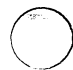
Data General

FORTRAN 77 Environment Manual (AOS)

093-000273-00



FORTRAN 77 Environment Manual (AOS)

093-000273-00

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. 093-000273
©Data General Corporation, 1983
All Rights Reserved
Printed in the United States of America
Revision 00, September 1983
Licensed Material - Property of Data General Corporation

NOTICE

DATA GENERAL CORPORATION (DGC) HAS PREPARED THIS DOCUMENT FOR USE BY DGC PERSONNEL, LICENSEES, AND CUSTOMERS. THE INFORMATION CONTAINED HEREIN IS THE PROPERTY OF DGC; AND THE CONTENTS OF THIS MANUAL SHALL NOT BE REPRODUCED IN WHOLE OR IN PART NOR USED OTHER THAN AS ALLOWED IN THE DGC LICENSE AGREEMENT.

DGC reserves the right to make changes in specifications and other information contained in this document without prior notice, and the reader should in all cases consult DGC to determine whether any such changes have been made.

THE TERMS AND CONDITIONS GOVERNING THE SALE OF DGC HARDWARE PRODUCTS AND THE LICENSING OF DGC SOFTWARE CONSIST SOLELY OF THOSE SET FORTH IN THE WRITTEN CONTRACTS BETWEEN DGC AND ITS CUSTOMERS. NO REPRESENTATION OR OTHER AFFIRMATION OF FACT CONTAINED IN THIS DOCUMENT INCLUDING BUT NOT LIMITED TO STATEMENTS REGARDING CAPACITY, RESPONSE-TIME PERFORMANCE, SUITABILITY FOR USE OR PERFORMANCE OF PRODUCTS DESCRIBED HEREIN SHALL BE DEEMED TO BE A WARRANTY BY DGC FOR ANY PURPOSE, OR GIVE RISE TO ANY LIABILITY OF DGC WHATSOEVER.

This software is made available solely pursuant to the terms of a DGC license agreement which governs its use.

CEO, DASHER, DATAPREP, ECLIPSE, ENTERPRISE, INFOS, microNOVA, NOVA, PROXI, SUPERNOVA, PRESENT, ECLIPSE MV/4000, ECLIPSE MV/6000, ECLIPSE MV/8000, TRENDVIEW, SWAT, GENAP, and **MANAP** are U.S. registered trademarks of Data General Corporation, and **AZ-TEXT, DG/L, ECLIPSE MV/10000, GW/4000, GDC/1000, REV-UP, XODIAC, DEFINE, SLATE, microECLIPSE, BusiPEN, BusiGEN** and **BusiTEXT** are U.S. trademarks of Data General Corporation.

FORTRAN 77
Environment Manual
(AOS)
093-000273-00

Revision History:	Effective with:
Original Release - September 1983	FORTRAN 77 Rev. 2.10

Preface

As a programmer fluent in FORTRAN 77 (F77) and familiar with the Advanced Operating System (AOS), you will find this environment manual a useful companion to the *FORTRAN 77 Reference Manual* (093-000162).

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 (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 FORTRAN 5, DG/L™, and PL/I programs can access.
Chapter 7	Gives several hints about writing better FORTRAN 77 programs.
Chapter 8	Introduces the technique of writing large programs with overlays and gives a sample program with overlaid subprograms.

Related Documentation

Other manuals you may find useful are as follows.

Manual Title	Manual No.
Advanced Operating System (AOS) Programmer's Manual	093-000120
Command Line Interpreter (CLI) User's Manual (AOS and AOS/VS)	093-000122
FORTRAN 5 Programmer's Guide (AOS)	093-000154
Advanced Operating System (AOS) Macroassembler (MASM) Reference Manual	093-000192
Advanced Operating System (AOS) Operator's Guide	093-000194
Advanced Operating System (AOS) Link and Library File Editor (LFE) User's Manual	093-000254
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: required_1 required_2 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

A Software Summary . . . 1-1
The Significance of AOS . . . 1-3
The Significance of Link and the Runtime Libraries . . . 1-3
The Significance of the Release and Update Notices . . . 1-6

Chapter 2 - Utility Runtime Routines

Documentation Categories . . . 2-1
DATE . . . 2-2
ERRCODE . . . 2-3
ERRTEXT . . . 2-7
EXIT . . . 2-10
RANDOM . . . 2-11
TIME . . . 2-18

Chapter 3 - System Call Interface

Structure . . . 3-1
Implementing ISYS: An Initial Approach . . . 3-2
    Sample Program . . . 3-3
    Program Testing . . . 3-3
    Summary . . . 3-3
Implementing ISYS: a Final Approach . . . 3-4
    Files Related to Program F77BUILD_SYM . . . 3-4
    Symbol Construction Rules . . . 3-6
    Operating Instructions for F77BUILD_SYM . . . 3-6
    Reducing QSYM.F77.IN . . . 3-7
    Error Messages . . . 3-10
    Updating your Operating System . . . 3-10
    ISYS and Sample Program LIST_DIRECTORY . . . 3-11
    ISYS and Subroutine CLI . . . 3-15
The ISYS Function and Multitasking . . . 3-21
IO_CHAN Function . . . 3-21
    Structure . . . 3-21
    Example . . . 3-22
    Reference . . . 3-22

Chapter 4 - Multitasking

What is a Task? 4-1
Single-task Programs 4-1
Single-tasking: a Nonsoftware Example 4-2
What is Multitasking? — a Nonsoftware Example 4-3
What is Multitasking? 4-5
Multitasking Program Organization 4-7
Task States, Transitions, and Subroutines 4-7
Task States 4-7
Task Transitions 4-11
Task Subroutines 4-11
Sample Program 4-14
Re-entrant Code 4-20
Multitasking Subroutines 4-22
Assembly Language Interface 4-24
Conversion of FORTRAN 5 Multitasking Programs 4-25
Multitasking via the ISYS Function? 4-27
Link Switches for F77 Multitasking 4-27
Task Fatal Errors 4-28
Initial Task 4-28
Documentation of Multitasking Calls 4-28
TQDQTSK 4-30
TQDRSCH 4-32
TQERSCH 4-33
TQIDKIL 4-34
TQIDPRI 4-35
TQIDRDY 4-36
TQIDSTAT 4-37
TQIDSUS 4-38
TQIQTSK 4-39
TQKILAD 4-40
TQKILL 4-41
TQMYTID 4-42
TQPRI 4-43
TQPRKIL 4-44
TQPROT 4-45
TQPRRDY 4-46
TQPRSUS 4-47
TQQTASK 4-48
TQREC 4-49
TQRECNW 4-50
TQSTASK 4-51
TQSUS 4-52
TQUNPROT 4-53
TQXMT 4-54
TQXMTW 4-55
Another Sample Multitasking Program 4-56
AOS F77 Multitask Stack Definition 4-65
Macro F77STACK 4-66
Macro MAINSTACK 4-66
Example Entries for F77STACK.SR 4-67
Operating Instructions for F77STACK 4-67
How Necessary is F77STACK? 4-67
An Example of Specific Stack Specifications 4-67

Chapter 5 - Debugging

Traditional Debugging Methods 5-1
The SWAT Debugger 5-2
Sample Program Modules SORT10.F77 and TEST_SORT10.F77 5-2
Sample Execution without the SWAT Debugger 5-5
SWAT Debugger Fundamentals 5-5
Sample Execution with the SWAT Debugger 5-6
Corrections to Sample Program Modules 5-12
The SWAT Debugger — a Summary 5-12
Avoid Errors BEFORE Coding 5-13
Data General Bugs? 5-13

Chapter 6 - Subprograms

F77 and Assembly Language Subprograms 6-1
Calling Conventions 6-1
Common Return Block 6-3
Coding Assembly Language Routines for Use with F77 with Macros 6-9
F77-to-Assembly Interface Examples 6-10
Macro F77_FMAC.SR 6-16
Compatibility Between Languages 6-17
Multidimension Array Storage 6-17
Case Sensitivity 6-20
Interlanguage Conflicts 6-20
A Sample Subprogram and Its Caller 6-20
High-Level Languages and F77 Subroutines 6-23
FORTRAN 5 and F77 6-23
DG/L and F77 Languages 6-25
PL/I and F77 6-28

Chapter 7 - Programming Hints

The F77 Error File 7-1
Improving Program Readability 7-1
Program Enhancements 7-1
Compiler Switches and Program Performance 7-2
Enhancing Computational Speed 7-3
Enhancing I/O Speed 7-3
F77 Output and Printing Special Forms 7-5
Background for Two Examples 7-6
Example 1 — Printing Labels 7-6
Example 2 — Printing Index Cards 7-9
Reducing Memory and Disk Usage of Program Files 7-12
Link — A Closer Look 7-12
The /KTOP=n Link Switch 7-14
An Example of Reducing a .PR File 7-14
Cautions about Specifying /KTOP=n 7-16
Other Ways 7-18

Chapter 8 - Overlays

Introduction 8-1
Example 1 — A Program Using Overlays 8-2
Example 2 — A Program Implementing Overlays 8-4

Illustrations

Figure

1-1 Selected Data General Software ... 1-2
1-2 The Compilation, Linking, and Execution of a Typical F77 Program ... 1-5

2-1 Program EXAMPLE_RANDOM.F77 ... 2-12
2-2 The Output from Program EXAMPLE_RANDOM ... 2-12
2-3 A Correspondence Between Selected Real Numbers and Integers ... 2-14
2-4 Program ROLL_DICE.F77 ... 2-16
2-5 Typical Output from Program ROLL_DICE ... 2-17

3-1 The Compilation, Linking, and Execution of a Typical F77 Program ... 3-5
3-2 Program NEW_TEST_SACL ... 3-9
3-3 Program LIST_DIRECTORY ... 3-12
3-4 Subroutine Subprogram ADD_NULL ... 3-14
3-5 Subroutine Subprogram CHECK ... 3-14
3-6 @CONSOLE Dialog During Execution of LIST_DIRECTORY ... 3-15
3-7 Subroutine Subprogram CLI ... 3-17
3-8 Program TEST_CLI ... 3-18
3-9 @CONSOLE Dialog During Execution of TEST_CLI ... 3-19
3-10 Program TEST1_CLI ... 3-20
3-11 @CONSOLE Dialog During Execution of TEST1_CLI ... 3-21

4-1 A One-Lane Tunnel with One Approach Lane (Single-Tasking) ... 4-2
4-2 A Two-Lane Tunnel with Four Approach Lanes ... 4-4
4-3 A Multitasking Program File ... 4-6
4-4 The Organization and Execution of a Single-Task Program ... 4-8
4-5 The Organization and Execution of a Multitask Program ... 4-9
4-6 Task States ... 4-10
4-7 Task States and Transitions ... 4-13
4-8 A Listing of Program MAIN5.F77 ... 4-15
4-9 A Listing of Subroutine TASK1.F77 ... 4-16
4-10 A Listing of Subroutine TASK2.F77 ... 4-18
4-11 Task Control Blocks and the Use of Re-entrant Code ... 4-21
4-12 A Listing of Subroutine TASK_11.F77 ... 4-57
4-13 A Listing of Subroutine TASK11.F77 ... 4-59
4-14 A Listing of Subroutine TASK12.F77 ... 4-60
4-15 A Listing of Subroutine TASK13.F77 ... 4-61
4-16 A Listing of Subroutine TASK14.F77 ... 4-62
4-17 A Listing of Subroutine TASK15.F77 ... 4-63

6-1 The Stack after Execution of a SAVE Instruction ... 6-5
6-2 A Listing of TEST_TYP_SUB.F77 and Its Generated Code ... 6-6
6-3 A Listing of TYP_SUB.F77 and Its Generated Code ... 6-8
6-4 Main Program TEST_RUNTM.F77 ... 6-11
6-5 Subroutine RUNTM.SR, Version 1 ... 6-12
6-7 An Example of Storage of Multidimension Arrays by F77 and Other Languages ... 6-18
6-8 Subroutine Subprogram GENERAL.F77 ... 6-21
6-9 Main Program TEST_GENERAL.F77 ... 6-22
6-10 Program TEST_GENERAL.FR ... 6-24
6-11 Program TEST_GENERAL.DG ... 6-26

7-1 File MEMBERS.DATA 7-6
7-2 Program PRINT_LABELS 7-7
7-3 A Typical Index Card 7-9
7-4 Program PRINT_CARDS 7-10
7-5 A Memory Model for an F77 Program File 7-13
7-6 A Listing of Program MEMPAGE.LS 7-15
7-7 A Portion of MEMPAGE.MAP 7-17

8-1 The Desired Organization of GRADUATION_CHECK.PR and Its Overlay File 8-3
8-2 The Desired Organization of SAMPLE_OVERLAY.PR and Its Overlay File 8-5
8-3 Main Program SAMPLE_OVERLAY.F77 8-6
8-4 Subprogram SUB_00_00.F77 8-7
8-5 Subprogram SUB_00_01.F77 8-7
8-6 Subprogram SUB_01_00.F77 8-8
8-7 Subprogram SUB_01_01.F77 8-8
8-8 Subprogram SUB_01_02.F77 8-9
8-9 A Portion of SAMPLE_OVERLAY.MAP 8-10

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

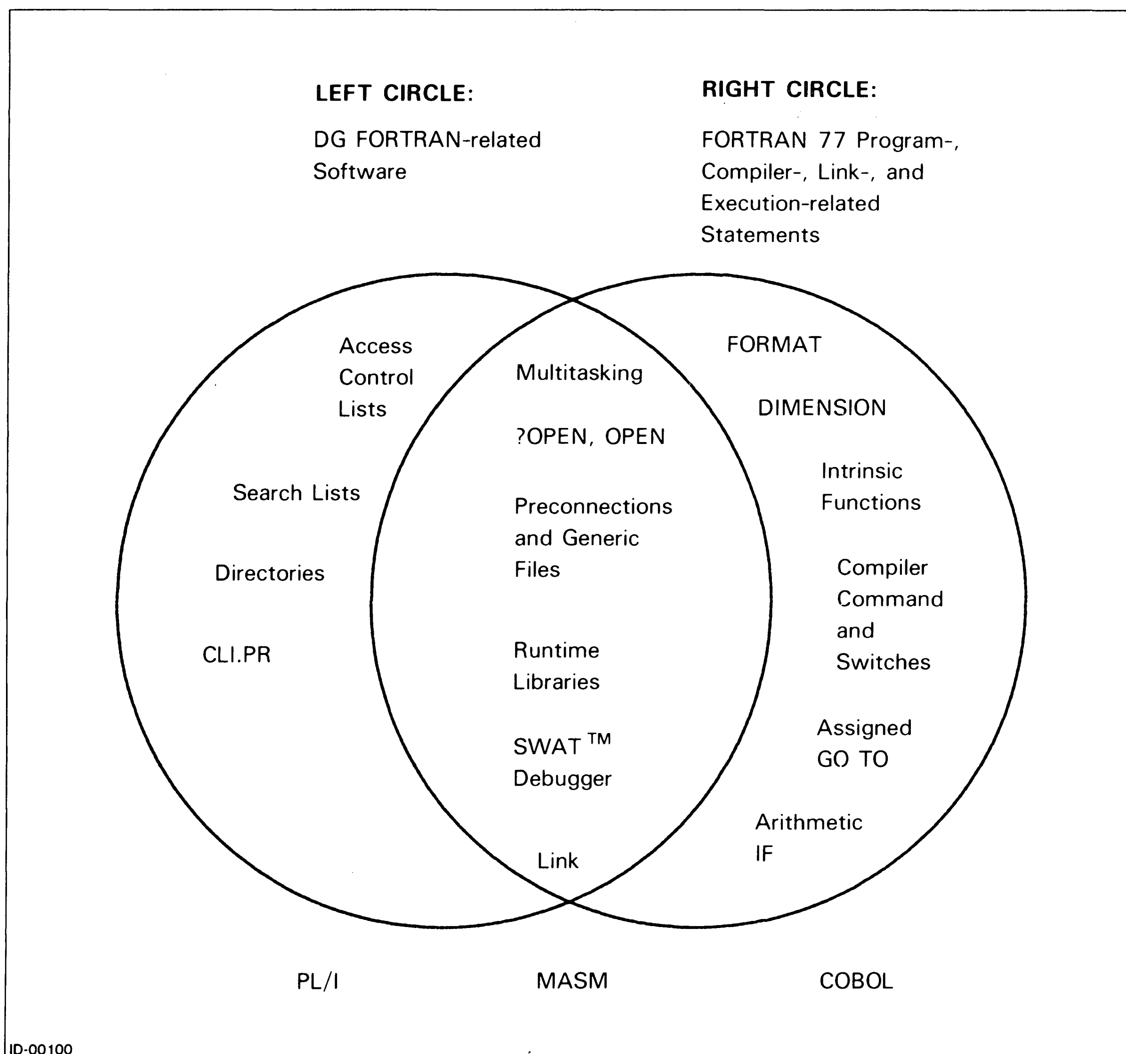


Figure 1-1. Selected Data General Software

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 DG/L™ program that CALLs a FORTRAN 77 subroutine to perform some number crunching.

