D     FIND      ({NEXT | PRIOR}, RECORD=rec_name,
  •             SET=set_name, WHERE(usinglist .EQ. CUR_SET)
  •             [,ASSIGN=fc_name] [,END=end_opt] [,ERR=err_opt])
```

Use this statement to find a record occurrence for which you know the contents of one or more specific fields.

If you specify UWA, the search will be conducted based on the values in the User Work Area, for the field(s) specified in usinglist.

If you specify CUR_SET, the search will be conducted based on the values in the record identified by current-of-set, for the field(s) specified in usinglist.

FIRST locates the first occurrence of the record type in the current set occurrence for which the search criteria are satisfied.

LAST locates the last occurrence of the record type in the current set occurrence for which the search criteria are satisfied.

NEXT locates the next occurrence (relative to the current-of-set) of the record type within the set occurrence, for which the search criteria are satisfied.

PRIOR locates the immediately previous occurrence (relative to the current-of-set) of the record type within the set occurrence, for which the search criteria are satisfied.

The ASSIGN clause assigns a free cursor to the located record.

This FIND statement has no effect on free cursors (unless the ASSIGN clause is used). It sets current-of-set to the located record in all set types in which the record is an owner or a connected member. It sets current-of-record to the located record occurrence.

If you include the *ERR=err_opt* clause and DG/DBMS returns an error, control passes to *err_opt*.

If you include the *END=end_opt* clause and DG/DBMS returns unsuccessful-keyed-search, control passes to *end_opt*.

FIND Keyed (SORT KEY) Statement

Format

```
D    FIND    (FIRST, RECORD=rec_name, SET=set_name,
  •          WHERE(SORT KEY {.EQ.|.GE.} UWA)
  •          [,ASSIGN=fc_name] [,END=end_opt] [,ERR=err_opt])
D    FIND    (LAST, RECORD=rec_name, SET=set_name,
  •          WHERE(SORT KEY {.EQ.|.LE.} UWA)
  •          [,ASSIGN=fc_name] [,END=end_opt] [,ERR=err_opt])
D    FIND    ({NEXT | PRIOR}, RECORD=rec_name,
  •          SET=set_name, WHERE(SORT KEY {.EQ.|.NE} CUR_SET)
  •          [,ASSIGN=fc_name] [,END=end_opt] [,ERR=err_opt])
```

Use this statement to find a record occurrence for which you know the contents of one or more specific fields on which the set is sorted.

If you specify UWA, the search will be conducted based on the values in the User Work Area, for the field(s) on which the set is sorted.

If you specify CUR_SET, the search will be conducted based on the values in the record occurrence indicated by current-of-set, for the field(s) on which the set is sorted.

FIRST locates the first occurrence of the record type in the current set occurrence, for which the search criteria are satisfied.

LAST locates the last occurrence of the record type in the current set occurrence, for which the search criteria are satisfied.

NEXT locates the next occurrence (relative to the current-of-set) of the record type within the set occurrence, for which the search criteria are satisfied.

PRIOR locates the immediately previous occurrence (relative to the current-of-set) of the record type within the set occurrence, for which the search criteria are satisfied.

The ASSIGN clause assigns a free cursor to the located record.

FIND has no effect on free cursors (unless the ASSIGN clause is used). It sets current-of-set to the located record in all set types in which the record is an owner or a connected member. It sets the current-of-record to the located record occurrence.

If you include the *ERR=err_opt* clause and DG/DBMS returns an error, control passes to *err_opt*.

If you include the *END=end_opt* clause and DG/DBMS returns unsuccessful-keyed-search, control passes to *end_opt*.

FINISH Statement

Format

D FINISH *[(ERR=err_opt)]*

The FINISH statement closes a database to your program.

If you include the *ERR=err_opt* clause and DG/DBMS returns an error, control passes to *err_opt*.

Note that FINISH does not COMMIT an outstanding transaction. An outstanding transaction will be rolled back.

FREE CURSOR Declarations

Format

D FREE CURSOR fc_name *[(n)]* , RECORD=rec_name

The FORTRAN (sub)program that contains the READY DML statement must contain a declaration for every free cursor. In all other subprograms, only the free cursors that are used in that subprogram have to be declared. The order of the declarations in any module is arbitrary. Free cursors can also be one-based, one-dimensional arrays. Furthermore, *n* can be an integer so that an array of FREE CURSORS is declared.

Free cursors can be used only in DML statements. Altering their values with FORTRAN 77 statements can lead to unexpected results.

FREE CURSOR statements are nonexecutable. They are used to declare the necessary data structures and and to make the association of fc_name with record type rec_name.

FREE CURSOR is a data declaration statement and must precede the READY statement.

GET Statement

Format

```
D      GET       ( {RECORD=rec_name | FREE CURSOR=fc_name}
  •              [,ERR=err_opt]) [itemlist]
```

The GET command moves data from the database into the User Work Area.

GET retrieves either an entire record occurrence or a specified list of items within a record occurrence. The record can be identified through either a free cursor or by the current-of-record.

The maximum number of items permitted in a partial GET is 15.

If no *itemlist* is specified, DG/DBMS will GET the entire record.

GET has no effect on free cursors or set cursors. The current-of-record is set to the retrieved occurrence.

If you include the *ERR=err_opt* clause and DG/DBMS returns an error, control passes to *err_opt*.

DBMS INCLUDE Statement

Format

```
D      INCLUDE        "path_name"
```

The DBMS INCLUDE statement allows code (possibly containing DML statements) to be copied into the program during the preprocessor step.

FORTRAN 77 %INCLUDE statements are also allowed in programs using the DG/DBMS interface, but the files that they include must not contain any DML statements.

Files named in FORTRAN 77 INCLUDE statements are copied into the program after the preprocessor step.

Note that the INCLUDE statement is not used to include the subschema source in your program. The INVOKE statement does this automatically.

DBMS INCLUDE statements can be nested up to seven levels deep.

INITIATE Statement

Format

```
D     INITIATE      ( ID=tx_id [,{UPDATE | RETRIEVAL}]
  •                 [,ERR=err_opt] )
```

INITIATE starts a transaction in DG/DBMS. DG/DBMS returns a transaction number in tx_id; the program may use this number as a backup/recovery aid. Refer to the *DG/DBMS Reference Manual* for additional information about backup and recovery.

UPDATE allows the program to modify the database in this transaction. You cannot INITIATE a transaction in UPDATE mode if you readied your database in RETRIEVAL mode.

RETRIEVAL allows your program to examine, but not modify, the database.

If you include the *ERR=err_opt* clause and DG/DBMS returns an error, control passes to *err_opt*.

INITIATE has no effect on cursors. However, committed transactions of other programs can change the records to which your cursors point.

The default specification for an INITIATE statement is UPDATE.

INVOKE Statement

Format