The Significance of AOS

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

- 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 (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 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 always executes. Furthermore, consider the F77 STOP statement. When its resulting instructions in a program file execute, they tell AOS 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 is normally the Command Line Interpreter (CLI), which communicates directly with AOS.

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 programs (F77.PR, F77PASS2.PR, F77PASS3.PR) are large and complicated programs that do 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, these other .OB files, and library 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*2 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 programs process statement 4 by, among other things, creating a note in SAMPLE.OB to LINK.PR. (Technically, this "note" is an External Reference -- an .EXTN statement). This notification 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 to a 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 an AND instruction. At runtime this instruction executes to find the logical AND of the 2-byte integer 8 and of the 2-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 refers to.

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

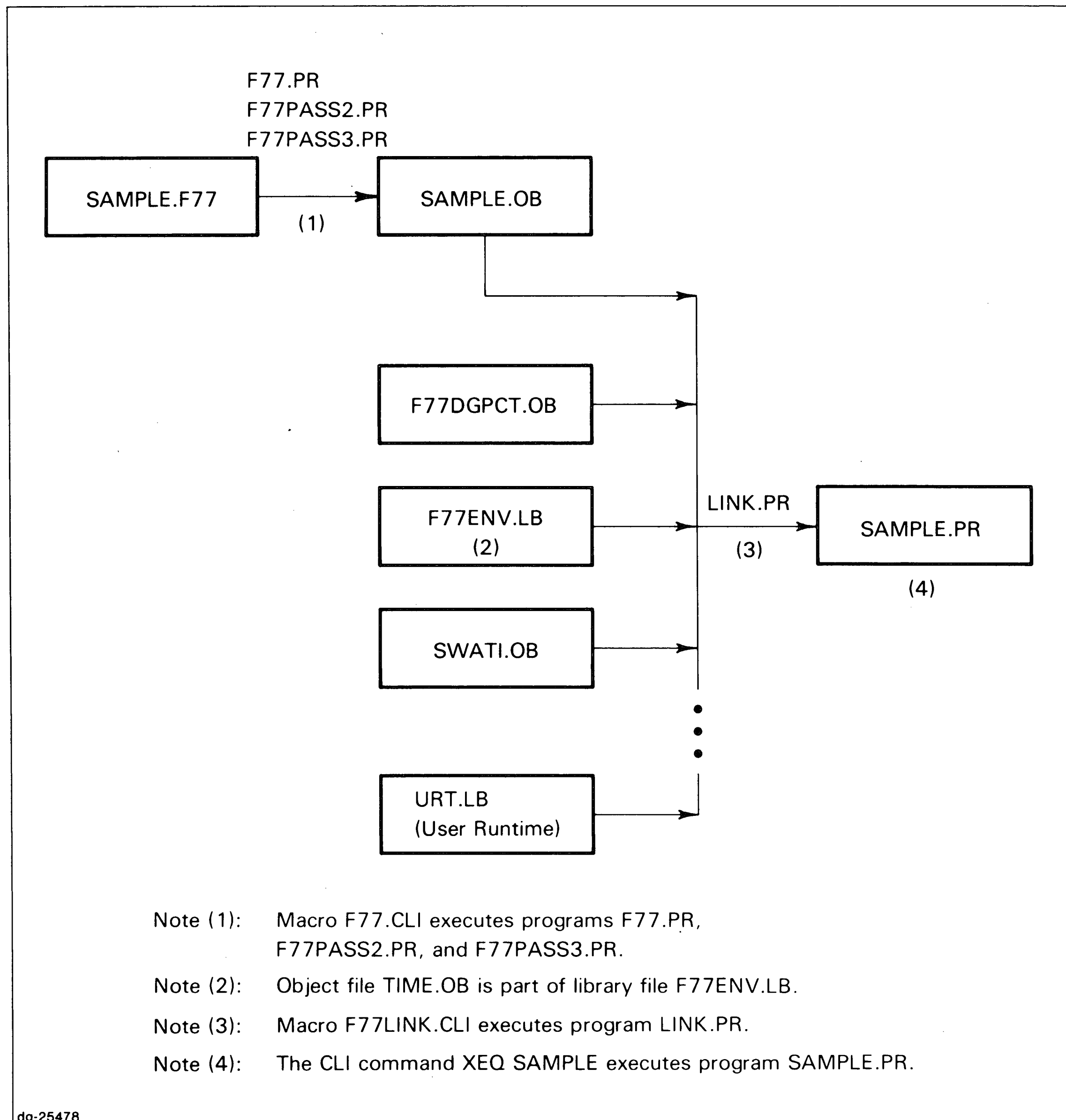


Figure 1-2. The Compilation, Linking, and Execution of a Typical F77 Program

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. And, not all of the F77 runtime library files appear in Figure 1-2. F77IO.LB is an example. You can print F77LINK.CLI to see the names of all Data-General-created object and 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, F77PASS2.PR, F77PASS3.PR, and LINK.PR. 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 a 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*2 array into whose first three elements DATE will place the current date from AOS:

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 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       ...
        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*2 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 Common Language Runtime Environment (CLRE). See the current F77 Release Notice for instructions to create this file.

sev is an optional INTEGER*2 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.

Example Program

Program TEST_ERRCODE lets you 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*2 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)
-----
RUNTIME ERROR 26000 at 074124
  in ROUTINE USER.ERR
called from .MAIN+174
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)
-----
RUNTIME ERROR 23420 at 074124
  in ROUTINE USER.ERR
called from .MAIN+174
UNKNOWN MESSAGE CODE 023420

-----

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)
-----
RUNTIME ERROR 44 at 074124
  in ROUTINE USER.ERR
called from .MAIN+174
DEVICE ALREADY IN SYSTEM

-----

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

END OF TESTING OF SUBROUTINE ERRCODE
STOP
```

ERRCODE (continued)

Please note the following about the execution of TEST_ERRCODE:

- Your "RUNTIME ERROR" and "called from" memory locations probably will differ from those shown (074124 and .MAIN+174, respectively).
- 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 can respond to more than just nonzero values in ERR.F77.IN. Here, 36 (= 44K) is a valid AOS system error code. ERMES must contain AOS error codes as well as those from F77.

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 Common Language Runtime Environment (CLRE). See the current Release Notice for instructions to create this file.

sev is an optional INTEGER*2 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 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)
-----
RUNTIME ERROR 26536 at 074150
  in ROUTINE USER.ERR
called from .MAIN+204
User defined ERROR text
EXECUTION continues
SAMPLE 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)
-----
RUNTIME ERROR 26536 at 074150
  in ROUTINE USER.ERR
called from .MAIN+204
User defined ERROR text
EXECUTION continues
SOME MORE 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 "RUNTIME ERROR" and "called from" memory locations probably will differ from those shown (074150 and .MAIN+204, respectively).
- 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 error code 26536 (decimal 11614) because this is the error code for user-defined error text.

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 with a console message whose minimal contents are "STOP".

Format

CALL EXIT

Arguments

none

Example

```
C       SAMPLE AOS 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 successive 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
```

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 ***
```

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 can land in 6x6=36 different ways on each roll. There are only two ways a total of three dots can 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.

RANDOM (continued)

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.

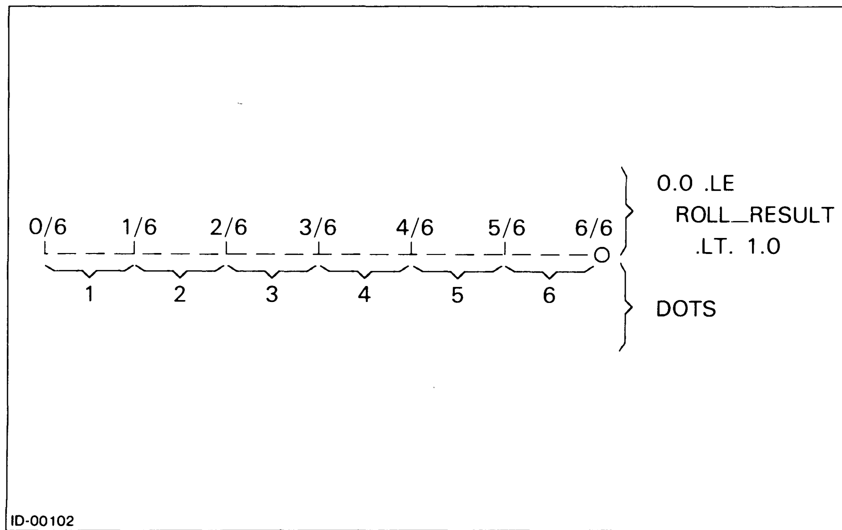


Figure 2-3. A Correspondence Between Selected Real Numbers and Integers

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

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

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

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 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
```

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

```
      WRITE (6, 80) MAXIMUM_CHI_SQUARE, CHI_SQUARE
80    FORMAT (1HO, '<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
```

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
```

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*2 array into whose first three elements TIME will place the absolute time (based on a 24-hour clock) from AOS:

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 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       ...
        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. It also explains the external function subprogram IO_CHAN that returns an AOS 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 Programmer's Manual* and from your program's requirements. At runtime, F77 attempts an AOS 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*2 expression that contains the value of an AOS system call code. This code comes from a statement in SYSID.SR that assigns the value to a system call symbol. SYSID.SR is normally in :UTIL.

AC0
AC1
AC2 are INTEGER*2 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 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 2.00 of AOS 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 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 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.SR and SYSID.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 ISYS
INTEGER 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 can 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.SR and PARU.SR).
- BYTEADDR and WORDADDR intrinsic functions.
- ISYS external function.

to create a FORTRAN 77 program that hooks into AOS 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 and SYSID 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 and SYSID 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.SR. Program F77BUILD_SYM by default transforms the second statement from its equivalent in listing file PARU.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 and SYSID .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 and SYSID .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.

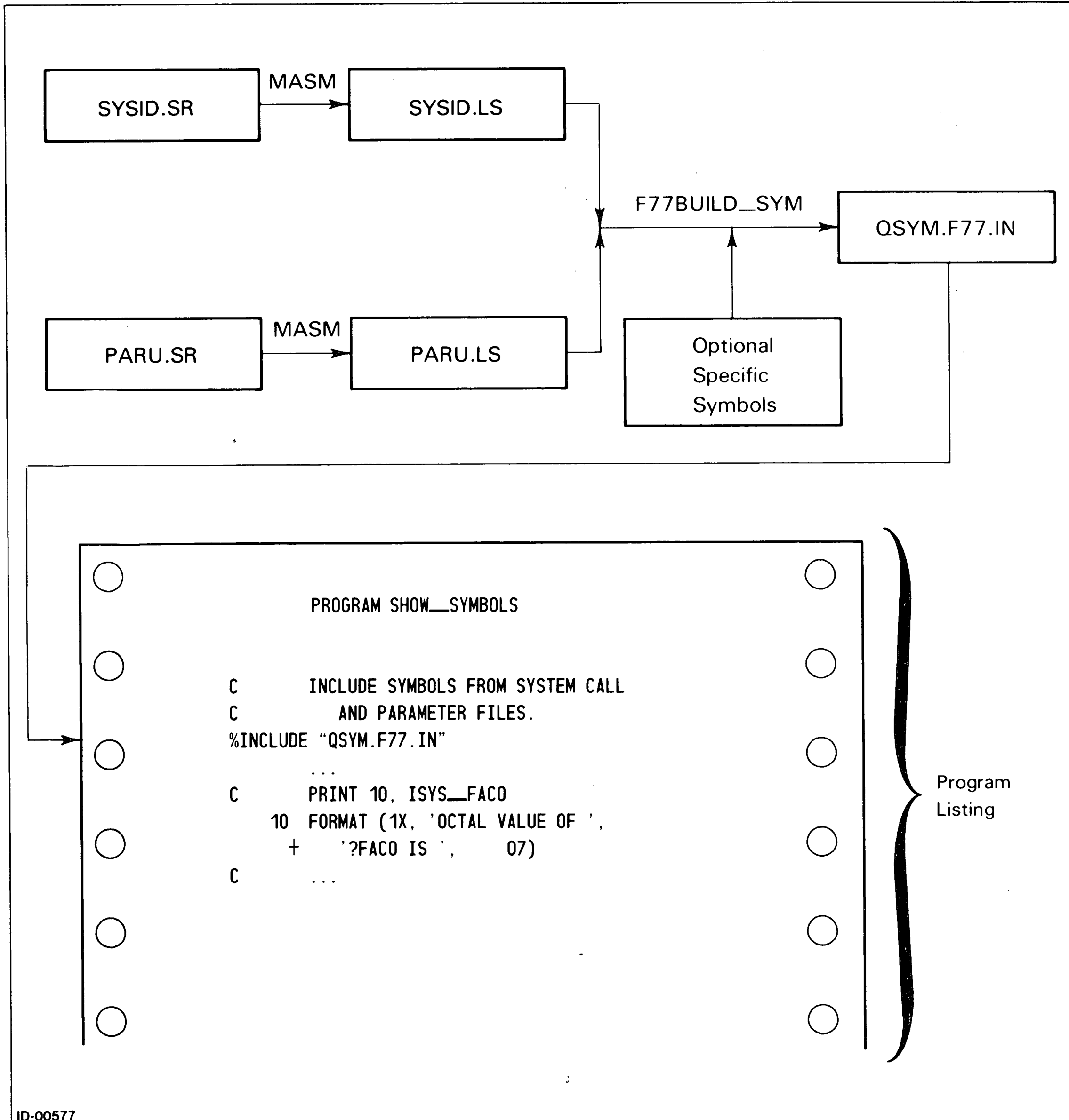


Figure 3-1. The Construction and Use of Parameter File QSYM.F77.IN

Symbol Construction Rules

F77BUILD_SYM follows these rules in sequence as it converts each PARU and SYSID .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.LS — sequentially.

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

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

statement. It handles symbols with values of 100000K (-32768) differently. However, you don't have to make any changes to your programs or even know these symbols' names.

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

NOTE: MASM may report errors such as

C .MACRO ?SCL1

on @OUTPUT. They will not appear in either .LS file, and you can ignore such error messages.

You should now place the statement

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

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

Reducing QSYM.F77.IN

Although comprehensive and usable by any F77 program that needs to interface with the operating system, file QSYM.F77.IN 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 1800+ 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 and SYSID 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.LS and PARU.LS remain from a prior assembly of SYSID.SR and of PARU.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 19-Oct-82 at 14:38:54 by AOS F77 Rev 2.10
Options: F77/INTEGER=2/LOGICAL=2/L=NEW_TEST_SACL.LS

   1  C       AOS PROGRAM NEW_TEST_SACL
   2          INTEGER ISYS, VALUE_OWARE, VALUE_RE
   3          CHARACTER*20 AC1 ! FOR THE NEW ACL
   4          INTEGER  BPTR_ACO, BPTR_AC1 ! BYTE POINTERS TO ACO, AC1
   5  %INCLUDE "SACL_SYMBOLS.F77.IN"
   6  ****  F77 INCLUDE file for system parameters ****
   7
   8  ****   Parameters for SYSID  ****
   9
  10          INTEGER*2  ISYS_SACL
  11          PARAMETER (ISYS_SACL = 76)      ! ?.SACL = 114K
  12
  13  ****    Parameters for PARU   ****
  14
  15
  16          INTEGER*2  ISYS_FACO
  17          PARAMETER (ISYS_FACO = 16)      ! ?FACO = 20K
  18
  19          INTEGER*2  ISYS_FACW
  20          PARAMETER (ISYS_FACW = 8)       ! ?FACW = 10K
  21
  22          INTEGER*2  ISYS_FACA
  23          PARAMETER (ISYS_FACA = 4)       ! ?FACA = 4K
  24
  25          INTEGER*2  ISYS_FACR
  26          PARAMETER (ISYS_FACR = 2)       ! ?FACR = 2K
  27
  28          INTEGER*2  ISYS_FACE
  29          PARAMETER (ISYS_FACE = 1)       ! ?FACE = 1K
  30
  31
  32  ****  END of F77 INCLUDE file for system parameters ****
  33  C       CONSTRUCT THE VALUE OF ?FACO+?FACW+?FACA+?FACR+?FACE .
  34          VALUE_OWARE = ISYS_FACO + ISYS_FACW + ISYS_FACA +
  35       1                 ISYS_FACR + ISYS_FACE
  36  C       CONSTRUCT THE VALUE OF ?FACR+?FACE .
  37          VALUE_RE    = ISYS_FACR + ISYS_FACE
  38  C       CONSTRUCT THE NEW ACL IN CHARACTER VARIABLE AC1.  NOTE THE
  39  C          USE OF THE CHAR INTRINSIC FUNCTION TO CONVERT AN INTEGER
  40  C          NUMBER TO ITS ASCII CHARACTER EQUIVALENT.  FOR EXAMPLE,
  41  C          VALUE_RE IS CURRENTLY (AOS/VS REVISION 1.50) 3 AND
  42  C          CHAR(VALUE_RE) IS '<3>'.
  43          AC1 = 'TOM<0>' // CHAR(VALUE_OWARE) // 'JERRY<0>' //
  44       1                    CHAR(VALUE_RE)    // '<0>'
  45          BPTR_ACO = BYTEADDR("NEW_STUDENTS<0>")
  46          BPTR_AC1 = BYTEADDR(AC1)
  47          IER = ISYS (ISYS_SACL, BPTR_ACO, BPTR_AC1, IAC2) ! DO IT!
  48          PRINT *, 'RESULT CODE FROM ISYS TO ?SACL IS ', IER
  49          END
```

Figure 3-2. Program NEW_TEST_SACL

Error Messages

The following error messages from F77BUILD_SYM could 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 and PARU .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.

Licensed Material-Property of Data General Corporation

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.

We have executed program F77BUILD_SYM to create a restricted symbol file for inclusion by each of program units LIST_DIRECTORY.F77 and CHECK.F77. The names and contents of the respective files given to F77BUILD_SYM are

LIST_DIRECTORY_SYMBOLS	CHECK_SYMBOLS
?.OPEN	RECF
?GNFN	RFEC
EREOF	RFER
?.CLOSE	?.RETURN

Although the output from F77BUILD_SYM is always file QSYM.F77.IN, we have renamed it to LIST_DIRECTORY_SYMBOLS.F77.IN, and then to CHECK_SYMBOLS.F77.IN. The respective statements

```
%INCLUDE 'LIST_DIRECTORY_SYMBOLS.F77.IN'   %INCLUDE 'CHECK_SYMBOLS.F77.IN'
```

do not appear in Figures 3-3 and 3-5. They are, of course, part of source program files LIST_DIRECTORY.F77 (at line 32) and CHECK.F77 (at line 11).

```
Source file: LIST_DIRECTORY.F77
Compiled on 21-Oct-82 at 14:41:44 by AOS F77 Rev 2.10
Options: F77/INTEGER=2/LOGICAL=2/L=LIST_DIRECTORY.LS

    1           PROGRAM LIST_DIRECTORY
    2
    3           INTEGER AC0,AC1,AC2          ! Accumulators
    4           INTEGER ISYS                 ! System interface function subprogram
    5           INTEGER RESULT_CODE           ! Result of calling ISYS
    6
    7           CHARACTER*132 FILENAME       ! Received by GNFN
    8           CHARACTER*132 DIRECTORY      ! Supplied to OPEN
    9           CHARACTER*132 TEMPLATE       ! Supplied to GNFN
   10
   11           INTEGER*2 OPEN_PACKET(0:11) / 12*0 /  ! Parameter packet for ?OPEN
   12              INTEGER*2 CHANNEL                  ! Offset ?ICH
   13              INTEGER*2 ISTI                     ! Offset ?ISTI
   14              INTEGER*2 ISTO                     ! Offset ?ISTO
   15              INTEGER*2 IBAD                     ! Offset ?IBAD
   16              INTEGER*2 IFNP                     ! Offset ?IFNP
   17              INTEGER*2 IMRS                     ! Offset ?IMRS
   18              INTEGER*2 IDEL                     ! Offset ?IDEL
   19
   20              EQUIVALENCE (OPEN_PACKET(0), CHANNEL)
   21              EQUIVALENCE (OPEN_PACKET(1), ISTI)
   22              EQUIVALENCE (OPEN_PACKET(2), ISTO)
   23              EQUIVALENCE (OPEN_PACKET(3), IBAD)
   24              EQUIVALENCE (OPEN_PACKET(9), IFNP)
   25              EQUIVALENCE (OPEN_PACKET(10),IMRS)
   26              EQUIVALENCE (OPEN_PACKET(11),IDEL)
   27
   28           INTEGER*2 GNFN_PACKET(0:2)             ! Parameter Packet for ?GNFN
   29
   30   C       %INCLUDE 'LIST_DIRECTORY_SYMBOLS.F77.IN'
   31   %LIST(OFF)
   56   %LIST(ON)
   57
   58     100   PRINT *,"Directory? "
   59           READ (*,10,END=1000) DIRECTORY   ! Accept a directory name.
   60      10   FORMAT(A)
   61   C       @INPUT end-of-file is CTRL-D.
   62
   63           CALL ADD_NULL(DIRECTORY)         ! Change the first (if any)
   64   C                                          space ('<040>') in the
   65   C                                          directory name to a null.
   66
   67   C       Prepare the parameter packet for ?OPEN.
   68           ISTI = 0                         ! Default ?OPEN options
   69           ISTO = 0                         ! Default file type
   70           IBAD = -1                        ! Default byte pointer to buffer
   71           IFNP = BYTEADDR(DIRECTORY)       ! Byte pointer to directory name
   72           IMRS = -1                        ! Default block size
   73           IDEL = -1                        ! Default delimiters
   74
   75           AC2 = WORDADDR(OPEN_PACKET)
   76
```

Figure 3-3. Program LIST_DIRECTORY (continues)

```
 77   C       Execute the ?OPEN system call to the accepted directory.
 78
 79           RESULT_CODE = ISYS(ISYS_OPEN, AC0, AC1, AC2)
 80
 81   C       If ?OPEN has executed successfully, then report nothing and
 82   C          continue.  Otherwise, report the error on @OUTPUT and STOP
 83   C          the program.
 84           CALL CHECK(RESULT_CODE,"On OPEN of directory " // DIRECTORY)
 85
 86           PRINT *,"Template?  "
 87           READ (*,20,END=1000) TEMPLATE        ! Typical response is  + .
 88       20  FORMAT(A)
 89
 90           CALL ADD_NULL(TEMPLATE)              ! Change the first (if any)
 91   C                                                space in TEMPLATE to
 92   C                                                a null.
 93
 94           GNFN_PACKET(0) = 0                   ! Offset ?NFKY
 95           GNFN_PACKET(1) = BYTEADDR(FILENAME)  ! Offset ?NFNM
 96           GNFN_PACKET(2) = BYTEADDR(TEMPLATE)  ! Offset ?NFTP
 97           AC1 = CHANNEL                        ! Channel number from ?OPEN
 98           AC2 = WORDADDR(GNFN_PACKET)
 99
100   C       Call ?GNFN to get the next filename from the current directory.
101      200  RESULT_CODE = ISYS(ISYS_GNFN, AC0, AC1, AC2)
102
103           IF ( RESULT_CODE .EQ. 0 ) THEN  ! Ignore the first (if any) null
104   C                                          in FILENAME and then print
105   C                                          the filename.
106                   NULL_POS = INDEX(FILENAME,"<NUL>")
107                           IF ( NULL_POS .EQ. 0 ) NULL_POS = LEN(FILENAME)-1
108                   PRINT *, FILENAME(1:NULL_POS-1)
109                   GOTO 200                ! Get the next filename.
110
111           ELSE IF ( AC0 .EQ. ISYS_EREOF ) THEN
112                   PRINT *
113                   PRINT *,"-- End of Directory --"
114                   PRINT *
115                   AC2 = WORDADDR(OPEN_PACKET)
116
117   C               Close the current directory and move to its superior.
118                   RESULT_CODE = ISYS(ISYS_CLOSE, AC0, AC1, AC2)
119                   CALL CHECK(RESULT_CODE, 'While closing the directory')
120
121                   GOTO 100                ! Get the next directory name
122
123           ELSE    ! A ?GNFN error, different from end-of-file, has occurred.
124                   CALL CHECK(AC0, 'During a ?GNFN Call')
125
126           ENDIF
127
128     1000  PRINT *
129           PRINT *,'<7>*** End of program LIST_DIRECTORY ***<NL>'
130           END
```

Figure 3-3. Program LIST_DIRECTORY (concluded)

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

    1          SUBROUTINE ADD_NULL(TEXT)
    2   C
    3   C      Change the first space in TEXT to a null.
    4   C
    5
    6          CHARACTER*(*) TEXT
    7          INTEGER SPACE_POS
    8
    9          SPACE_POS = INDEX(TEXT,'<040>')
   10          IF ( SPACE_POS .NE. 0 ) TEXT(SPACE_POS:SPACE_POS) = '<NUL>'
   11          RETURN
   12          END
```

Figure 3-4. Subroutine Subprogram ADD_NULL

```
Source file: CHECK.F77
Compiled on 21-Oct-82 at 14:43:24 by AOS F77 Rev 2.10
Options: F77/INTEGER=2/LOGICAL=2/L=CHECK.LS

    1          SUBROUTINE CHECK(ECODE,TEXT)
    2
    3          INTEGER ECODE         ! Error code returned from ISYS
    4          CHARACTER*(*) TEXT    ! Error text from main program to
    5   C                                 accompany ECODE
    6
    7          INTEGER AC2
    8
    9   C      %INCLUDE 'CHECK_SYMBOLS.F77.IN'
   10   %LIST(OFF)
   34   %LIST(ON)
   35
   36          IF ( ECODE .EQ. 0 ) RETURN  ! ISYS executed without an error.
   37
   38   C      ISYS executed with an error, so report it.
   39          AC2 = ISYS_RFCF + ISYS_RFEC + ISYS_RFER
   40          AC2 = AC2 + MIN(LEN(TEXT),255)
   41
   42   C      Execute ?RETURN and report the error from ISYS.
   43          IER = ISYS(ISYS_RETURN, ECODE, BYTEADDR(TEXT), AC2)
   44          STOP '- Impossible-to-occur error occurred during ?RETURN'
   45          END
```

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 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.

We have executed program F77BUILD_SYM to create a restricted symbol file for inclusion by program unit CLI.F77. The name and contents of the file given to F77BUILD_SYM are

CLI_SYMBOLS

?PFEX
?PROC

Although the output from F77BUILD_SYM is always file QSYM.F77.IN, we have renamed it to CLI_SYMBOLS.F77.IN. The statement

```
%INCLUDE 'CLI_SYMBOLS.F77.IN'
```

does not appear in Figure 3-7. It is, of course, part of source program file CLI.F77 (at line 32).

```
Source file: CLI.F77
Compiled on 21-Oct-82 at 14:44:20 by AOS F77 Rev 2.10
Options: F77/INTEGER=2/LOGICAL=2/L=CLI.LS

    1           SUBROUTINE CLI(TEXT, RESULT_CODE)
    2
    3     C     This subroutine receives a string of CLI commands from the main
    4     C        program.  The subroutine then creates a CLI son process and
    5     C        gives it the string of commands to execute.
    6
    7           INTEGER ADDRESS_OF_PROGRAM_NAME      ! Program name of the son
    8     C                                            process is CLI.PR.
    9           INTEGER ADDRESS_OF_STRING           ! The string is the string
   10     C                                            of CLI commands.
   11           INTEGER ADDRESS_OF_MESSAGE_HEADER    ! Packet for ?ISEND header
   12           INTEGER AC0, AC1, AC2              ! Accumulators
   13           INTEGER ISYS                       ! System interface function
   14           INTEGER RESULT_CODE                !    Number it returns to
   15     C                                            this subroutine and
   16     C                                            then to the main program.
   17
   18           INTEGER*2 PROC_PACKET(0:15) / 16*-1/ ! Packet for ?PROC call
   19              EQUIVALENCE ( ADDRESS_OF_PROGRAM_NAME, PROC_PACKET(1) )
   20              EQUIVALENCE ( ADDRESS_OF_MESSAGE_HEADER, PROC_PACKET(2) )
   21
   22           INTEGER*2 ISEND_HEADER(0:6) / 7*0 /  ! Packet for ?ISEND header
   23     C                                            for interprocess
   24     C                                            communication (IPC).
   25              EQUIVALENCE ( ADDRESS_OF_STRING, ISEND_HEADER(6) )
   26
   27           CHARACTER*(*) TEXT                 ! String of CLI commands
   28           CHARACTER*(256) TEMPORARY_TEXT
   29
   30     C     %INCLUDE 'CLI_SYMBOLS.F77.IN'
   31     %LIST(OFF)
   49     %LIST(ON)
   50
   51           TEMPORARY_TEXT = TEXT               ! Move the CLI commands to
   52     C                                            a fixed-length buffer.
   53
   54     C     Prepare ?ISEND header packet.
   55           ISEND_HEADER(5) = 128               ! Maximum length of the IPC
   56     C                                            message in words
   57           ADDRESS_OF_STRING = WORDADDR(TEMPORARY_TEXT)
   58
   59           PROC_PACKET(0) = ISYS_PFEX   ! Set ?PFEX bit so that CLI.PR will
   60     C                                    execute with its father blocked.
   61           ADDRESS_OF_PROGRAM_NAME = BYTEADDR(':CLI.PR<0>')
   62           ADDRESS_OF_MESSAGE_HEADER = WORDADDR(ISEND_HEADER)
   63           AC2 = WORDADDR(PROC_PACKET)
   64
   65     C     Do it!
   66           RESULT_CODE = ISYS(ISYS_PROC, AC0, AC1, AC2)
   67     C     The main program receives the value of RESULT_CODE.
   68
   69           RETURN
   70           END
```

Figure 3-7. Subroutine Subprogram CLI

```
Source file: TEST_CLI.F77
Compiled on 21-Oct-82 at 13:26:30 by AOS F77 Rev 2.10
Options: F77/INTEGER=2/LOGICAL=2/L=TEST_CLI.LS

    1           PROGRAM TEST_CLI          ! to test subroutine CLI
    2           CHARACTER*80 CLI_STRING  ! string of CLI commands
    3           INTEGER IER              ! error variable returned from
    4   C                                   subroutine CLI and from its
    5   C                                   reference to function ISYS
    6
    7           WRITE (6, 20)
    8      20   FORMAT (1HO, 'GIVE ME A CLI COMMAND STRING: ', $)
    9           READ (5, 30, END=60) CLI_STRING
   10      30   FORMAT (A)
   11   C       @INPUT end-of-file is CTRL-D.
   12
   13           WRITE (6, 40)
   14      40   FORMAT (1H , 'HERE WE GO ...', /, /)
   15           CALL CLI (CLI_STRING, IER)
   16           WRITE (6, 50)
   17      50   FORMAT (1H , 'JUST RETURNED FROM SUBROUTINE CLI')
   18           IF ( IER .NE. 0 ) THEN
   19               PRINT *
   20               PRINT *, 'ERROR ', IER, ' OCCURRED DURING ',
   21        1                 'REFERENCE TO ISYS'
   22               PRINT *, '   WHEN SUBROUTINE CLI EXECUTED.'
   23           ENDIF
   24
   25      60   WRITE (6, 70)
   26      70   FORMAT (1HO, '*** END OF PROGRAM ***', /)
   27
   28           STOP
   29           END
```

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 F77 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 7not7 execute its statements in lines 19-22.

```
) XEQ TEST_CLI ↲
GIVE ME A CLI COMMAND STRING:  TIME; DATE; DIRECTORY; WHO ↲
HERE WE GO ...
15:28:11
21-OCT-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-25471

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 can 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 21-Oct-82 at 13:37:12 by AOS F77 Rev 2.10
Options: F77/INTEGER=2/LOGICAL=2/L=TEST1_CLI.LS

    1          PROGRAM TEST1_CLI        ! to test subroutine CLI
    2          CHARACTER*80 CLI_STRING  ! string of CLI commands
    3          INTEGER IER             ! error variable returned from
    4   C                                   subroutine CLI and from its
    5   C                                   reference to function ISYS
    6
    7
    8
    9          CLI_STRING = 'TIME; DATE; WHO; RUNTIME'
   10
   11
   12
   13          WRITE (6, 40)
   14      40  FORMAT (1H , 'HERE WE GO ...', /, /)
   15          CALL CLI (CLI_STRING, IER)
   16          WRITE (6, 50)
   17      50  FORMAT (1H , 'JUST RETURNED FROM SUBROUTINE CLI')
   18          IF ( IER .NE. 0 ) THEN
   19               PRINT *
   20               PRINT *, 'ERROR ', IER, ' OCCURRED DURING ',
   21        1                'REFERENCE TO ISYS'
   22               PRINT *, '   WHEN SUBROUTINE CLI EXECUTED.'
   23          ENDIF
   24
   25      60  WRITE (6, 70)
   26      70  FORMAT (1H0, '*** END OF PROGRAM ***', /)
   27
   28          STOP
   29          END
```