```
D     INVOKE      (SUBSCHEMA="ss_name",SCHEMA="s_name")
```

INVOKE names the subschema to be used by the program. INVOKE also copies the subschema source code into the program. It is a nonexecutable statement that affects the declaration of the data structure to be used in accessing the database.

ss_name is any valid FORTRAN 77 subschema that the DBA has defined and bound.

s_name is the AOS/VS pathname of the database directory. It can be a relative pathname.

The INVOKE statement must precede all free cursor declarations and all other DML statements.

MEMBER Function

Format

```
D      logical = [.NOT.] MEMBER (SET=set_name)
D      IF (      [.NOT.] MEMBER (SET=set_name)) stmt
```

This function tests to see if the current-of-set is on a member record occurrence.

DML logical functions must not contain any additional logical tests.

stmt may not be a DML statement or a DO statement.

MODIFY Statement

Format

```
D      MODIFY       ({RECORD=rec_name | FREE CURSOR=fc_name}
   •                [,ERR=err_opt]) [itemlist]
```

MODIFY moves the fields specified (or the entire record if no fields are specified) from the UWA into the record occurrence indicated by either current-of-record or a free cursor. This movement overwrites the information in the database.

Up to 15 items can be specified in *itemlist*.

If no *itemlist* is specified, the entire record is modified.

If you include the *ERR=err_opt* clause and DG/DBMS returns an error, control passes to *err_opt*.

MODIFY has no effect on free cursors. The current-of-record is set to the modified occurrence. The current-of-set is set to the modified record for all sets in which the occurrence is an owner or a connected member. Note that DG/DBMS automatically reorders the members in a sorted set occurrence if you modify the sort key in a connected member.

NULL Function

Format

D   logical = *[.NOT.]* NULL (SET=set_name)

D   IF (*[.NOT.]* NULL (SET=set_name)) stmt

D   logical = *[.NOT.]* NULL
•   ({RECORD=rec_name | FREE CURSOR=fc_name})

D   IF (*[.NOT.]* NULL
•   ({RECORD=rec_name | FREE CURSOR=fc_name})) stmt

This function tests to see whether or not a current-of-set, current-of-record, or a free cursor is set to null or not. A cursor that does not identify a specific set occurrence or record occurrence is null.

DML logical functions must not contain any additional logical tests.

stmt cannot be a DML statement or a DO statement.

OWNER Function

Format

D   logical = *[.NOT.]* OWNER (SET=set_name)

D   IF (*[.NOT.]* OWNER (SET=set_name)) stmt

This function tests to see if the current-of-set is on an owner record occurrence.

DML logical functions must not contain any additional logical tests.

stmt cannot be a DML statement or a DO statement.

READY Statement

Format

D READY *[([{CONCURRENT | EXCLUSIVE}] [,{UPDATE | RETRIEVAL}]*
• *[,ERR=<err_opt>])]*

READY opens a database, allowing access to it through the subschema specified in the INVOKE statement.

CONCURRENT permits other users to access the database while you are accessing it.

EXCLUSIVE prevents any other user from accessing the database while you are accessing it. You cannot specify an exclusive READY while another user has the database open.

UPDATE permits your program to modify the database (if the subschema permits it).

RETRIEVAL allows your program to examine the database. It does not, however, allow your program to modify the database.

If you include the *ERR=<err_opt>* clause and DG/DBMS returns an error, control passes to *<err_opt>*.

The READY statement sets all cursors to null.

The default specifications for a READY statement are CONCURRENT and UPDATE.

RECONNECT Statement

Format

D RECONNECT (FREE CURSOR=fc_name, SET=set_name
• *[,ERR=err_opt]*)

DG/DBMS disconnects the record occurrence marked with free cursor fc_name from set type set_name and then connects it into the set identified by the current-of-set in set_name.

If you include the *ERR=err_opt* clause and DG/DBMS returns an error, control passes to *err_opt*.

RECONNECT has no effect on free cursors. It sets current-of-record to the reconnected member occurrence. It sets current-of-set of set_name to the reconnected member occurrence.

In addition to using RECONNECT to change the owner of a member record occurrence, you can use it to reorder members in a set occurrence.

For example, if the ORDER of the MEMBERS set is NEXT, free cursor FC1 is on MEMBER 1, and current-of-set is on MEMBER 2, then the statement

```
D     RECONNECT (FREE CURSOR=FC1, SET=MEMBERS)
```

will place MEMBER 1 after MEMBER 2.

ROLLBACK Statement

Format

D ROLLBACK *[(ERR=err_opt)]*

ROLLBACK discards changes made to the database since the start of the current transaction. You must INITIATE a new transaction before you can again access the database.

If you include the *ERR=err_opt* clause and DG/DBMS returns an error, control passes to *err_opt*.

ROLLBACK resets all cursors to their values at the time the transaction was initiated.

STORE Statement

Format

D STORE (RECORD=rec_name *[,ASSIGN=fc_name]*
• *[,ERR=err_opt]*)

STORE creates a new occurrence of the record type rec_name, using the values in the UWA. Only fields that the subschema includes in rec_name are stored; the rest are left empty.

For each AUTOMATIC set, DG/DBMS connects the occurrence in the position determined by the set's ORDER clause (FIRST, LAST, NEXT, PRIOR, KEY). DG/DBMS connects the new record to the set occurrence defined by current-of-set.

The new record occurrence becomes current-of-record and current-of-set for all sets in which the record is an owner or AUTOMATIC member. STORE has no effect on free cursors (unless you use the ASSIGN clause).

If you specify the ASSIGN clause, the free cursor fc_name marks the stored occurrence of the record type.

If you include the *ERR=err_opt* clause and DG/DBMS returns an error, control passes to *err_opt*.

End of Chapter

Chapter 12
How to Compile and Link F77/DBMS Programs

Using the Preprocessor Under AOS/VS

You execute the preprocessor in much the same way as the FORTRAN 77 compiler, but with a few additional considerations. Your searchlist must also allow access to the FORTRAN 77 compiler because the preprocessor invokes this compiler. More specifically, the macro DB.F77.CLI initiates the preprocessor; then DB.F77.CLI invokes the FORTRAN 77 compiler.

You execute the preprocessor (and then the F77 compiler) via the following command.

DB.F77*[function switches]* inputpathname

Switches

You may give DB.F77.CLI any F77.CLI switches. DB.F77.CLI passes them intact to F77.CLI. If you don't specify the /O switch, DB.F77.CLI will pass the switch /O=inputpathname to F77.CLI.

In addition, DB.F77.CLI interprets the following switches as it directs the preprocessor to process statements with a "D" in column 1 (and to pass other statements on to the F77 compiler).

/CARDFORMAT	Punched card format. All characters after column 72 are ignored (but listed). If this switch is omitted, the entire input line is considered a part of each statement.
/E=pathname	Error messages go to pathname. If the /E switch is omitted, error messages go to the file defined by the /L switch. If there is no /L switch, the current @OUTPUT file is used.
/L	Produce a listing on the current CLI LIST file.
/L=pathname	Produce a listing on pathname. If this switch is omitted, no listing is produced. Notice that the preprocessor will produce a listing only if there were errors that prevent it from invoking F77.

Temporary Files

The preprocessor creates several temporary files for its own use. Their names are as follows, where <inputpathname> is the input pathname specified to DB.F77.CLI and <?pid> is a three digit process ID number.

- inputpathname.CM
- ?pid.DBF77PP.COM.TMP
- ?pid.DBF77PP.ERR.TMP
- ?pid.DBF77PP.FCF.TMP
- ?pid.DBF77PP.INT.TMP
- ?pid.DBF77PP.OUT.TMP

Avoid using these names for any files of your own, because the preprocessor will delete them.

Linking Your FORTRAN 77 Program

You must link a successfully preprocessed and compiled FORTRAN 77 program with the DBMS runtime routines before executing it. You invoke the usual F77LINK macro and specify the two DBMS files ?DBMS32.OB and DB.F77R32.LB.

For example, suppose your F77/DBMS program TRY_DBMS contains calls to subroutines SUB1 and SUB2, and to routines in your library file MY_STUFF.LB. Then the F77LINK command would be

```
F77LINK TRY_DBMS SUB1.OB SUB2.OB MY_STUFF.LB ?DBMS32.OB DB.F77R32.LB
```

End of Chapter

Chapter 13
Sample FORTRAN 77 Application Programs

Recall the sample subschema in Figure 8-3 and the data structure in Figure 8-4. Let's observe FORTRAN 77 programmer Alice McDonald as she creates, compiles, and tests two programs to access data in the hospital information system. The assumptions of this chapter are:

- Alice's username is ALICE.
- When she logs on, her CLI.PR's process identification number (pid) is 024.
- Her working directory is :UDD:$GUEST:ALICE.
- The DG/DBMS database is in :UDD:$GUEST:ALICE:PATIENTS.
- Her searchlist allows access to FORTRAN 77 and the DG/DBMS software shown in Figure 8-2.

Program DEMO1

The FORTRAN 77 program DEMO1 is shown in Figure 13-1. Its purpose is to list all patients under the care of Dr. Brian Hackenbush.

```
      PROGRAM DEMO1

C     FIND ALL THE PATIENTS UNDER THE CARE OF DR. BRIAN HACKENBUSH.

C                                SET UP A VARIABLE FOR TRANSACTION ID
C                                   AND INVOKE THE SUBSCHEMA.
      DOUBLE PRECISION TNUM
D     INVOKE(SUBSCHEMA="PATIENT_SEARCH",
     .   SCHEMA=":UDD:$GUEST:ALICE:PATIENTS")

C                                OPEN THE DATABASE FILE;
C                                   IF AN ERROR OCCURS, GO TO 5000.
D     READY(RETRIEVAL,ERR=5000)

C                                START A TRANSACTION.
D     INITIATE(ID=TNUM,RETRIEVAL,ERR=5000)

C                                WE KNOW THE DOCTOR'S NAME, SO
C                                   FIND HIS OCCURRENCE IN THE SET.
      DOCTOR_LAST_NAME="HACKENBUSH"
      DOCTOR_FIRST_NAME="BRIAN"
D     FIND(FIRST,RECORD=DOCTOR,SET=DOCTORS_BY_NAME,
     .   WHERE(SORT KEY .EQ. UWA),ERR=5000)

C                                FIND THE DOCTOR'S FIRST TREATMENT.
D     FIND(FIRST,RECORD=TREATMENTS,SET=DOCTOR_TREATMENTS,
     .   END=4000,ERR=5000)

   10 CONTINUE

C                                NOW FIND AND GET THE PATIENT
C                                   WHO IS RECEIVING THE TREATMENT.
D     FETCH(OWNER,RECORD=PATIENT,SET=PATIENT_TREATMENTS,ERR=5000)
C
      PRINT *, "Patient name: ", PATIENT_LAST_NAME, ", ",
     .   PATIENT_FIRST_NAME

C                                NOW THE NEXT TREATMENT
D     FIND(NEXT,RECORD=TREATMENTS,SET=DOCTOR_TREATMENTS,
     .   END=4000,ERR=5000)

      GO TO 10

 4000 CONTINUE
      PRINT *, "Done: End of Patient List"
D     COMMIT(ERR=5000)
D     FINISH
      STOP
```

DG-25253

Figure 13-1. Program DEMO1 (continues)

```
 5000 CONTINUE
      WRITE (10,5001) DBSTATUS
 5001 FORMAT ("DATABASE ERROR ENCOUNTERED - DBSTATUS IS ", 06)
C     NOTE: SINCE WE ARE NOT MODIFYING THE DATABASE, ROLLBACK IS
C        NOT REALLY NECESSARY (WE INCLUDE IT FOR ILLUSTRATION).
D     ROLLBACK
D     FINISH
      STOP

      END
```

DG-25253

Figure 13-1. Program DEMO1 (concluded)

Alice now has to create DEMO1.OB from DEMO1.F77. This occurs in two steps:

- The preprocessor creates a temporary F77 source program file from DEMO1.F77.
- The F77 compiler creates DEMO1.OB from the temporary F77 source program file.

The name of this temporary file is ?pid.DBF77P.OUT.TMP. The value of <pid> is not 024 because AOS/VS creates a son process whose father (here, CLI.PR) is 024. In Alice's case, assume that <pid> is 058.

Her *one* CLI command is

```
DB.F77 DEMO1
```

Recall from the previous chapter that DB.F77.CLI also invokes the F77 compiler.

File ?058.DBF77P.OUT.TMP is in shown Figure 13-2. When the F77 compiler processes it, Alice might see a warning (severity level 1) message about mixing CHARACTER and nonCHARACTER data elements in a COMMON block. She ignores such messages, and so should you.

Macro DB.F77.CLI does not save the temporary files it creates (whose names begin with ?<pid>). If you interrupt it at the proper time with the CTRL-C CTRL-B sequence you'll have access to these files.

```
      PROGRAM DEMO1

C     FIND ALL THE PATIENTS UNDER THE CARE OF DR. BRIAN HACKENBUSH.