Figure 3-10. Program 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*2 statement.
unit	is an INTEGER*2 expression that specifies an F77 I/O unit number.

```
) XEQ TEST1_CLI ↲
HERE WE GO ...
15:41:22
21-OCT-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-25472
```

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

Example

```
C       AOS PROGRAM DEMO_IO_CHAN
C       ...
        INTEGER*2 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 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
- Multitasking Stack Definition

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 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 can execute more than once during the task's existence. For example, such an instruction can 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 can 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 has to wait only for its predecessor's completion before CPU execution. (An exception to this rule occurs when there is overlap in floating-point instruction executions.)

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.

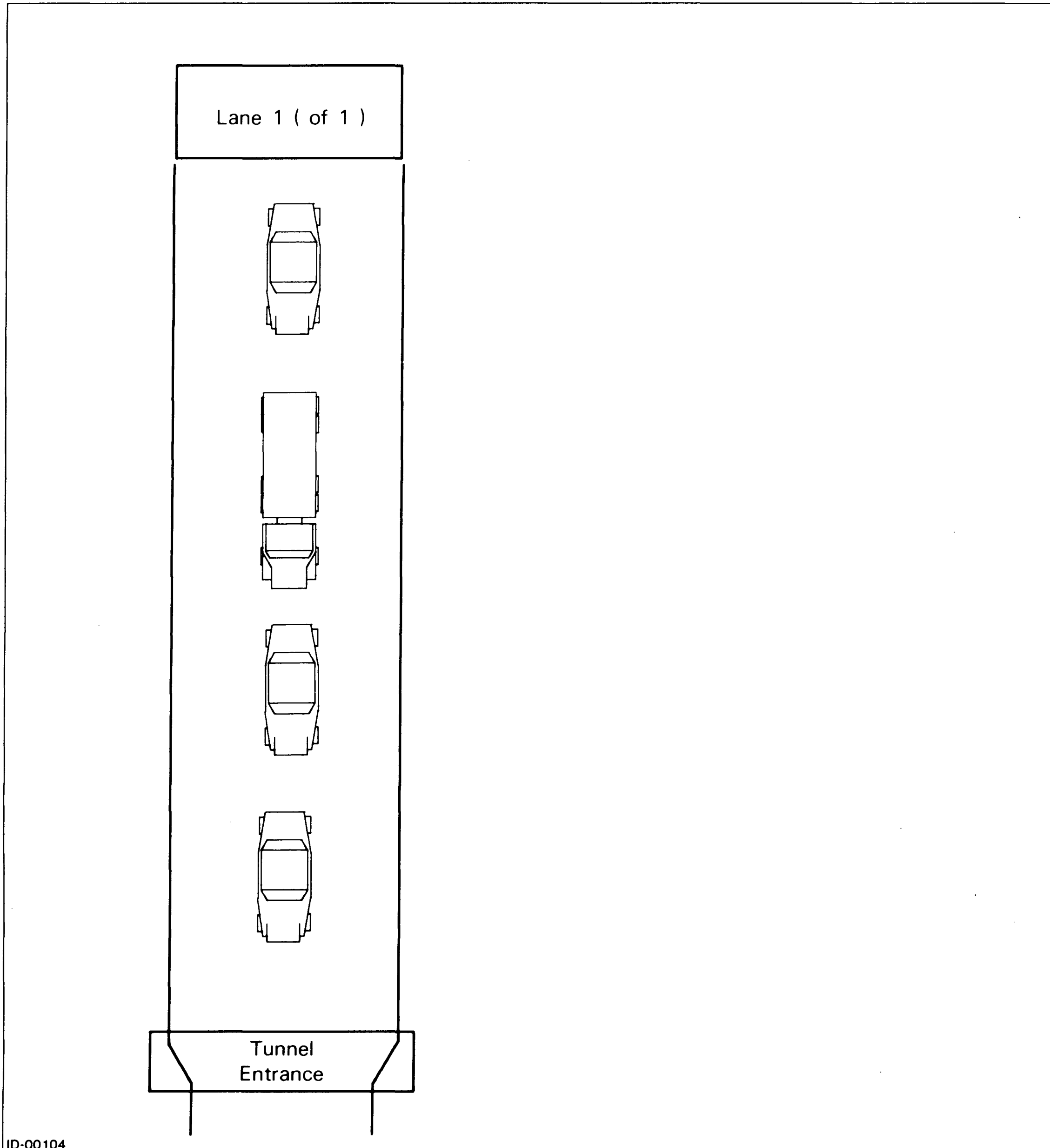


Figure 4-1. A One-Lane Tunnel with One Approach Lane (Single-Tasking)

Licensed Material-Property of Data General Corporation

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 cars behind the truck have a longer wait than cars behind other cars.

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 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* can 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.

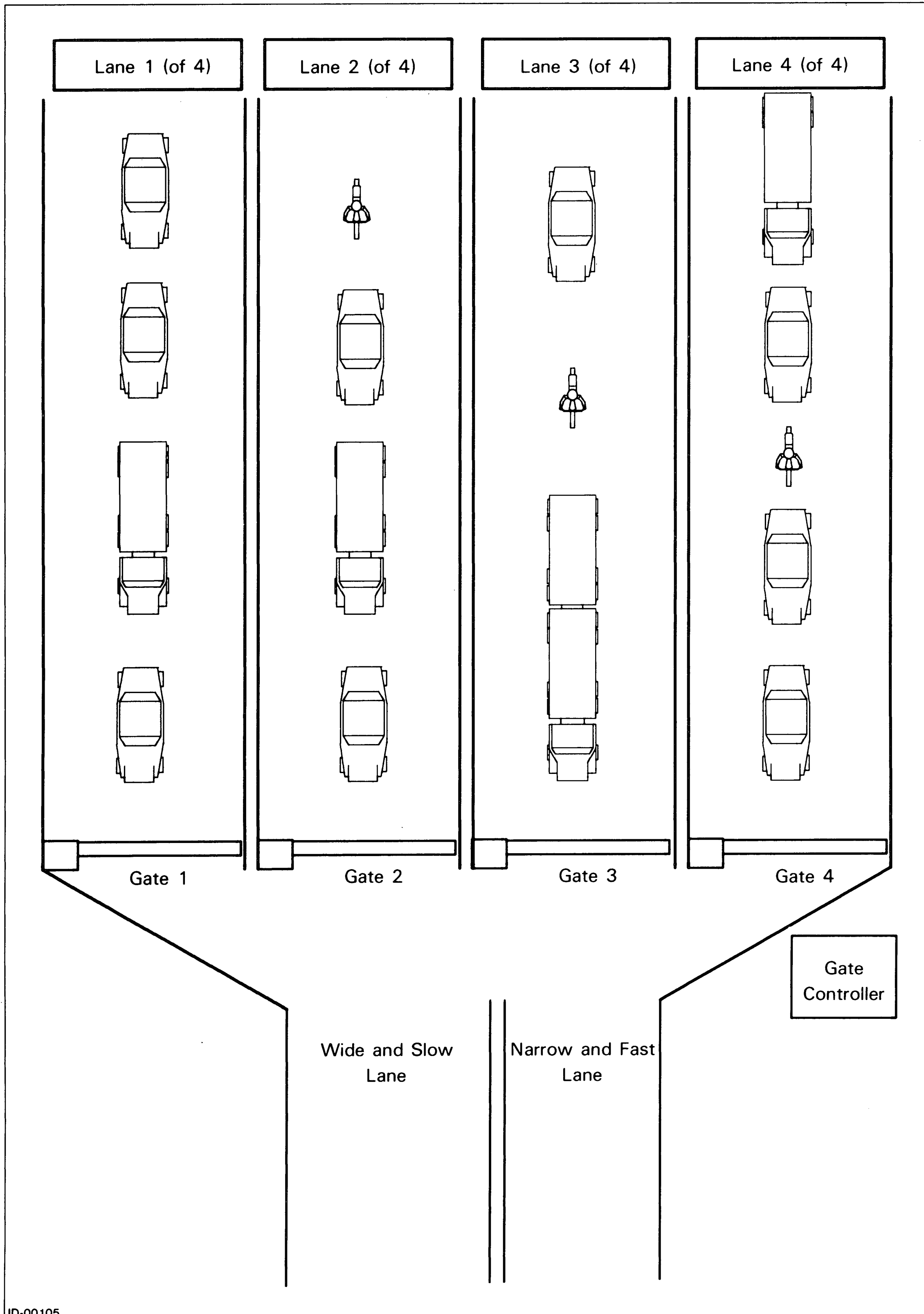


Figure 4-2. A Two-Lane Tunnel with Four Approach Lanes

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 take care of such bookkeeping activities. As many as 32 tasks can 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, allocates CPU control to the highest priority task that is ready either to perform or to continue to perform its function.

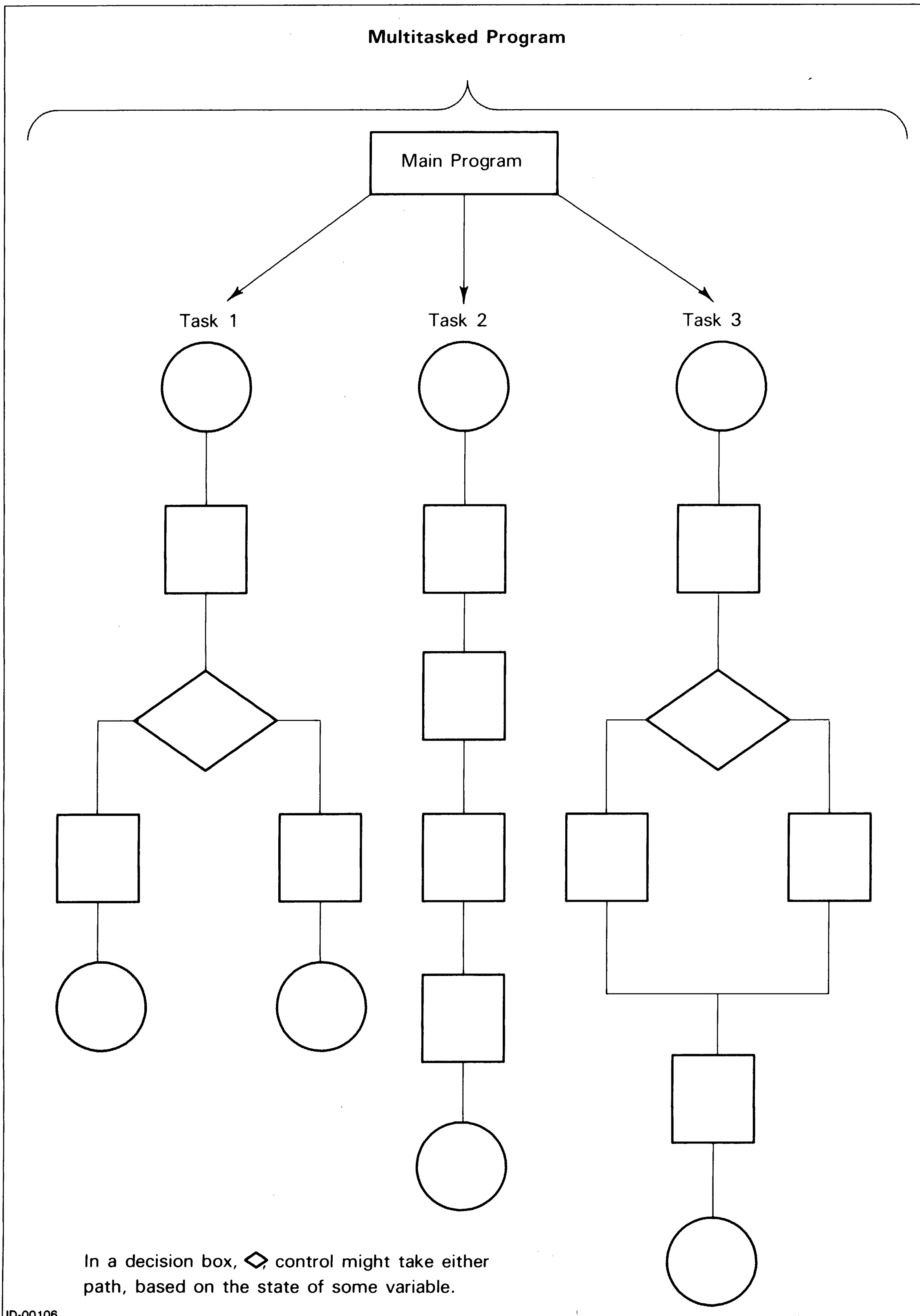


Figure 4-3. A Multitasking Program File

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 may not, 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 or not MAIN5 has executed its CALL EXIT statement. (Execution of CALL EXIT and END statements, along with the RETURN statement, results 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.

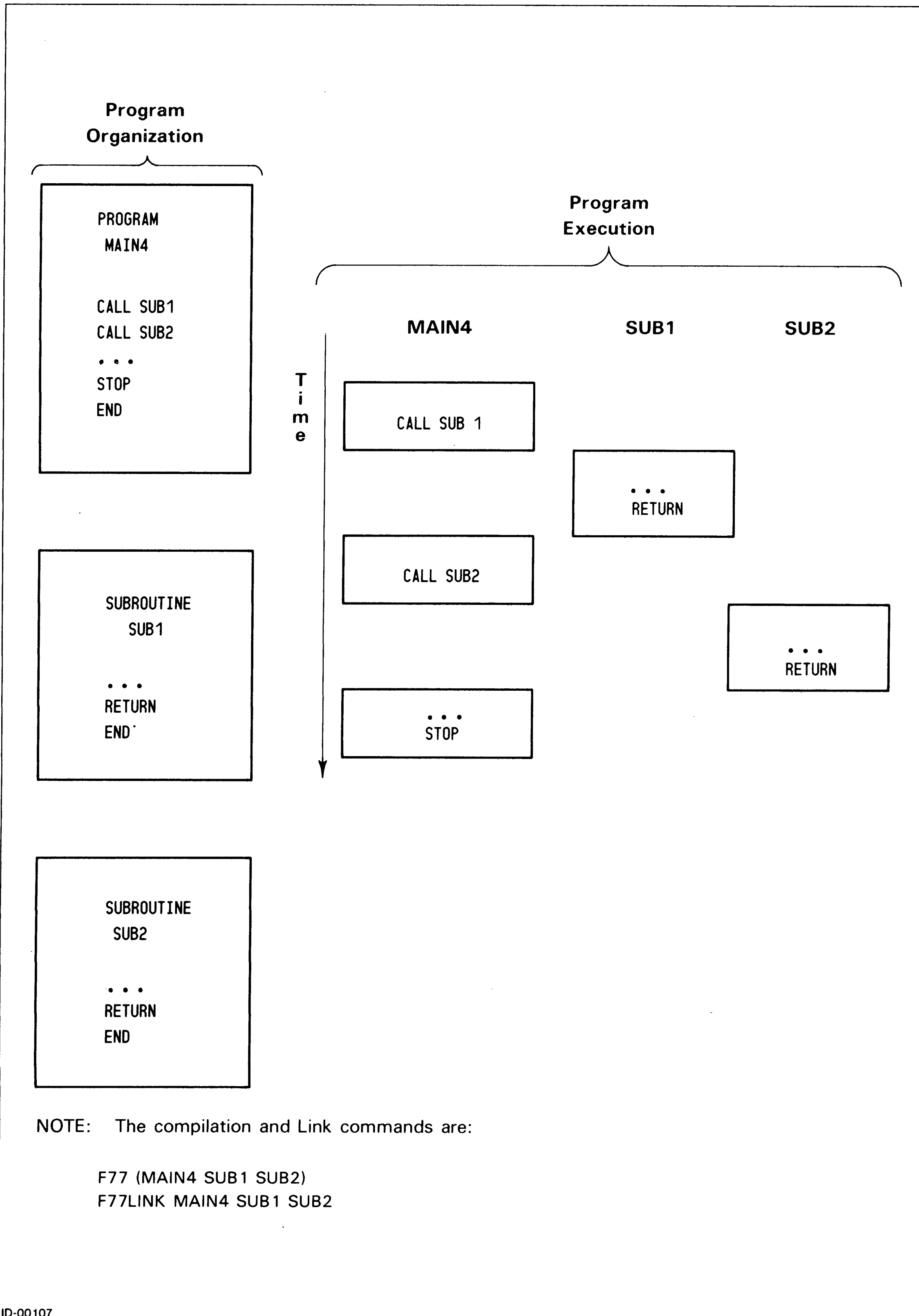


Figure 4-4. The Organization and Execution of a Single-Task Program

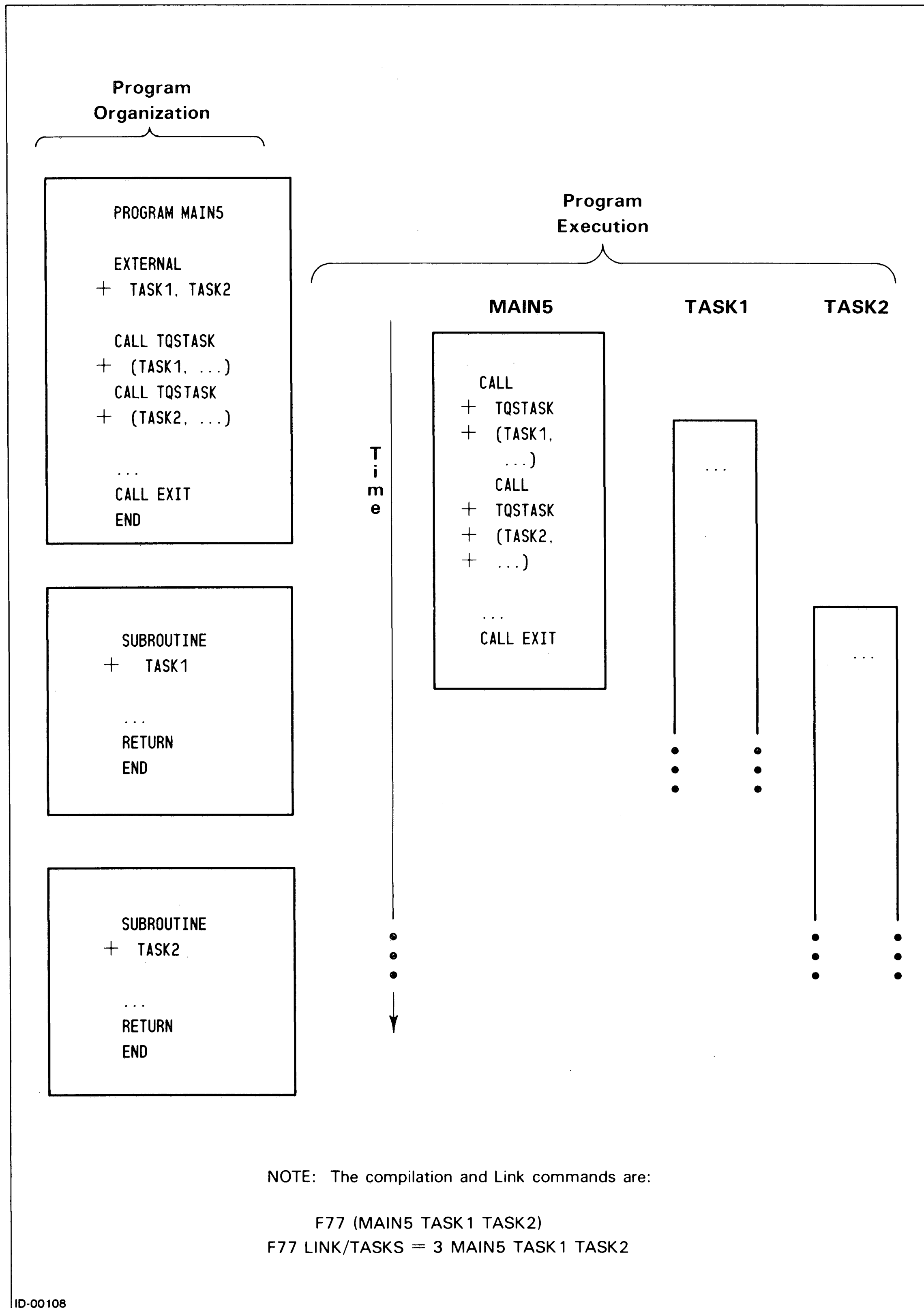


Figure 4-5. The Organization and Execution of a Multitask Program

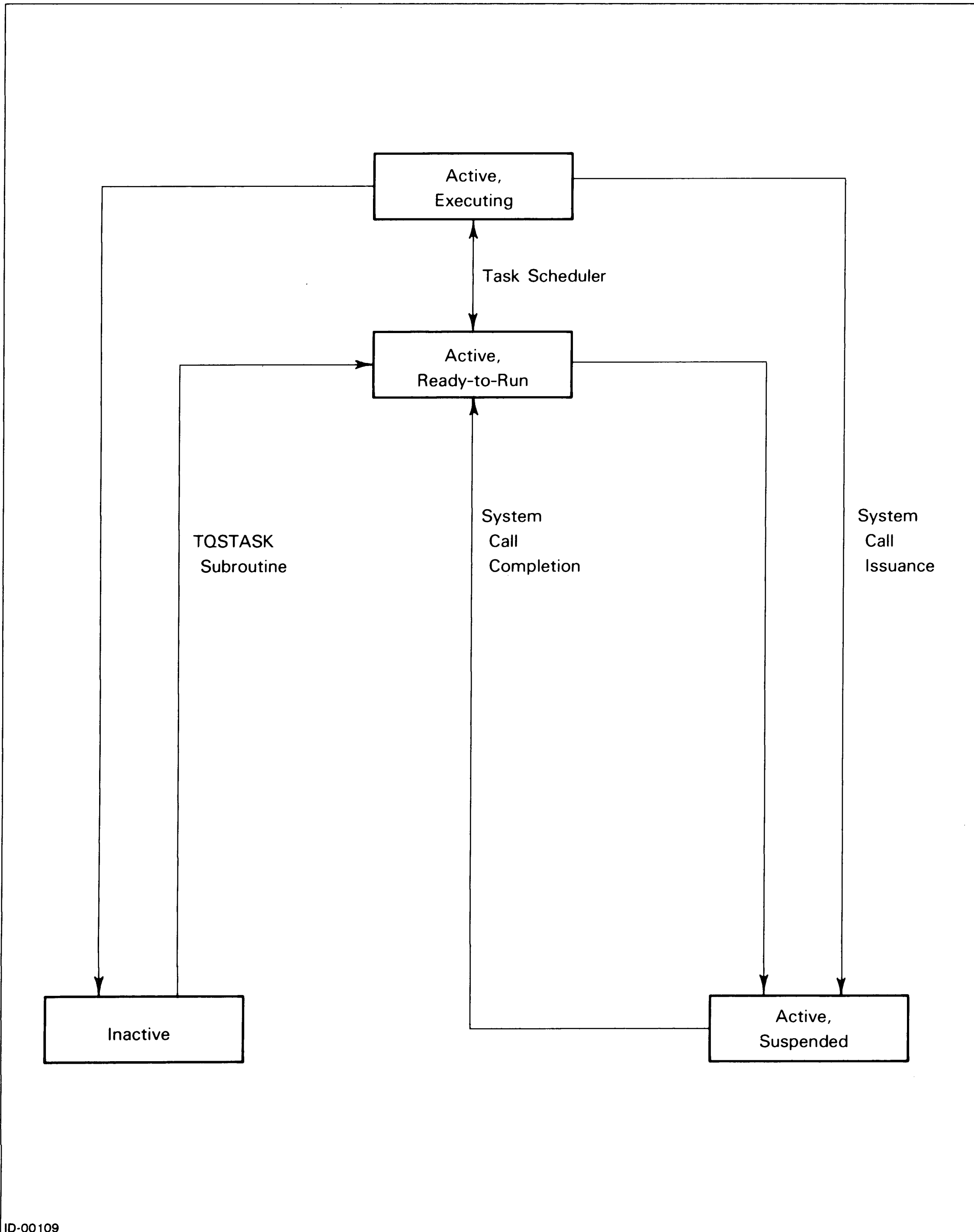


Figure 4-6. Task States

The runtime states a task can have (in order of 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 can 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 can 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.

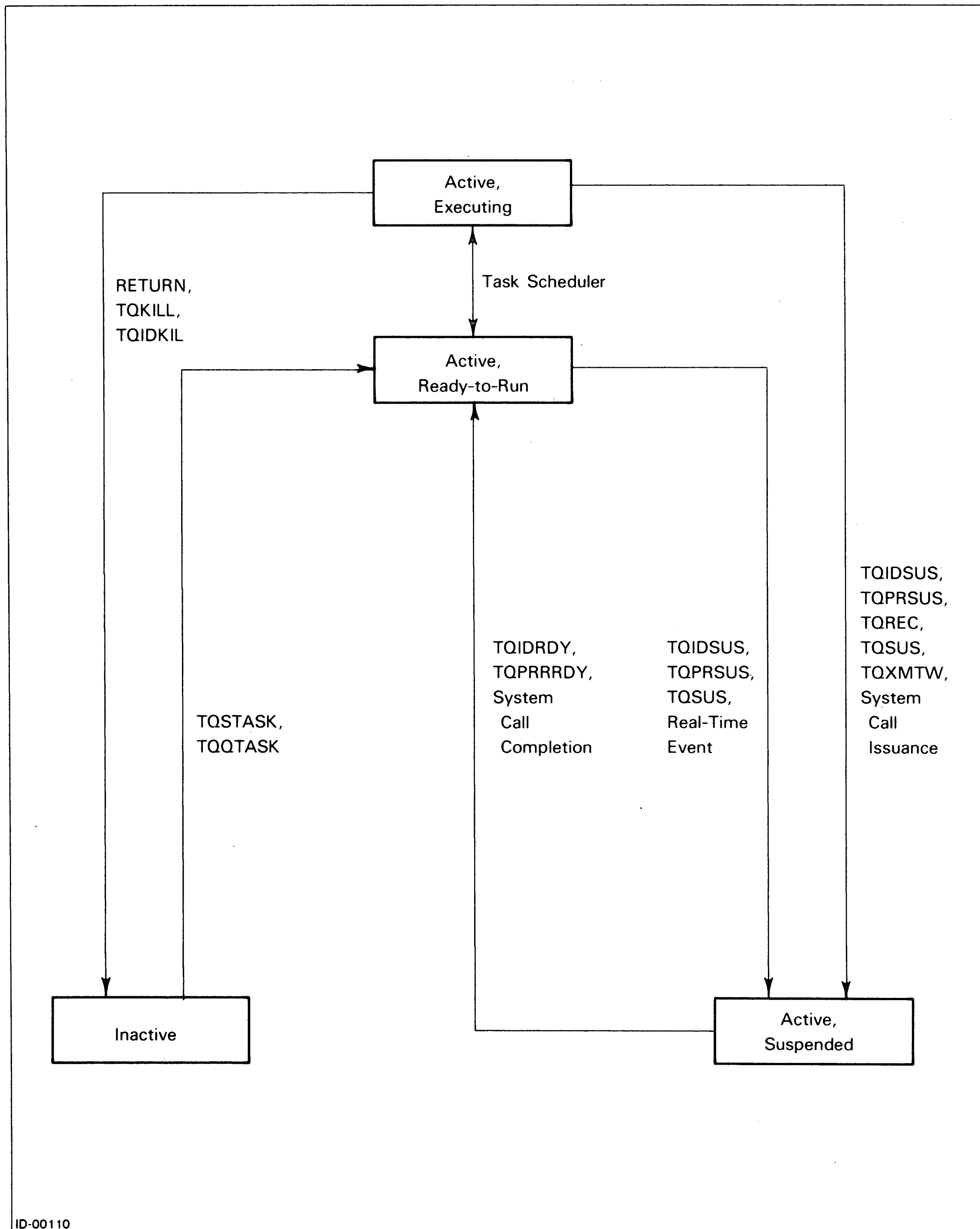


Figure 4-7. Task States and Transitions

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
```