C                                 SET UP A VARIABLE FOR TRANSACTION ID
C                                    AND INVOKE THE SUBSCHEMA.
      DOUBLE PRECISION TNUM
C     INVOKE(SUBSCHEMA="PATIENT_SEARCH",
C    .   SCHEMA=":UDD:$GUEST:ALICE:PATIENTS")

      INTEGER*2 DB_SSIG(4)/-23250,8364,-23250,8541/
      COMMON/DBSTATUS/DBSTATUS
      INTEGER*4 DBSTATUS
      INTEGER*4 DBBADDR
      EXTERNAL DBBADDR


C  SUBSCHEMA NAME IS "PATIENT_SEARCH"
C     ALLOWS ERASE GET MODIFY STORE

C  SET DEFINITION SECTION.

C     SET = DOCTORS_BY_NAME
C           ALLOWS RECONNECT
C           OWNER IS SYSTEM
C           MEMBER IS DOCTOR
C           AUTOMATIC MANDATORY
C           ORDER IS SORTED BY KEY ASCENDING
C              KEYS ARE:
C                       DOCTOR_LAST_NAME
C                       DOCTOR_FIRST_NAME
C              DUPLICATES ALLOWED
C           MEMBER LIMIT IS NONE

C     SET = PATIENTS_BY_NAME
C           ALLOWS RECONNECT
C           OWNER IS SYSTEM
C           MEMBER IS PATIENT
C           AUTOMATIC MANDATORY
C           ORDER IS SORTED BY KEY ASCENDING
C              KEYS ARE:
C                       PATIENT_LAST_NAME
C                       PATIENT_FIRST_NAME
C              DUPLICATES ALLOWED
C           MEMBER LIMIT IS NONE
```

DG-25254

Figure 13-2. Temporary F77 Program ?058.DBF77P.OUT.TMP (continues)

```
C     SET = PATIENT_TREATMENTS
C           ALLOWS RECONNECT
C           OWNER IS PATIENT
C           MEMBER IS TREATMENTS
C           AUTOMATIC MANDATORY
C           ORDER IS NEXT
C           MEMBER LIMIT IS NONE

C     SET = DOCTOR_TREATMENTS
C           ALLOWS RECONNECT
C           OWNER IS DOCTOR
C           MEMBER IS TREATMENTS
C           AUTOMATIC MANDATORY
C           ORDER IS NEXT
C           MEMBER LIMIT IS NONE

C  RECORD DEFINITION SECTION.

C  RECORD = DOCTOR     ALLOWS ERASE GET MODIFY STORE

      CHARACTER*1 DOCTOR
      COMMON/DOCTOR/DOCTOR_LAST_NAME
             CHARACTER*25 DOCTOR_LAST_NAME
C                CONTENTS: CHAR*25L     ALLOWS GET MODIFY

      COMMON/DOCTOR/DOCTOR_FIRST_NAME
             CHARACTER*20 DOCTOR_FIRST_NAME
C                CONTENTS: CHAR*20L     ALLOWS GET MODIFY

      COMMON/DOCTOR/SPECIALTY
             CHARACTER*15 SPECIALTY
C                CONTENTS: CHAR*15L     ALLOWS GET MODIFY

      COMMON/DOCTOR/INFO
             CHARACTER*40 INFO
C                CONTENTS: CHAR*40L     ALLOWS GET MODIFY

      COMMON/DOCTOR/BEEPER
             INTEGER*2 BEEPER
C                CONTENTS: NUMERIC      ALLOWS GET MODIFY
C                RANGE:     -9999 TO +9999

      EQUIVALENCE (DOCTOR,DOCTOR_LAST_NAME)
```

DG-25254

Figure 13-2. Temporary F77 Program ?058.DBF77P.OUT.TMP (continued)

```
C  RECORD = PATIENT     ALLOWS ERASE GET MODIFY STORE

      CHARACTER*1 PATIENT
      COMMON/PATIENT/PATIENT_LAST_NAME
             CHARACTER*20 PATIENT_LAST_NAME
C                CONTENTS: CHAR*20L     ALLOWS GET MODIFY

      COMMON/PATIENT/PATIENT_FIRST_NAME
             CHARACTER*14 PATIENT_FIRST_NAME
C                CONTENTS: CHAR*14L     ALLOWS GET MODIFY

      COMMON/PATIENT/WARD
             CHARACTER*4 WARD
C                CONTENTS: CHAR*4L      ALLOWS GET MODIFY

      COMMON/PATIENT/ROOM
             INTEGER*2 ROOM
C                CONTENTS: NUMERIC      ALLOWS GET MODIFY
C                RANGE:    +0 TO +999

      EQUIVALENCE (PATIENT,PATIENT_LAST_NAME)

C  RECORD = TREATMENTS     ALLOWS ERASE GET MODIFY STORE

      CHARACTER*1 TREATMENTS
      COMMON/TREATMENTS/DISEASE
             CHARACTER*100 DISEASE
C                CONTENTS: CHAR*100L    ALLOWS GET MODIFY

      COMMON/TREATMENTS/MEDICATION
             CHARACTER*25 MEDICATION (5)

C                CONTENTS: CHAR*25L     ALLOWS GET MODIFY

      COMMON/TREATMENTS/DIET
             CHARACTER*200 DIET
C                CONTENTS: CHAR*200L    ALLOWS GET MODIFY

      COMMON/TREATMENTS/SPECIAL_INSTRUCTIONS
             CHARACTER*40 SPECIAL_INSTRUCTIONS (5)
C                CONTENTS: CHAR*40L     ALLOWS GET MODIFY

      EQUIVALENCE (TREATMENTS,DISEASE)

C  END OF FORTRAN 77 "PATIENT_SEARCH" SUBSCHEMA.
```

DG-25254

Figure 13-2. Temporary F77 Program ?058.DBF77P.OUT.TMP (continued)

```
C                               OPEN THE DATABASE FILE;
C                                  IF AN ERROR OCCURS, GO TO 5000.
C     READY(RETRIEVAL,ERR=5000)

      CALL DBREADY (2,":UDD:$GUEST:ALICE:PATIENTS","PATIENT_SEARCH",
     .              DB_SSIG,0,0,0,0)
      IF (DBSTATUS .NE. 0) GO TO 5000
C                               START A TRANSACTION.
C     INITIATE(ID=TNUM,RETRIEVAL,ERR=5000)

      CALL DBSTARTX (DB_SSIG,2,0,TNUM)
      IF (DBSTATUS .NE. 0) GO TO 5000
C                               WE KNOW THE DOCTOR'S NAME, SO
C                                  FIND HIS OCCURRENCE IN THE SET.
      DOCTOR_LAST_NAME="HACKENBUSH"
      DOCTOR_FIRST_NAME="BRIAN"
C     FIND(FIRST,RECORD=DOCTOR,SET=DOCTORS_BY_NAME,
C    .   WHERE(SORT KEY .EQ. UWA),ERR=5000)

      CALL DBFSKEY (DB_SSIG,2,0,1,1,DBBADDR(DOCTOR))
      IF (DBSTATUS .NE. 0) GO TO 5000
C                               FIND THE DOCTOR'S FIRST TREATMENT.
C     FIND(FIRST,RECORD=TREATMENTS,SET=DOCTOR_TREATMENTS,
C    .   END=4000,ERR=5000)

      CALL DBFMSEQN (DB_SSIG,3,0,3,3,DBBADDR(TREATMENTS))
      IF (DBSTATUS .EQ. 017410K) GO TO 4000
      IF ((DBSTATUS .NE. 0).AND.(DBSTATUS .NE. 017410K)) GO TO 5000
   10 CONTINUE

C                               NOW FIND AND GET THE PATIENT
C                                  WHO IS RECEIVING THE TREATMENT.
C     FETCH(OWNER,RECORD=PATIENT,SET=PATIENT_TREATMENTS,ERR=5000)
      CALL DBFOWNER (DB_SSIG,2,16,2,4,DBBADDR(PATIENT))
      IF (DBSTATUS .NE. 0) GO TO 5000
C
      PRINT *, "Patient name: ", PATIENT_LAST_NAME, ", ",
     .   PATIENT_FIRST_NAME

C                               NOW THE NEXT TREATMENT
C     FIND(NEXT,RECORD=TREATMENTS,SET=DOCTOR_TREATMENTS,
C    .   END=4000,ERR=5000)

      CALL DBFMSEQN (DB_SSIG,3,8,3,3,DBBADDR(TREATMENTS))
      IF (DBSTATUS .EQ. 017410K) GO TO 4000
      IF ((DBSTATUS .NE. 0).AND.(DBSTATUS .NE. 017410K)) GO TO 5000
      GO TO 10
```

DG-25254

Figure 13-2. Temporary F77 Program ?058.DBF77P.OUT.TMP (continued)

```
       4000 CONTINUE
            PRINT *, "Done: End of Patient List"
      C     COMMIT(ERR=5000)
            CALL DBSTOPX (DB_SSIG,2)
            IF (DBSTATUS .NE. 0) GO TO 5000
      C     FINISH
            CALL DBFINISH (DB_SSIG,0)
            STOP

       5000 CONTINUE
            WRITE (10,5001) DBSTATUS
       5001 FORMAT ("DATABASE ERROR ENCOUNTERED - DBSTATUS IS ", 06)
      C     NOTE: SINCE WE ARE NOT MODIFYING THE DATABASE, ROLLBACK IS
      C        NOT REALLY NECESSARY (WE INCLUDE IT FOR ILLUSTRATION).
      C     ROLLBACK
            CALL DBROLLBK (DB_SSIG,0)
      C     FINISH
            CALL DBFINISH (DB_SSIG,0)
            STOP

            END
```

DG-25254

Figure 13-2. Temporary F77 Program ?058.DBF77P.OUT.TMP (concluded)

The CLI command Alice gives to create DEMO1.PR from DEMO1.OB, the DG/DBMS libraries, and the FORTRAN 77 libraries is

F77LINK DEMO1 ?DBMS32.OB DB.F77R32.LB

Alice gives the CLI command

XEQ DEMO1

and sees the following results on her console.

Patient name: MCINTOSH , MARY
Patient name: VERLUCCI , ENRICO
Patient name: KELLEY , JOHN
Done: End of Patient List

STOP

Program DEMO2.F77

One way to interpret program DEMO1.F77 is in terms of Figure 8-4. DEMO1 uses DOCTOR and TREATMENTS records to find patients related to a specific doctor. The purpose of program DEMO2.F77 is to use PATIENT and TREATMENTS records to find doctors related to a specific patient. The patient's name is John Kelley. Program DEMO2.F77 is shown in Figure 13-3.

```
      PROGRAM DEMO2

C     FIND ALL THE DOCTORS TREATING JOHN KELLEY.

C                             SET UP A VARIABLE FOR TRANSACTION ID
C                                AND INVOKE THE SUBSCHEMA.
      DOUBLE PRECISION TNUM
D     INVOKE(SUBSCHEMA="PATIENT_SEARCH",
     .   SCHEMA=":UDD:$GUEST:ALICE:PATIENTS")

C                             OPEN THE DATABASE FILE;
C                                IF AN ERROR OCCURS GO TO 5000
D     READY(RETRIEVAL,ERR=5000)

C                             START A TRANSACTION
D     INITIATE(ID=TNUM,RETRIEVAL,ERR=5000)

C                             WE KNOW THE PATIENT'S NAME;
C                                 FIND HIS OCCURRENCE IN THE SET.
      PATIENT_LAST_NAME="KELLEY"
      PATIENT_FIRST_NAME="JOHN"
D     FIND(FIRST,RECORD=PATIENT,SET=PATIENTS_BY_NAME,
     .   WHERE(SORT KEY .EQ. UWA),ERR=5000)

C                             FIND THE PATIENT'S FIRST TREATMENT.
D     FIND(FIRST,RECORD=TREATMENTS,SET=PATIENT_TREATMENTS,
     .  END=4000,ERR=5000)
C
   10 CONTINUE
C                             NOW FIND AND GET THE DOCTOR
C                                 WHO IS GIVING THE TREATMENT.
D     FETCH(OWNER,RECORD=DOCTOR,SET=DOCTOR_TREATMENTS,ERR=5000)
C
      PRINT *,"Doctor name: ",DOCTOR_LAST_NAME,", ",DOCTOR_FIRST_NAME

C                             NOW THE NEXT TREATMENT
D     FIND(NEXT,RECORD=TREATMENTS,SET=PATIENT_TREATMENTS,
     .   END=4000,ERR=5000)
      GO TO 10
 4000 CONTINUE
      PRINT *,"Done: End of Doctor List"
D     COMMIT(ERR=5000)
D     FINISH
      STOP
C

 5000 CONTINUE
      WRITE(10,5001)DBSTATUS
 5001 FORMAT("DATABASE ERROR ENCOUNTERED - DBSTATUS IS ",O6)
C     NOTE: SINCE WE ARE NOT MODIFYING THE DATABASE, ROLLBACK
C           IS NOT REALLY NECESSARY (WE INCLUDE IT FOR ILLUSTRATION)
D     ROLLBACK
D     FINISH
      STOP
      END
```

DG-25255

Figure 13-3. Program DEMO2.F77

The CLI commands Alice gives to compile, link, and execute DEMO1.F77 are