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

```
Source file: TASK1.F77
Compiled on 21-Dec-82 at 14:51:49 by AOS F77 Rev 2.10
Options: F77/INTEGER=2/LOGICAL=2/L=TASK1.LS

   1          SUBROUTINE TASK1
   2
   3          COMMON /COLD/ MAILBOX
   4
   5   %INCLUDE 'TASK1_SYMBOLS.F77.IN'   ! FOR ?DELAY SYSTEM CALL
   6   ****  F77 INCLUDE file for system parameters ****
   7
   8   ****   Parameters for SYSID  ****
   9
  10
  11          INTEGER*2  ISYS_DELAY
  12          PARAMETER (ISYS_DELAY = 13)     ! ?DELAY = 15K
  13
  14   ****    Parameters for PARU   ****
  15
  16
  17
  18   ****  END of F77 INCLUDE file for system parameters ****
  19
  20          OPEN (1, FILE='TASK1.OUT', STATUS='FRESH',
  21       1          RECFM='DATASENSITIVE', CARRIAGECONTROL='LIST')
  22
  23          WRITE (1, 100)
  24     100  FORMAT ('IN FILE TASK1.OUT: TASK1 HAS BEGUN<NL>')
  25
  26   C      SEND THE "MESSAGE" 377K TO ALL TASKS WHO ARE WAITING FOR ONE TO
  27   C         ARRIVE IN A SHARED MEMORY LOCATION ("COMMON MAILBOX"), AND
  28   C         WAIT UNTIL THE MESSAGE ARRIVES.
  29
  30          CALL TQXMTW(MAILBOX, 377K, -1, IER)
  31          IF ( IER .NE. 0 )
  32       1         WRITE (1, 110) IER
  33     110         FORMAT ('ERROR ', O8, ' OCCURRED DURING TQXMTW<NL>')
  34
  35   C      DELAY (SUSPEND) THIS TASK FOR 1.5 SECONDS.
  36
  37          IAC0 = 0        ! AC0 AND
  38          IAC1 = 1500     !     AC1 SPECIFY A DELAY OF 1500 MILLISECONDS
  39          IAC2 = 0
  40          IER = ISYS(ISYS_DELAY, IAC0, IAC1, IAC2)
  41          IF ( IER .NE. 0 ) THEN
  42                 PRINT *, 'ERROR ', IER, ' OCCURRED IN TASK1 DURING ',
  43       1                  'A ?DELAY SYSTEM CALL'
  44                 STOP '-- PROGRAM ENDS NOW'
  45          ENDIF
  46
  47   C      1 1/2 SECONDS HAVE ELAPSED; NOW KILL TASK2.
  48          WRITE (1, 120)
  49     120  FORMAT ('PAST TQXMTW; NOW I KILL TASK2<NL>')
  50
```

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

```
51          CALL TQIDKIL (12, IER)
52          IF ( IER .NE. 0 )
53        1         WRITE (1, 130) IER
54      130         FORMAT ('ERROR ', O8, ' OCCURRED TQIDKILing TASK2<NL>')
55
56          WRITE (1, 140)
57      140 FORMAT ('NOW I RETURN TO MAIN PROGRAM   MAIN5')
58
59          RETURN
60          END
```

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

```
      SUBROUTINE TASK2

      COMMON /COLD/ MAILBOX

      INTEGER 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
```

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   21-OCT-82   16:58:24     104
TASK2.OUT      TXT   21-OCT-82   16:58:24      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   21-OCT-82   16:59:30     104
TASK2.OUT      TXT   21-OCT-82   16:59:30     318
) 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   21-OCT-82   17:00:16    104
TASK2.OUT    TXT   21_OCT-82   17:00:18    1046
)
```

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 data 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
        ...
        CALL TQSTASK(SUBRA, 11, ...)    ! ID IS 11
        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
```