```
DB.F77 DEMO2
F77LINK DEMO2 ?DBMS32.OB DB.F77R32.LB
XEQ DEMO2
```

At runtime, DEMO2.PR displays the following.

Doctor name: ROSENBERG   , AARON
Doctor name: HACKENBUSH , BRIAN
Done: End of Doctor List
STOP

End of Chapter

Chapter 14
DBMS Usage Considerations

Character and Bit Strings

When using bit or character data in FORTRAN 77, observe some important rules:

1. Declaring a character string as right-justified only affects the way the string is manipulated by DG/DBMS and *not* the way the string is manipulated by FORTRAN 77. FORTRAN 77 treats all strings as left-justified. Thus, if a string named TEMPSTR is declared as CHAR*6L in the schema and CHAR*8R in the subschema, the operations

   TEMPSTR = "ABCDEFGHIJ"
   STORE record containing TEMPSTR
   GET record containing TEMPSTR

   will return the value "CDEFGH" to TEMPSTR.

2. If a data item is declared to be BIT*n in the subschema, and the area allocated for the data item is larger than the length specified, the extra bit positions are undefined. For example, an item declared as BIT*4 with a data type of INTEGER*2 will have the 12 rightmost bits left undefined.

3. Schema bit data that is specified to be NUMERIC in the subschema is treated as a right-justified string instead of left-justified, as above. The entire data item will always be defined as the (bit) length is determined by the type of the data item. If the source string is shorter than the destination, zero bits are used to pad on the left. Notice that this is not a sign extension.

Separate Compilation of Subroutines

Since FORTRAN 77 allows the linking of separately compiled modules, you must follow some simple conventions to make your program execute correctly:

- There must be an INVOKE statement in every subroutine that uses any DML statements. The information in the INVOKE statement is required by the preprocessor to make the association of symbolic names to numeric identifiers.
- The INVOKE statements in each separately compiled subroutine must refer to the same schema and subschema (The exception to this is described in the next section). Each process can operate on only one database, using one subschema, at a time. Runtime errors will occur if this rule is ignored.
- Every subroutine that uses free cursors must specify all the free cursors that it uses. The subroutine (or main program) that contains the READY statement must specify *all* free cursors used in *any* subroutine. If this is not done, a subroutine using a free cursor not specified in the routine doing the READY statement will receive an error indication.
- Notice that only one READY statement is needed, and that READY is a *global* operation. Thus, a program that uses several subroutines to perform DML operations still needs only one READY statement to be executed. A READY is not required in each subroutine. If a second READY is to be executed to change usage or mode or to change databases (see the next section), a FINISH statement is required to close the database before another READY statement can be executed.

Accessing Multiple Databases

As stated in the previous section, a process can access only one database (schema) through one subschema at a time. This does not preclude one process from using more than one schema and subschema, as long as it is done correctly. Whether one, or more than one, schema and subschema is involved, nesting of READY statements is not allowed.

That is, the following sequence is illegal.

READY ... READY ... FINISH ... FINISH

But, this sequence is legal.

READY ... FINISH ... READY ... FINISH

The INVOKE statements associated with the different READY statements can refer to the same or to different schemas and subschemas. Therefore, by using separately compiled subroutines that INVOKE different schemas or subschemas (or both), a single process can legally operate on more than one schema and subschema. The only restriction is that operation on one subschema must be finished before another is readied.

Preprocessor-Generated Symbolic Names

The preprocessor generates FORTRAN code, and this code contains various symbolic names. Most of the names are names of data items and records defined by the subschema. There are, however, several names that are generated by the preprocessor and that are not under your control. These names must not be used in your FORTRAN program. Similarly, they must not be chosen as set, record or data item names. These generated names are as follows.

- *Runtime Routine Names*

  DBASSIGN
  DBBADDR
  DBCHECKX
  DBCONECT
  DBCONMEM
  DBDISCON
  DBEMPTY
  DBERASE
  DBFCURSR
  DBFDFLDS
  DBFDSKEY
  DBFFLDS
  DBFINISH
  DBFMORDN
  DBFMSEQN
  DBFOWNER
  DBFSKEY
  DBGETITM
  DBGETREC
  DBMEMBER
  DBMODITM
  DBMODREC
  DBNULL
  DBOWNER
  DBPUTREC
  DBREADY
  DBRECON
  DBROLLBK
  DBSTARTX
  DBSTOPX

- *Local Vectors*

  DB_FCVEC
  DB_ITEM_LIST
  DB_SSIG
  DB_USING_LIST

- *External Names in Your Programs*

  DBERROR
  DBSTATUS

Other Restrictions

Observe these restrictions as you create F77 programs that interface with DG/DBMS.

1. One-dimensioned arrays are permitted for items. Multidimensioned items are not allowed. If necessary, you can implement them by using EQUIVALENCE statements.
2. Since the schema has no equivalent data type, COMPLEX data items are not supported. You can implement them by using EQUIVALENCE statements.
3. Multitasking is not supported.
4. Since each DML statement can be translated to more than one FORTRAN statement, labels are not allowed on DML statements. Use a labeled CONTINUE statement just before any DML statement you want to label.
5. Records that have items with CHARACTER data type and items with non-CHARACTER type will cause a compiler warning to be issued. Allowing mixed data types in the same COMMON block is an extension to ANSI FORTRAN 77. This message is only a warning and can be disregarded.
6. All F77 OPEN statements must appear after all data declarations. This means that the INVOKE statement must be BEFORE any F77 OPEN statements, since INVOKE inserts data declarations into the program.
7. Only one program module is handled by the preprocessor. This differs from standard FORTRAN 77 which allows multiple program units to be compiled in a single file.
8. FORTRAN 77 subschemas require all names to be unique within the first eight characters. This is more restrictive than other F77 names that need to be unique within the first 32 characters. The DDF enforces this rule.
9. The preprocessor interface does not process F77 %INCLUDE files. Therefore, these %INCLUDE files must not contain any DML statements. The DBMS "INCLUDE" statement can be used to include DML statements.
10. You must declare the variables **tx_id** and **posit** as DOUBLE PRECISION, and the variable **status** as INTEGER*2. Erroneous results or runtime errors can be obtained by your program if these requirements are not observed.
11. FORTRAN 77 currently allows a maximum of 256 external symbols in a single subprogram. Keep this in mind when designing a subschema or a program. Both records and free cursors are COMMON blocks, which are external. Other external symbols include DBSTATUS and each runtime routine called as a result of a DML statement.

12. Caution is advised when using bit data items stored in real or double precision variables. When these data items are used in assignment statements, the destination data item is assigned in normalized floating-point form. Thus, the statement

```
BITS_2 = BITS_1
```

could result in BITS_2 being set to all zero bits, if BITS_1 happens to have all zero bits in the mantissa portion of the floating-point number. Similarly, in IF statements, FORTRAN 77 considers floating-point numbers to be equal (.EQ.), if they both have all zero bits in the mantissa. These problems can be avoided by using the FLD function.

13. Because of the way FORTRAN 77 handles character strings as arrays of other variable types, all character strings are allocated an even number of bytes of storage. This means that odd-length character strings have a default FORTRAN 77 length, which is 1 byte longer than the schema length. In records where the schema length of the record plus the number of odd-length character strings is greater than 1024, the default FORTRAN 77 subschema length will exceed the maximum record length. In such a subschema, item(s) will have to be either excluded, or explicitly described to DDF with shorter lengths. (In this case, truncation could become a problem.) This restriction is an important database design consideration only in the case of very large records with sufficient numbers of odd-length character strings.

All other DBMS restrictions are listed in the *DBMS Reference Manual*.

End of Chapter

Chapter 15
DG/DBMS Error Messages

The preprocessor detects many errors. They report them with one or more of the following error messages.

A terminal error has occurred, preprocessing abandoned
ASSIGN clause not allowed in this statement
DIRECTION parameter not allowed in this statement
DML statement requires DIRECTION parameter
DML statement requires FREE CURSOR clause
DML statement requires RECORD clause
DML statement requires schema name
DML statement requires SET clause
DML statement requires subschema name
DML statement requires TRANSACTION ID parameter
DML statement requires TRANSACTION STATUS parameter
Data item may not be subscripted
Duplicate ASSIGN parameter
Duplicate DIRECTION parameter
Duplicate END parameter
Duplicate ERR parameter
Duplicate FREE CURSOR definition
Duplicate FREE CURSOR parameter
Duplicate ID parameter
Duplicate MODE parameter
Duplicate RECORD parameter
Duplicate SCHEMA parameter
Duplicate SET parameter
Duplicate SUBSCHEMA parameter
Duplicate USAGE parameter
Duplicate WHERE parameter
END clause not allowed in this statement
ERR clause not allowed in this statement
Error in argument list to preprocessor
Error opening INCLUDE file
Error opening subschema source file
Error while creating temporary file
Error while deleting temporary file
Error while determining existence of source file
Error while determining existence of temporary file
FORTRAN source file not found
FREE CURSOR clause not allowed in this statement
FREE CURSOR dimension must be integer
FREE CURSOR NAME may not be dimensioned
FREE CURSOR name requires subscript
FREE CURSOR not defined
FREE CURSOR specification follows a READY statement

Field does not have GET access
Field does not have MODIFY access
Field name undefined
First character is an underscore
INCLUDE statement requires a file name
INTERNAL ERROR: BAD PRODUCTION NUMBER
INTERNAL ERROR: CG_ASSIGN CALL IS UNKNOWN
INTERNAL ERROR: EMPTY PARSE STACK
INTERNAL ERROR: HAS_THIS_EXT ROUTINE DETECTS ERROR
INTERNAL ERROR: INTERNAL METADATA ERROR
INTERNAL ERROR: INVALID FC NUMBER IN GET_FC_ORD CALL
INTERNAL ERROR: INVALID MESSAGE NUMBER
INTERNAL ERROR: INVALID RSE CODE INPUT TO CG_FF
INTERNAL ERROR: INVALID STMT# INPUT TO CODE_GENERATOR
INTERNAL ERROR: PARSE STACK OVERFLOW
INTERNAL ERROR: PARSE STACK UNDERFLOW
INTERNAL ERROR: UNDECIPHERABLE FIND OR FETCH STATEMENT
INTERNAL ERROR: UNEXPECTED ERROR DETECTED BY MAIN
ITEM list not allowed in this statement
Illegal character in a token
Illegal character in col 2-6 of a non-continuation line
Illegal combination of RECORD, SET, and FREE CURSOR clauses
Illegal label format
Illegal number format
Illegal operator
Internal table overflow caused by too many FREE CURSOR NAMES
Invalid direction on FIND or FETCH
Invalid DML statement
Invalid or incomplete WHERE clause
Invalid relational operator on FIND or FETCH
Invalid syntax in this statement
Invalid token format
Keyword or identifier is longer than 32 characters
Maximum nesting of INCLUDE files exceeded
Missing closing quotation mark
MODE parameter not allowed in this statement
More than 255 FREE CURSORS
No FORTRAN statement following IF
No INVOKE statement before this statement
No file name specified for /O switch
Only one INVOKE statement allowed
RECORD clause not allowed in this statement
Record does not have ERASE access
Record does not have GET access
Record does not have MODIFY access
Record does not have STORE access
Record is not a member of set type
Record is not an owner of set type
Record name undefined
SET clause not allowed in this statement
SORT KEY parameter invalid, set is not sorted
Schema name not allowed in this statement
Set does not have CONNECT access
Set does not have DISCONNECT access
Set does not have RECONNECT access

Set name undefined
Source file exists but it cannot be opened
Subschema name not allowed in this statement
Subschema name not found
Token is longer than 128 characters
Too many items in USING/ITEM list
TRANSACTION ID parameter not allowed in this statement
TRANSACTION STATUS parameter not allowed in this statement
Unable to open temporary file which has been created
Unable to reference subschema information
USAGE parameter not allowed in this statement
WHERE clause not allowed in this statement

End of Chapter

Appendix A
Runtime Memory Configuration

This appendix describes changes you can make to the FORTRAN 77 heap and stack organization.

Heap and Stack Organization

There are a number of options available during Link time that provide a way to alter how the runtime initializer configures memory (i.e., maps additional unshared pages into the program context by issuing ?MEMI system calls to set up the stack and heap). We use the term "map" in this appendix to mean making ?MEMI calls. This information is useful if your program is linked for a very large address space and is designed to make ?MEMI system calls.

Unless otherwise stated, the *stack* we refer to in this section is the stack of the initial (or default) task.

The *heap* refers to the area in memory from which F77 runtime routines allocate temporary storage and task stacks.

A *fixed stack* is a stack with a stack limit that does not change. The stack may grow until reaching this limit, at which time the program aborts.

A *dynamic stack* is a stack that uses the top of the unshared address space as the stack limit. The stack fault handler will issue a ?MEMI system call to map additional pages, and reset the stack limit as the program's stack requirements grow.

A *fixed heap* is a heap organization with a fixed amount of storage available for allocation. This storage is set aside just for the heap. The system allocates storage from this heap until space runs out, at which time the program aborts.

A *dynamic heap* is a heap that uses the top of the unshared address space (highest addresses) for storage. The heap management runtime routines map additional unshared pages with the ?MEMI call as the program's heap requirements grow.

A *dynamic stack/heap* is a memory organization in which the stack and the heap share the same area of memory. The heap occupies the higher addresses and grows downward. The stack occupies the lower addresses. The stack limit is the current bottom of the heap area; it serves as the boundary between the stack and the heap. The stack limit is adjusted down and up as the heap grows and shrinks. The program aborts if the stack reaches the stack limit, or if the heap bottom (stack limit) becomes less than the current stack pointer. F77 multitasking programs cannot use a dynamic stack/heap.

A program requires a heap if it uses F77 multitasking or I/O facilities.

Memory Configuration Options

The stack and heap are set up by the runtime initializer according to

- Whether or not the program requires a heap.
- Whether or not the following symbols have been defined (at Link time) to have values other than -1 (which signifies that the symbol is not defined):

  .RESERVE

  .STKSIZE

  .HPSIZE

  .STORAGE

The value of .RESERVE (if other than -1) denotes the number of pages (1024. words) the initializer is to leave unmapped between the unshared and the shared partitions. The program can change the number of unshared pages by issuing the ?MEMI system call. The address of the unmapped area is ?SBOT-1024.*.RESERVE. All addresses below this are mapped as unshared.

The value of .STKSIZE (if other than -1) denotes the size of a fixed stack of .STKSIZE words.

The value of .HPSIZE (if other than -1) denotes the size of a fixed heap of .HPSIZE words.

The value of .STORAGE (if other than -1) denotes the size of an area of storage of .STORAGE words to be used as a dynamic stack/heap. Multitasking programs cannot use a dynamic stack/heap.

Default Values

By default, the FORTRAN 77 runtime routines have the following values for the four heap and stack specifiers.

No Multitasking (No /TASKS= F77LINK switch)		**Multitasking (/TASKS= F77LINK switch)**	
.RESERVE	−1	.RESERVE	−1
.STKSIZE	−1	.STKSIZE	100000 (decimal)
.HPSIZE	−1	.HPSIZE	−1
.STORAGE	−1	.STORAGE	−1

You may override these values if you wish, but do so very carefully.

Assigning Values

You can set the value of these four symbols by assigning values in the F77LINK command, or by linking with object files (.OBs) produced by the macroassembler (MASM) that define their values. For example, you can specify .STKSIZE to have a value of 10000 (decimal) in one of two ways.

1. Specify a value for .STKSIZE in a F77LINK command.

   F77LINK */switches* MAIN_PROGRAM .STKSIZE/VAL=10000 ...

2. Create an assembly language program that gives .STKSIZE its value, assemble it, and then include it in a F77LINK command.