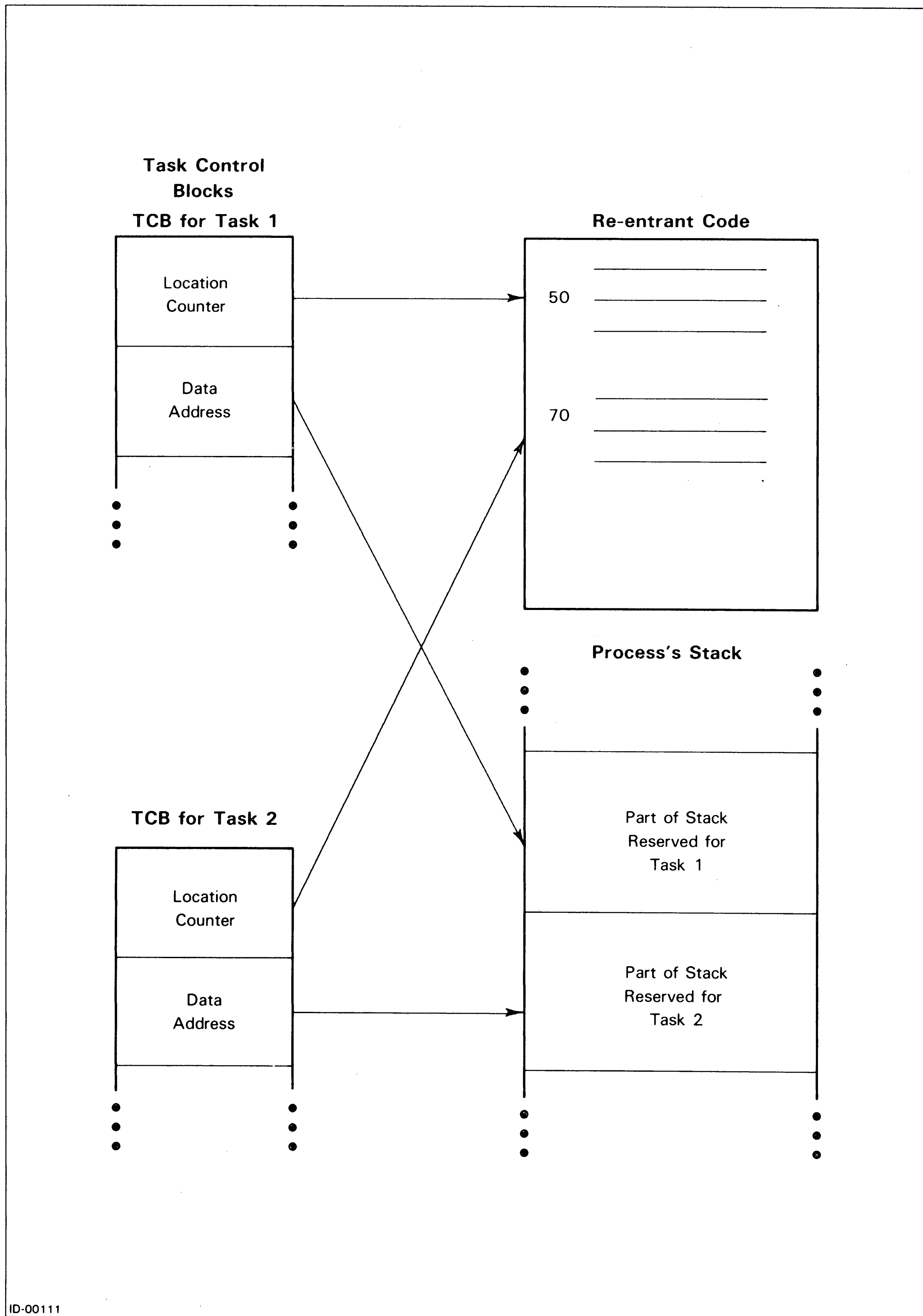


Figure 4-11. Task Control Blocks and the Use of Re-entrant Code

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) 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 Multitasking Calls and their Functions

F77	AOS	Function
TQDQTSK	?DQTSK	Dequeue a previously queued task.
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	?IDSTAT	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	none	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 F77 program unit contains a

CALL TQIDPRI (arguments)

statement. When Link processes the program unit's .OB file, it places code from F77MT.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 F77MT.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 F77MT.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 between the replacement routines and system calls is that the former provide the same protection of the runtime database integrity as do the F77-callable routines; the latter do not.

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

```
EJSR T?<call>   ; make a system call via a routine in F77MT.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 .EXTN statements. For example, you would replace

```
?IDKIL   with   .EXTN T?IDKIL
                 EJSR 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.

```
        .EXTN   T?IDPRI ; DECLARE ROUTINES AS EXTERNALS.
        ; ...
        LDA     0,C7    ; TASK NUMBER 7 WILL HAVE A ...
        LDA     1,C5    ;    ... PRIORITY OF 5.
        EJSR    T?IDPRI ; DO IT!  (FORMERLY: ?IDPRI ; DO IT!)
        JMP     ERIDPRI ; ERROR RETURN
                        ; NORMAL RETURN: CONTINUE
        ; ...
ERIDPRI:                ; RESPOND TO ERROR FROM T?IDPRI.
        ; ...
C7:             7
C5:             5
        ; ...
```

Routine Names

The complete list of multitasking routines accessible via the

EJSR <routine name>

mechanism is as follows.

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

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

EJSR 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.

F77MT.LB provides 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 F77MT.LB and F77ENV_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 library file F77MT.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 RUNTIME ENVIRONMENT.
        SUBROUTINE ARDY (PRIORITY)
        INTEGER PRIORITY, IER
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 F77MT.LB, F77ENV_MT.LB, and other F77 library files. 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 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:

- Using traditional system calls from assembly language programs, such as ?IDPRI.
- Using FORTRAN 77 CALLs such as CALL TQIDPRI (arguments) to multitasking routines in F77MT.LB.
- Using assembly language interface routines for system calls resulting in statements such as EJSR T?IDPRI.

It isn't possible to use the ISYS function to perform AOS multitasking calls. All AOS multitasking routines reside in URT.LB and are linked into .PR files from there. These routines execute code in user context; the F77 ISYS function does not support them.

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 will detect 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 ...

F77LINK responds to the /TASKS switch and chooses the appropriate libraries for linking.

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.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 CLREERMES.SR.

For example, suppose a F77 program contains the statements

```
INTEGER 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*2 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 F77 PROGRAM CALL_TQDQTSK
        EXTERNAL SUB_QDTASK   ! SUBROUTINE WHOSE NAME IS AN ARGUMENT
C                                  TO    TQQTASK
        INTEGER*2 ETDP(0:14)  ! EXTENDED TASK DEFINITION PACKET
C                                  MUST BE INTEGER*2
        INTEGER*2 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 15-WORD EXTENDED TASK DEFINITION PACKET.  TQQTASK
C          WILL USE THIS PACKET AND   TQDQTSK   WILL ALSO USE IT.

        ETDP(00) = 0    ! ?DLNK:    0 TO SPECIFY AN EXTENDED PACKET

        ETDP(01) = 7    ! ?DPRI:    THE PRIORITY NUMBER FOR
                        !              EACH TASK

        ETDP(02) = 14   ! ?DID:     TASK IDS ARE THE NONZERO
                        !              NUMBERS 14, 15, 16, ...

        ETDP(03) = 0    ! ?DPC:     TASK STARTING ADDRESS
                        !              IS SUPPLIED BY F77.

        ETDP(04) = 0    ! ?DAC2:    TASK MESSAGE IS ZERO.
```

```
      ETDP(05) = 0   ! ?DSTB:    ACCEPT F77'S STACK BASE.

      ETDP(06) = 0   ! ?DSSZ:    EACH TASK SHARES A DEFAULT-SIZE STACK.

      ETDP(07) = 0   ! ?DSFLT:   ACCEPT F77'S STACK FAULT HANDLER.

      ETDP(08) = 0   ! ?DFLGS:   0 FOR THIS RESERVED WORD

      ETDP(09) = 0   ! ?DRES:    0 FOR THIS RESERVED WORD

      ETDP(10) = 6   ! ?DNUM:    THERE ARE SIX TASKS.

      ETDP(11) = 14  ! ?DSH:     INITITATE THE TASKS AT THE
      ETDP(12) = 906 ! ?DSMS:       NEXT OCCURRENCE OF 2:15:06 PM.

      ETDP(13) = 3   ! ?DCC:     THREE INITIALIZATION ATTEMPTS
                     !              ARE ENOUGH, ...
      ETDP(14) = 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     DON'T ALTER <ETDP> WHILE IT'S IN THE QUEUE.

C
C     ...
C     START THE TASKS VIA CALLS TO   TQSTASK   .
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*2 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*2 variable or array element that receives the result code.

Example

```
C       SAMPLE F77 PROGRAM CALL_TQDRSCH
        LOGICAL PREV_DIS
        INTEGER 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*2 variable or array element that receives the result code.

Example

```
C       SAMPLE F77 PROGRAM CALL_TQERSCH
        INTEGER IER
C       ...
        CALL TQERSCH(IER)
        PRINT 10, IER
   10   FORMAT (' ERROR CODE RETURNED FROM TQERSCH IS ', 06, 'K')
C       ...
        END
```

TQIDKIL

Kill a task specified by its ID.

Format

CALL TQIDKIL(taskid, ier)

Arguments

taskid is an INTEGER*2 expression that contains the ID of the task you want to kill.

ier is an INTEGER*2 variable or array element that receives the result code.

Example

```
C       SAMPLE F77 PROGRAM CALL_TQIDKIL
        INTEGER 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 ', 06, '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*2 expression that contains the ID of the task whose priority you want to change.

priority — is an INTEGER*2 expression that contains the new priority of the task.

ier — is an INTEGER*2 variable or array element that receives the result code.

Example

```
C       SAMPLE F77 PROGRAM CALL_TQIDPRI
        INTEGER 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*2 expression that contains the ID of the task you want to make ready.

ier is an INTEGER*2 variable or array element that receives the result code.

Example

```
C       SAMPLE AOS PROGRAM CALL_TQIDRDY
        INTEGER 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 ', 06, 'K')
C       ...
        END
```

TQIDSTAT

Get a specified task's status.

Format

CALL TQIDSTAT(taskid, status, ier)

Arguments

taskid is an INTEGER*2 expression that contains the task's ID.

status is an INTEGER*2 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*2 variable or array element that receives the result code.

Example

```
C       SAMPLE F77 PROGRAM CALL_TQIDSTAT
        INTEGER 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 ", 06, "K")
C       ...
        END
```

TQIDSUS

Suspend a task specified by its ID.

Format

CALL TQIDSUS(taskid, ier)

Arguments

taskid is an INTEGER*2 expression that contains the ID of the task you want to suspend.

ier is an INTEGER*2 variable or array element that receives the result code.

Example

```
C       SAMPLE AOS PROGRAM CALL_TQIDSUS
        INTEGER 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*2 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*2 variable or array element that specifies the priority of the task. For proper execution, it should be the highest priority task (i.e., have priority number 0).

ier — is an INTEGER*2 variable or array element that receives the result code.

Example

```
C       SAMPLE F77 PROGRAM CALL_TQIQTSK
        INTEGER 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 0.
        TASK_ID = 5
        PRIORITY = 0
        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, subroutine-name will *not* receive control if task "B" terminates itself via its own TQKILL, STOP, or RETURN statements. Declare subroutine-name EXTERNAL in any task containing a CALL to TQKILAD.

ier — is an INTEGER*2 variable or array element that receives the result code.

Example

```
C       ASSUME THAT THIS IS AOS 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 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*2 variable or array element that receives the result code.

Example

```
C       SAMPLE AOS PROGRAM CALL_TQKILL
        INTEGER 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*2 variable or array element that receives the ID of the calling (i.e., the current) task.

priority — is an INTEGER*2 variable or array element that receives the priority of the calling (i.e., the current) task.

ier — is an INTEGER*2 variable or array element that receives the result code.

Example

```
C       SAMPLE F77 PROGRAM CALL_TQMYTID
        INTEGER 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*2 expression that specifies the new priority of the calling (i.e., the current) task.

ier
: is an INTEGER*2 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 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*2 expression that specifies the priority of the tasks to be killed.

ier is an INTEGER*2 variable or array element that receives the result code.

Example

```
C       SAMPLE AOS PROGRAM CALL_TQPRKIL
        INTEGER 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
```


Licensed Material-Property of Data General Corporation

TQPROT

Start a protected area.

Format

CALL TQPROT(ier)

Argument

ier is an INTEGER*2 variable or array element that receives the result code.

Explanation

This routine has no direct counterpart in AOS.

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 task that eventually kills or suspends A is unknown.

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 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*2 expression that specifies the priority of the tasks to be made ready.

ier is an INTEGER*2 variable or array element that receives the result code.

Example

```
C       SAMPLE AOS PROGRAM CALL_TQPRRDY
        INTEGER 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
```


Licensed Material-Property of Data General Corporation

TQPRSUS

Suspend all tasks of a specified priority.

Format

CALL TQPRSUS(priority, ier)

Arguments

priority is an INTEGER*2 expression that specifies the priority of the tasks to be suspended.

ier is an INTEGER*2 variable or array element that receives the result code.

Example

```
C       SAMPLE F77 PROGRAM CALL_TQPRSUS
        INTEGER 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 ', 06, '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*2 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. Every task must have a unique ID number.
- ?DSTB (stack base), if zero or negative, is replaced by F77's own value. Otherwise, F77 uses a positive number as the address of the stack base. Then, you must declare an array of length ?DNUM*?DSSZ and use the WORDADDR function to place the address of this array in ?DSTB.
- ?DSFLT (stack fault handler) is replaced by F77's own value.
- If you set ?DSSZ (stack size) to zero, F77 provides a default size.
- ?DPC is replaced (by F77) by the subroutine's address.

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*2 variable or array element that specifies the word from which you will receive a message from another task. Note, mailbox must be in a COMMON area shared by both this and the sending task.

message — is an INTEGER*2 variable or array element that contains a nonzero message; this message arrives from the previous argument, mailbox.

ier — is an INTEGER*2 variable or array element that receives the result code.

Example

```
C       SAMPLE F77 PROGRAM CALL_TQREC
        INTEGER MAILBOX, MESSAGE, IER
        COMMON /COLD/ MAILBOX
C       ...
C       SEE IF THERE IS A NON-ZERO MESSAGE IN VARIABLE MAILBOX.  IF
C          SO, MOVE THE CONTENTS OF MAILBOX TO VARIABLE MESSAGE
C          AND PLACE A ZERO IN VARIABLE MAILBOX.  IF THERE IS NO
C          SUCH MESSAGE, THEN WAIT FOR THE MESSAGE.
        CALL TQREC (MAILBOX, MESSAGE, IER)
        PRINT 10, MESSAGE, IER
   10   FORMAT (' MESSAGE RECEIVED IS: ', O6, 'K', /,
     1         ' ERROR CODE VALUE IS: ', O6, 'K')
C       ...
        END
```

TQRECNW

Receive an intertask message without waiting.

Format

CALL TQRECNW(mailbox, message, ier)

Arguments

mailbox — is an INTEGER*2 variable or array element that specifies the word from which you will receive a message from another task. Note, mailbox must be in a COMMON area shared by both this and the sending task.

message — is an INTEGER*2 variable or array element that contains a nonzero message; this message arrives from the previous argument, mailbox.

ier — is an INTEGER*2 variable or array element that receives the result code.

Example

```
C       SAMPLE F77 PROGRAM CALL_TQRECNW
        INTEGER MAILBOX, MESSAGE, IER
        COMMON /COLD/ MAILBOX
C       ...
C       SEE IF AOS HAS PLACED A ONE-WORD MESSAGE IN VARIABLE MAILBOX.
C          IF SO, MOVE THE CONTENTS OF MAILBOX TO VARIABLE
C          MESSAGE, AND PLACE 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: ', O6, '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*2 expression that contains the task's ID number.

priority — is an INTEGER*2 expression between 0 and 255, inclusive, which specifies the task's priority.

stacksize — is an INTEGER*2 expression that specifies the size of your stack in words.

You can specify zero, and F77 creates a default-size stack. It creates such a stack by dividing available memory equally among the n tasks specified by the F77LINK switch /TASKS=n.

If you specify stacksize, it should be large enough for your task's local variables, return blocks, and I/O buffers. For more information, see the last section of this chapter ("AOS F77 Multitask Stack Definition").

ier — is an INTEGER*2 variable or array element that receives the result code.

Example

```
C       SAMPLE AOS PROGRAM CALL_TQSTASK
        INTEGER 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*2 variable or array element that receives the result code.

Example

```
C       SAMPLE AOS PROGRAM CALL_TQSUS
        INTEGER 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 ', O6, 'K')
C       ...
        END
```


Licensed Material-Property of Data General Corporation

TQUNPROT

Exit a protected area.

Format

CALL TQUNPROT(ier)

Argument

ier   is an INTEGER*2 variable or array element that receives the result code.

Explanation

This routine has no direct counterpart in AOS.

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*2 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 COMMON area shared by the receiving task or tasks, and mailbox must contain zero before the call.

message is an INTEGER*2 expression that contains a nonzero message; this message goes to the previous argument, mailbox.

flag is an INTEGER*2 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*2 variable or array element that receives the result code.