```
.TITLE   ASGN_STK          ; Module to assign stack size
.ENT     .STKSIZE
.STKSIZE = 10000.
.END
```

X MASM ASGN_STK
F77LINK/*switches* MAIN_PROGRAM ASGN_STK ...

Valid Configuration Combinations

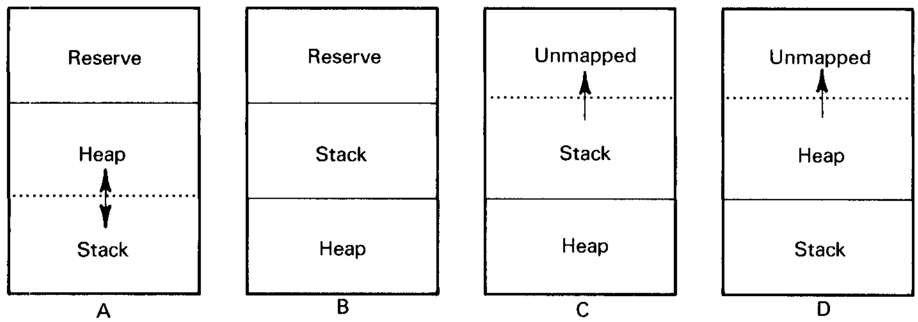
F77 supports only certain combinations of values for the symbols .RESERVE, .STKSIZE, .HPSIZE, and .STORAGE. For example, you cannot define values other than -1 for .STKSIZE, .HPSIZE, and .STORAGE all at the same time. This results in an unpredictable initialization.

The following list describes the valid combinations, and the resulting actions by the initializer. Figure A-1 illustrates these combinations. The description for *none* is the default behavior.

<none>	If a heap is required, designate all available pages as unshared with the ?MEMI call, and use as a dynamic stack/heap. (A in Figure A-1) If no heap is required, use ?MEMI to map just enough pages for an initial stack. The stack fault handler will map additional unshared pages as required. (C in Figure A-1)
.RESERVE	Map all but .RESERVE pages as unshared and use as a dynamic stack/heap. .RESERVE pages are available for remapping if the program issues ?MEMI. (A in Figure A-1)
.STKSIZE	If a heap is required, map just enough pages for a fixed stack of .STKSIZE words and an initial heap. Heap runtimes will map additional unshared pages as needed for the heap. The program must not map additional unshared pages. (D in Figure A-1) If no heap is required, map just enough pages for an initial stack of .STKSIZE words. The stack fault handler will map additional unshared pages as required. The program must not map additional unshared pages. (C in Figure A-1)
.STORAGE	Map enough pages for a dynamic stack/heap of .STORAGE words. Remaining pages are available for remapping by the program. (A in Figure A-1)
.HPSIZE	Map enough pages for a fixed heap of .HPSIZE words and an initial stack. The stack fault handler will map additional unshared pages as required. The program must not map additional unshared pages. (C in Figure A-1)
.RESERVE,.STKSIZE	Map all but .RESERVE pages. Use .STKSIZE words for fixed stack and the rest as a fixed heap. .RESERVE pages are available for mapping by program. (B in Figure A-1)
.RESERVE,.HPSIZE	Map all but .RESERVE pages. Use .HPSIZE words as a fixed heap and the rest as a fixed stack. .RESERVE pages are available for mapping by program. (B in Figure A-1)

.STKSIZE,.HPSIZE — Map just enough pages for a fixed stack of .STKSIZE words and a fixed heap of .HPSIZE words. Remaining pages are available for mapping by program. (B in Figure A-1)

Figure A-1 depicts the four basic memory configurations. Each configuration illustrates the portion of the address space between ?NMAX (the bottom of the figures) and ?SBOT (the top of the figures). The dotted lines (...) indicate boundaries, which change in the directions indicated during the execution of the program.



A — A dynamic stack/heap; heap grows and shrinks by moving stack limit. Multitasking programs cannot use this configuration.

B — A fixed size stack, and a fixed size heap.

C — A fixed size heap and a dynamic stack; stack fault handler maps additional pages as required.

D — A fixed size stack and a dynamic heap; heap manager maps additional pages as required.

ID-00220

Figure A-1. Memory Configurations

End of Appendix

Index

Within this index, "f" or "ff" after a page number means "and the following page" (or "pages"). In addition, primary page references for each topic are listed first. Commands, calls, and acronyms are in uppercase letters (e.g., CREATE); all others are lowercase.