Example

```
C       SAMPLE F77 PROGRAM CALL_TQXMT
        INTEGER 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*2 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 COMMON area shared by the receiving task or tasks, and mailbox must contain zero before the call.

message — is an INTEGER*2 expression that contains a nonzero message; this message goes to the previous argument, mailbox.

flag — is an INTEGER*2 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*2 variable or array element that receives the result code.

Example

```
C       SAMPLE F77 PROGRAM CALL_TQXMTW
        INTEGER 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 the tasks' output.
- TASK11 writes a message into TASK0.OUT every 5 seconds.
- TASK12 writes a message into TASK0.OUT every 15 seconds.
- 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 05-Nov-82 at 14:30:52 by AOS F77 Rev 2.10
Options: F77/INTEGER=2/LOGICAL=2/L=TASKO.LS

    1          PROGRAM TASKO          ! MAIN PROGRAM TO INITIALIZE TASKS
    2   C                                  TASK11, TASK12, TASK13, TASK14,
    3   C                                  AND TASK15.
    4
    5          EXTERNAL TASK11, TASK12, TASK13, TASK14, TASK15
    6
    7          COMMON /COLD/ MAIL34, MAIL45, IARRAY(10) ! FOR TASK13 -> TASK14
    8   C                        COMMUNICATION, TASK14 -> TASK 15 COMMUNICATION,
    9   C                        AND THE ARRAY TO BE OBTAINED, SORTED, AND PRINTED.
   10          MAIL34 = 0
   11          MAIL45 = 0
   12
   13   C      ALL OUTPUT GOES TO FRESH FILE <TASKO.OUT>.
   14          OPEN (1, FILE='TASKO.OUT', STATUS = 'FRESH',
   15        1          RECFM='DATASENSITIVE', CARRIAGECONTROL='LIST')
   16          WRITE (1, 10)
   17      10  FORMAT ('IN FILE TASKO.OUT: TASKO HAS BEGUN<NL>')
   18
   19   C      INITIATE THE TASKS VIA SUBROUTINE <TQSTASK> BY GIVING AS
   20   C         ARGUMENTS EACH TASKS'S NAME, ID NUMBER, PRIORITY,
   21   C         AND SYSTEM-SELECTED STACK SIZE.
   22
   23          CALL TQSTASK (TASK11, 11, 7, 0, IER)
   24          IF ( IER .NE. 0 ) THEN
   25                 PRINT *, 'ERROR ', IER, ' OCCURRED IN TASKO WHILE ',
   26        1                 'INITIATING TASK11'
   27                 STOP '-- PROGRAM ENDS NOW'
   28          ENDIF
   29
   30          CALL TQSTASK (TASK12, 12, 7, 0, IER)
   31          IF ( IER .NE. 0 ) THEN
   32                 PRINT *, 'ERROR ', IER, ' OCCURRED IN TASKO WHILE ',
   33        1                 'INITIATING TASK12'
   34                 STOP '-- PROGRAM ENDS NOW'
   35          ENDIF
   36
   37          CALL TQSTASK (TASK13, 13, 7, 0, IER)
   38          IF ( IER .NE. 0 ) THEN
   39                 PRINT *, 'ERROR ', IER, ' OCCURRED IN TASKO WHILE ',
   40        1                 'INITIATING TASK13'
   41                 STOP '-- PROGRAM ENDS NOW'
   42          ENDIF
   43
   44          CALL TQSTASK (TASK14, 14, 7, 0, IER)
   45          IF ( IER .NE. 0 ) THEN
   46                 PRINT *, 'ERROR ', IER, ' OCCURRED IN TASKO WHILE ',
   47        1                 'INITIATING TASK14'
   48                 STOP '-- PROGRAM ENDS NOW'
   49          ENDIF
```

Figure 4-12. A Listing of Subroutine TASK_11.F77 (continues)

```
50
51          CALL TQSTASK (TASK15, 15, 7, 0, IER)
52          IF ( IER .NE. 0 ) THEN
53                  PRINT *, 'ERROR ', IER, ' OCCURRED IN TASKO WHILE ',
54       1                   'INITIATING TASK15'
55                  STOP '-- PROGRAM ENDS NOW'
56          ENDIF
57
58    C     I'M DONE.
59          PRINT *, 'TASKO IS DYING'
60          CALL TQKILL (IER)
61          IF ( IER .NE. 0 ) THEN
62                  PRINT *, 'ERROR ', IER, ' OCCURRED IN TASKO WHILE ',
63       1                   'KILLING (TQKILL) TASKO'
64                  STOP '-- PROGRAM ENDS NOW'
65          ENDIF
66
67          END
```

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

```
Source file: TASK11.F77
Compiled on 05-Nov-82 at 14:31:44 by AOS F77 Rev 2.10
Options: F77/INTEGER=2/LOGICAL=2/L=TASK11.LS

   1          SUBROUTINE TASK11
   2
   3   C      THIS TASK WRITES A MESSAGE INTO FILE <TASKO.OUT> EVERY 5
   4   C         SECONDS.  <TASKO.OUT> IS OPENED BY MAIN PROGRAM <TASKO>.
   5
   6          COMMON /COLD/ MAIL34, MAIL45, IARRAY(10)
   7          DIMENSION ITIME(3)
   8
   9   %INCLUDE 'TASK11_SYMBOLS.F77.IN'   ! FOR ?DELAY SYSTEM CALL
  10   ****  F77 INCLUDE file for system parameters ****
  11
  12   ****   Parameters for SYSID  ****
  13
  14
  15          INTEGER*2  ISYS_DELAY
  16          PARAMETER (ISYS_DELAY = 13)     ! ?DELAY = 15K
  17
  18   ****    Parameters for PARU   ****
  19
  20
  21
  22   ****  END of F77 INCLUDE file for system parameters ****
  23
  24          WRITE (1, 10)
  25      10  FORMAT ('IN FILE TASKO.OUT: TASK11 HAS BEGUN <NL>')
  26
  27      20  IACO = 0           ! ACO AND
  28          IAC1 = 5000        !     AC1 SPECIFY A DELAY OF 5000 MILLISECONDS
  29          IAC2 = 0
  30   C      DELAY (SUSPEND) THIS TASK FOR 5 SECONDS.
  31          IER = ISYS(ISYS_DELAY, IACO, IAC1, IAC2)
  32          IF ( IER .NE. 0 ) THEN
  33                  PRINT *, 'ERROR ', IER, ' OCCURRED IN TASK11 DURING ',
  34       1                   'A ?DELAY SYSTEM CALL'
  35                  STOP '-- PROGRAM ENDS NOW'
  36          ENDIF
  37
  38          CALL TIME (ITIME)
  39          WRITE (1, 30) ITIME
  40      30  FORMAT ('TASK11 REPORTS AFTER A  5-SECOND DELAY AT ',
  41       1                I2, ':', I2, ':', I2)
  42          GO TO 20
  43
  44          END
```

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

```
Source file: TASK12.F77
Compiled on 05-Nov-82 at 14:32:15 by AOS F77 Rev 2.10
Options: F77/INTEGER=2/LOGICAL=2/L=TASK12.LS

    1           SUBROUTINE TASK12
    2
    3    C       THIS TASK WRITES A MESSAGE INTO FILE <TASKO.OUT> EVERY 15
    4    C          SECONDS.  <TASKO.OUT> IS OPENED BY MAIN PROGRAM <TASKO>.
    5
    6           COMMON /COLD/ MAIL34, MAIL45, IARRAY(10)
    7           DIMENSION ITIME(3)
    8
    9    %INCLUDE 'TASK12_SYMBOLS.F77.IN'   ! FOR ?DELAY SYSTEM CALL
   10    ****  F77 INCLUDE file for system parameters ****
   11
   12    ****   Parameters for SYSID  ****
   13
   14
   15           INTEGER*2  ISYS_DELAY
   16           PARAMETER (ISYS_DELAY = 13)     ! ?DELAY = 15K
   17
   18    ****    Parameters for PARU   ****
   19
   20
   21
   22    ****  END of F77 INCLUDE file for system parameters ****
   23
   24           WRITE (1, 10)
   25       10   FORMAT ('IN FILE TASKO.OUT: TASK12 HAS BEGUN <NL>')
   26
   27       20   IAC0 = 0           ! AC0 AND
   28           IAC1 = 15000       !     AC1 SPECIFY A DELAY OF 15000 MILLISECONDS
   29           IAC2 = 0
   30    C       DELAY (SUSPEND) THIS TASK FOR 15 SECONDS.
   31           IER = ISYS(ISYS_DELAY, IAC0, IAC1, IAC2)
   32           IF ( IER .NE. 0 ) THEN
   33                   PRINT *, 'ERROR ', IER, ' OCCURRED IN TASK12 DURING ',
   34        1                   'A ?DELAY SYSTEM CALL'
   35                   STOP '-- PROGRAM ENDS NOW'
   36           ENDIF
   37
   38           CALL TIME (ITIME)
   39           WRITE (1, 30) ITIME
   40       30   FORMAT ('TASK12 REPORTS AFTER A 15-SECOND DELAY AT ',
   41        1                I2, ':', I2, ':', I2)
   42           GO TO 20
   43
   44           END
```

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

```
Source file: TASK13.F77
Compiled on 05-Nov-82 at 14:32:45 by AOS F77 Rev 2.10
Options: F77/INTEGER=2/LOGICAL=2/L=TASK13.LS

    1           SUBROUTINE TASK13
    2
    3   C       THIS TASK ACCEPTS INTO <IARRAY> 10 INTEGERS FROM THE CONSOLE
    4   C          AND THEN SENDS A MESSAGE TO <TASK14>.
    5
    6           COMMON /COLD/ MAIL34, MAIL45, IARRAY(10)
    7
    8           PRINT *
    9           PRINT *, 'GIVE ME 10 INTEGERS'
   10           PRINT *
   11
   12           DO 10 I = 1, 10
   13           PRINT *, 'INTEGER NUMBER ', I, ' ?  '
   14           READ *, IARRAY(I)
   15      10   CONTINUE
   16
   17           PRINT *
   18
   19   C       NOTIFY <TASK14> THAT I'M DONE SO IT CAN SORT <IARRAY>.
   20   C          I'LL SEND IT THE NUMBER   3   AS THE MESSAGE.
   21
   22           CALL TQXMT (MAIL34, 3, -1, IER)
   23
   24           IF ( IER .NE. 0 ) THEN
   25                   PRINT *, 'ERROR ', IER, ' OCCURRED IN TASK13 DURING ',
   26        1                   'A CALL TO   TQXMT   .'
   27                   STOP '-- PROGRAM ENDS NOW'
   28           ENDIF
   29
   30           END
```

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

```
Source file: TASK14.F77
Compiled on 05-Nov-82 at 14:33:24 by AOS F77 Rev 2.10
Options: F77/INTEGER=2/LOGICAL=2/L=TASK14.LS

    1             SUBROUTINE TASK14
    2
    3    C        THIS TASK AWAITS THE RECEIPT OF THE MESSAGE WHOSE VALUE IS 3
    4    C           FROM <TASK13>.  THEN, IT SORTS THE ELEMENTS OF <IARRAY>
    5    C           INTO ASCENDING ORDER AND FINISHES BY SENDING A MESSAGE TO
    6    C           <TASK14>.
    7
    8             COMMON /COLD/ MAIL34, MAIL45, IARRAY(10)
    9
   10       10    CALL TQREC (MAIL34, MESSAGE, IER)
   11             IF ( IER .NE. 0 ) THEN
   12                     PRINT *, 'ERROR ', IER, ' OCCURRED IN TASK14 DURING ',
   13        1                     'A CALL TO   TQREC    '
   14                     STOP '-- PROGRAM ENDS NOW'
   15             ENDIF
   16
   17    C        <MESSAGE> MUST BE 3; WAIT SOME MORE IF IT ISN'T
   18             IF ( MESSAGE .EQ. 3 ) GO TO 20
   19             GO TO 10        ! <MESSAGE> DOES NOT CONTAIN 3.
   20
   21       20    CONTINUE        ! <MESSAGE> DOES CONTAIN 3.
   22       30    KSWAP = 0       ! COUNT OF SWAPS FOR THE NEXT PASS THROUGH <IARRAY>
   23
   24             DO 40 I = 1, 9
   25             IF ( IARRAY(I) .LE. IARRAY(I+1) ) GO TO 40
   26    C        SWAP THE CONTENTS OF THE CURRENT TWO <IARRAY> ELEMENTS.
   27             ITEMP = IARRAY(I)
   28             IARRAY(I) = IARRAY(I+1)
   29             IARRAY(I+1) = ITEMP
   30             KSWAP = KSWAP + 1    ! COUNT THIS SWAP
   31       40    CONTINUE
   32             IF ( KSWAP .GE. 1 ) GO TO 30 ! <IARRAY> MIGHT NOT BE SORTED YET
   33
   34    C        <IARRAY> IS SORTED NOW, SO SEND A MESSAGE WHOSE VALUE IS 4
   35    C           TO <TASK15>.
   36
   37             CALL TQXMT (MAIL45, 4, -1, IER)
   38             IF ( IER .NE. 0 ) THEN
   39                     PRINT *, 'ERROR ', IER, ' OCCURRED IN TASK14 DURING ',
   40        1                     'A CALL TO   TQXMT    '
   41                     STOP '-- PROGRAM ENDS NOW'
   42             ENDIF
   43
   44             END
```

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

```
Source file: TASK15.F77
Compiled on 05-Nov-82 at 14:34:03 by AOS F77 Rev 2.10
Options: F77/INTEGER=2/LOGICAL=2/L=TASK15.LS

    1          SUBROUTINE TASK15
    2
    3   C      THIS TASK AWAITS THE RECEIPT OF THE MESSAGE WHOSE VALUE IS 4
    4   C         FROM <TASK14>.  THEN, IT DISPLAYS THE SORTED ELEMENTS OF
    5   C         <IARRAY> AND SEQUENTIALLY KILLS ALL ACTIVE TASKS, INCLUDING
    6   C         ITSELF.
    7
    8          COMMON /COLD/ MAIL34, MAIL45, IARRAY(10)
    9
   10      10  CALL TQREC (MAIL45, MESSAGE, IER)
   11          IF ( IER .NE. 0 ) THEN
   12                 PRINT *, 'ERROR ', IER, ' OCCURRED IN TASK15 DURING ',
   13       1                  'A CALL TO   TQREC    '
   14                 STOP '-- PROGRAM ENDS NOW'
   15          ENDIF
   16
   17   C      <MESSAGE> MUST BE 4; WAIT SOME MORE IF IT ISN'T
   18          IF ( MESSAGE .EQ. 4 ) GO TO 20
   19          GO TO 10        ! <MESSAGE> DOES NOT CONTAIN 4.
   20
   21      20  CONTINUE        ! <MESSAGE> DOES CONTAIN 4.
   22
   23          DO 30 I = 1, 10
   24          PRINT *, I, '<TAB>', IARRAY(I)
   25      30  CONTINUE
   26
   27          WRITE (1, 40)    ! CLEAN-UP MESSAGE
   28      40  FORMAT ('<NL>*** TASK 15 REPORTS: THIS IS THE LAST RECORD ***<NL>')
   29
   30   C      KILL THE OTHER TASKS AND THEN MYSELF.
   31          PRINT *
   32          PRINT *, 'TASK15 IS ABOUT TO KILL ALL OTHER TASKS AND THEN ITSELF'
   33          PRINT *
   34
   35          DO 50 I = 11, 15
   36          CALL TQIDKIL(I, IER)
   37          IF ( IER .NE. 0 ) THEN
   38                 PRINT *
   39                 PRINT *, 'ERROR ', IER, ' OCCURRED IN TASK15 DURING ',
   40       1                  'A CALL TO   TQIDKIL    '
   41                 PRINT *, 'THE ID OF THE TASK   TQIDKIL   FAILED ON IS ', I
   42                 PRINT *
   43          ENDIF
   44      50  CONTINUE
   45
   46          END
```

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. Macro 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

IN FILE TASK0.OUT: TASK12 HAS BEGUN

IN FILE TASK0.OUT: TASK11 HAS BEGUN

```
TASK11 REPORTS AFTER A  5-SECOND DELAY AT 14:35:21
TASK11 REPORTS AFTER A  5-SECOND DELAY AT 14:35:26
TASK12 REPORTS AFTER A 15-SECOND DELAY AT 14:35:31
TASK11 REPORTS AFTER A  5-SECOND DELAY AT 14:35:31
TASK11 REPORTS AFTER A  5-SECOND DELAY AT 14:35:36
TASK11 REPORTS AFTER A  5-SECOND DELAY AT 14:35:41
TASK12 REPORTS AFTER A 15-SECOND DELAY AT 14:35:46
TASK11 REPORTS AFTER A  5-SECOND DELAY AT 14:35:46
TASK11 REPORTS AFTER A  5-SECOND DELAY AT 14:35:52

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

Several questions arise from an examination of TASK0's output. We also present some answers.

1. Why does error 12, "TASK I.D. ERROR" (from the symbol ERTID in PARU.SR), occur when TASK15 issues a TQIDKIL call to tasks with ID numbers 13 and 14?

   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.

2. TASK0 issues TQSTASK calls to TASK11 and TASK 12 in that order. Yet, TASK12 places its start-up message in TASK0.OUT before TASK11 does. Why?

   The task scheduler has many steps to perform, and these steps have certain time-dependent relationships. As a result, the order of task execution can vary slightly from one execution of a program to another. Repeated execution of TASK0.PR would vary the order of appearance of the initial messages from TASK11 and TASK12 in TASK0.OUT.

3. TASK0.OUT shows TASK11 reporting at 14:35:46 and next at 14:35:52. These times are 6 seconds apart, not 5. Why?

   Again, the task scheduler has much work to do. The fraction of a second that elapses between the completion of the ?DELAY and ?GTOD system calls can be enough to result in a reported difference of 6 seconds.

AOS F77 Multitask Stack Definition

An AOS F77 multitask program, during its initialization phase at runtime, divides all available memory into a number of memory blocks. Each task has a memory block to contain the task's stack and other information about the task. The portion of a memory block with this other information is called the *per task area.* The initialization phase follows these steps:

1. Obtain all stack memory available to the process.
2. If the program file does not include an .OB file that used macro F77STACK to request specific stack sizes, then go to step 3. Otherwise, the program requests specific stack lengths via the inclusion of specifications from macro F77STACK; so, create the memory blocks to provide the requested stacks.
3. Calculate the number of default-size memory blocks to create. This equals the number of tasks specified with the /TASKS= F77LINK switch, minus the number of memory blocks set aside in step 2. If you have not used macro F77STACK to request specific stack lengths, then the number of tasks given by the /TASKS= F77LINK switch specifies the number of default-size memory blocks (each containing one default-size stack) that will be created.
4. Set aside the memory blocks for the default-size stacks by dividing the amount of remaining available memory by the number of default-size stacks needed.

   If any stack specified by macro F77STACK cannot fit within remaining available memory, then F77 reports a STACK OVERFLOW error during initialization.

The runtime routines select a memory block for a task when they create it. As you have seen, this block is used for task information and for its runtime stack. The method the F77 runtime routines use to select an appropriate memory block follows these basic steps:

1. Obtain the requested stack size, specified either by the <stacksize> argument to TQSTASK or the ?DSSZ value of <task_definition_packet> to TQQTASK.
2. Scan a list of "memory block identifiers" to find free blocks of memory that are large enough to provide the requested stack space. The previous four steps explain the creation of the blocks of memory.
3. Choose the closest fitting memory block, lock it in use, build a task packet, and start a task whose stack is in the chosen memory block.

   If no memory block is found that is large enough, then an error is returned to the routine attempting to start the task.

File F77STACK.SR, which comes with your release of AOS F77, contains two assembly language (MASM) macros. You use these macros to request special stack lengths to the F77 runtime initializer. The macros are named F77STACK and MAINSTACK; their descriptions follow.

Macro F77STACK

This macro requests the initialization of memory blocks with a specified stack length. The following rules apply to F77STACK:

- All numbers are decimal.
- The first call must be

  ```
  F77STACK
  ```

  to initialize the request table.
- The last call must be

  ```
  F77STACK
  ```

  to terminate the request table.
- Calls in between can be either

  ```
  F77STACK <size>
  ```

  or

  ```
  F77STACK <size> <count>
  ```

  where:

  <size> is the number of words to be reserved for a stack.

  <count> is the number of stacks of length <size>. If <count> is absent, then its assumed value is 1.

Macro MAINSTACK

This macro specifies a stack size for the main task. The following rules apply to MAINSTACK:

- All numbers are decimal.
- Make the call by writing

  ```
  MAINSTACK <size>
  ```

where:

<size> is the stack size for the main task.

NOTE: This macro does not reserve any stack area. It simply selects an appropriately sized stack from those stacks previously defined to the F77STACK macro.

Example Entries for F77STACK.SR

You create an assembly language source file whose principal entries have a pattern such as follows.

```
F77STACK                ; FIRST call INITIALIZES the table.
F77STACK  500           ; Request 1 stack of 500. words.
F77STACK  300  8        ; Request 8 stacks of 300. words.
F77STACK                ; LAST call TERMINATES the table.

MAINSTACK 500           ; The main program will use the
                        ;    500-word stack requested above.
```

The object module resulting from the assembly of these entries requests the F77 runtime initializer to set up memory blocks for (1+8)=9 stacks. If the /TASKS= F77LINK switch indicates a number of tasks greater than 9, then the F77 initializer will calculate the difference and set up default-sized stacks for the additional (beyond 9) tasks.

Operating Instructions for F77STACK

Perform the following steps to use macro F77STACK.

1. Create an assembly language source file. We refer to it here as YOURFILE.SR for identification purposes only, but you can choose any appropriate name. A .TITLE statement should appear early in YOURFILE.SR and an .END statement should appear at its end.
2. Insert your selected F77STACK and MAINSTACK entries in YOURFILE.SR.
3. Assemble YOURFILE.SR. The command is

   ```
   X MASM/8 F77STACK/S YOURFILE
   ```

4. Create a program file containing the stack creation instructions from YOURFILE.OB. The general command is

   ```
   F77LINK/TASKS=number mainprogram YOURFILE
   ```

How Necessary is F77STACK?

In most cases it will *not* be necessary to use either of the F77STACK or MAINSTACK macros. When the program file doesn't include an object module that requests any special stack lengths, F77 will automatically create the /TASKS= number of default-size stacks. For many programs the default-size stacks are large enough and there is no need to use F77STACK.SR. However, if a CALL to TQSTASK or TQQTASK requests a stack size larger than the default value, a runtime error occurs.

An Example of Specific Stack Specifications

For some large and complicated multitasked programs it may be necessary to divide up the stack areas in a specific manner. For example, consider a program where one task acts as a controller of several minor tasks and some "average" size tasks. The controller task has a large number of local variables, calls several levels of subroutines, and does I/O, while the minor tasks do only arithmetic calculations

on scalar variables. The one controller task might need a stack that is larger than the default, while the smaller tasks will not need all the stack memory of a default size stack. In this case, you want to request special stack sizes with the F77STACK macro. To make these special requests, create an assembly language source file such as the following.

```
.TITLE          SAMPLE

F77STACK          ; Initialize the request table.
F77STACK  12000   ; Request a stack of 12000 words for
                  ;    the controller task.
F77STACK  199 3   ; Request 3 stacks of 199 words for
                  ;    the minor tasks.
; Make no specific request for the "average" tasks.
F77STACK          ; Terminate the request table.

; Also assume that the controller task is
;    the main/initial task, and we want it to have
;    a stack of 12000. words.

MAINSTACK 12000 ; Assign the main program the 12000. word
                ;    stack specified above.

.END
```

If the program file is named CONTROLLER.F77 and has been compiled, then give the following commands to create and execute CONTROLLER.PR with the stack sizes specified in SAMPLE.SR.

```
X MASM/8 F77STACK/S SAMPLE
F77LINK/TASKS=6 CONTROLLER SAMPLE
XEQ CONTROLLER
```

During the program initialization stage of CONTROLLER.PR's execution, available memory will include memory blocks containing:

- One stack of 12000 words.
- Three stacks of 199 words.
- Two default size stacks.

If CONTROLLER.F77 contains statements equivalent to the following five, then the tasks will have the desired stacks.

```
CALL TQSTASK (MINOR_1, 11, 5, 199, IER)
CALL TQSTASK (MINOR_2, 12, 5, 199, IER)
CALL TQSTASK (MINOR_3, 13, 5, 199, IER)
CALL TQSTASK (AVERAGE_1, 21, 3, 0, IER)
CALL TQSTASK (AVERAGE_2, 22, 3, 0, IER)
```

The main task, because of the creation of CONTROLLER.PR by F77LINK with SAMPLE.OB, will have a stack of 12000 words.

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 22-Oct-82 at 17:10:36 by AOS F77 Rev 2.10
Options: F77/INTEGER=2/LOGICAL=2/DEBUG/L=TEST_SORT10.LS

   1          PROGRAM TEST_SORT10           !  TO TEST SUBROUTINE SORT10
   2
   3          CHARACTER*80 ALL_OF_THE_NAMES  ! ALL THE NAMES, IN ONE CONVENIENT
   4    C                                      AND EASY-TO-CONSTRUCT STRING
   5          CHARACTER*10 NAMES(8)         ! <NAMES> WILL CONTAIN THE EIGHT
   6    C                                     ELEMENTS THAT <SORT10> WILL
   7    C                                     SORT ALPHABETICALLY.
   8
   9    C     THE NEXT TWO LINES HELP TO CONSTRUCT <ALL_OF_THE_NAMES>.
  10    C000000001111111111222222222233333333334444444444555555555566666666666777
  11    C23456789012345678901234567890123456789012345678901234567890123456789012
  12          DATA  ALL_OF_THE_NAMES  / 'MIKE      HENRIETTA ENRICO    LISA
  13         + JEFFREY   BETSY     ALICE     NORMAN    ' /
  14
  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 '.
  21       10 CONTINUE
  22
  23    C     SORT THE NAMES INTO ALPHABETICAL ORDER.
  24          CALL SORT10 (NAMES, 8)
  25
  26    C     PRINT THE RESULTS.
  27          WRITE (10, *)
  28          WRITE (10, *) 'THE SORTED NAMES ARE:'
  29          WRITE (10, *)
  30          DO 30 I = 1, 8
  31                WRITE (10, *) NAMES(I)
  32       30 CONTINUE
  33
  34          WRITE (10, *)
  35          WRITE (10, *) '*** END OF JOB ***'
  36          STOP
  37          END
```

```
Source file: SORT10.F77
Compiled on 22-Oct-82 at 17:12:07 by AOS F77 Rev 2.10
Options: F77/INTEGER=2/LOGICAL=2/DEBUG/L=SORT10.LS

    1         SUBROUTINE SORT10 (C_ARRAY, N)
    2
    3   C     THIS SUBROUTINE SORTS THE FIRST <N> ELEMENTS OF A
    4   C        CHARACTER*10 ARRAY, <C_ARRAY>, WITH AT MOST 100 ELEMENTS
    5   C        (EACH 10 BYTES LONG).
    6
    7   C     SORTING METHOD: TRADITIONAL "BUBBLE" SORT WHICH MOVES THE
    8   C        HIGHER-VALUED  ELEMENTS (SUCH AS "ZACHARY") TO THE RIGHT IN
    9   C        THE ARRAY AND THE LOWER-VALUED ELEMENTS (SUCH AS "AMANDA")
   10   C        TO THE LEFT ELEMENTS OF THE ARRAY.
   11
   12         CHARACTER*10 C_ARRAY(100)
   13         CHARACTER*10 TEMP         ! TEMPORARY STORAGE AREA REQUIRED
   14   C                                    BY THE SORT ROUTINE
   15
   16         IF ( N .LT. 2 ) GO TO 30  ! NO NEED TO SORT.
   17
   18         N_LESS_1 = N - 1
   19
   20   C     HERE WE GO ...
   21
   22         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
   36              TEMP = C_ARRAY(I)
   37              C_ARRAY(I) = C_ARRAY(I+1)
   38              C_ARRAY(I+1) = TEMP
   39
   40      10      CONTINUE
   41      20 CONTINUE
   42
   43   C     DONE!
   44
   45      30 RETURN
   46
   47         END
```


Licensed Material-Property of Data General Corporation

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
ENVIRONMENT	Select the program unit, usually used to move from one program unit to another (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, use

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.

```
 1  SWAT REVISION 02.00 ON 10/25/82 AT 11:27:26
 2  PROGRAM -- :UDD:MARLL:F77:TEST_SORT10
 3  > %
 4  > % Set a breakpoint to see if <NAMES> receives its elements correctly
 5  > %    from <ALL_OF_THE_NAMES>.
 6  > BREAKPOINT 21
 7  Set at :TEST_SORT10:21
 8  > %
 9  > % Also set a breakpoint just before the CALL to SORT10.
10  > BREAKPOINT 24
11  Set at :TEST_SORT10:24
12  > %
13  > % Verify the breakpoints.
14  > LIST 20, 25
15  20 C                        'HENRIETTA ' AND <NAMES(2)> IS ALSO 'HENRIETTA '.
16  21B          10 CONTINUE
17  22
18  23 C     SORT THE NAMES INTO ALPHABETICAL ORDER.
19  24B             CALL SORT10 (NAMES, 8)
20  25
21  > %
22  > % Move to subroutine SORT10 and set appropriate breakpoints.
23  > ENVIRONMENT :SORT10
24  :SORT10
25  > BREAKPOINT 22, 36
26  Set at :SORT10:22
27  Set at :SORT10:36
28  > %
29  > % Verify the breakpoints.
30  > LIST 22, 36
31  22B             DO 20 J = 1, N_LESS_1
32  23            M = N-J
33  24
34  25            DO 10 I = 1, M
35  26            IF ( C_ARRAY(I)     .LE.
36  27      1          C_ARRAY(I+1) )      GO TO 10
37  28
38  29 C          IT'S NECESSARY TO SWAP TWO ADJACENT ELEMENTS OF <C_ARRAY>.
39  30 C          FOR EXAMPLE, <C_ARRAY(2)> MIGHT CONTAIN "EDWARD    " AND
40  31 C          <C_ARRAY(3)> MIGHT CONTAIN "BEVERLY   "; THEN THE NEXT
41  32 C          THREE STATEMENTS EXECUTE TO PERFORM THE SWAP.  AFTER THE
42  33 C          SWAP, <C_ARRAY(2)> WILL CONTAIN "BEVERLY   " AND
43  34 C          <C_ARRAY(3)> WILL CONTAIN "EDWARD    ".
44  35
45  36B                  TEMP = C_ARRAY(I)
46  > %
47  > % Return to the main program ...
48  > ENVIRONMENT @MAIN
49  :TEST_SORT10
50  > % ... and begin program execution.
51  > CONTINUE
52
```

```
53  Breakpoint trap at :TEST_SORT10:21
54  > %
55  > % Look at the first few elements of <NAMES> while the program
56  > %   continues to execute.
57  > TYPE I, NAMES(I) ; CONTINUE
58  1
59  "MIKE      "
60
61  Breakpoint trap at :TEST_SORT10:21
62  > TYPE I, NAMES(I) ; CONTINUE
63  2
64  "HENRIETTA "
65
66  Breakpoint trap at :TEST_SORT10:21
67  > TYPE I, NAMES(I) ; CONTINUE
68  3
69  "ENRICO    "
70
71  Breakpoint trap at :TEST_SORT10:21
72  > %
73  > % So far, so good.  Since <NAMES> seems OK, I'll clear this breakpoint
74  > %   and continue.
75  > CLEAR 21
76  Cleared at :TEST_SORT10:21
77  > CONTINUE
78
79  Breakpoint trap at :TEST_SORT10:24
80  > %
81  > % Go ahead and let SORT10 execute.
82  > CONTINUE
83
84  Breakpoint trap at :SORT10:22
85  %
86  % Now I'm in subroutine SORT10.
87  > TYPE N, N_LESS_1
88  8
89  7
90  > %
91  > % OK -- move into the   DO 20   and   DO 10   loops that sort <C_ARRAY>.
92  > CONTINUE
93
94  Breakpoint trap at :SORT10:36
95  > TYPE J, I, C_ARRAY(I), C_ARRAY(I+1)
96  1
97  1
98  "MIKE      "
99  "HENRIETTA "
100  > %
101  > % OK -- C_ARRAY(1) and C_ARRAY(2) have to swap their values.
102  > CONTINUE
```

```
103
104  Breakpoint trap at :SORT10:36
105  > TYPE J, I, C_ARRAY(I), C_ARRAY(I+1)
106  1
107  2
108  "MIKE      "
109  "ENRICO    "
110  > %
111  > % OK -- C_ARRAY(2) and C_ARRAY(3) have to swap their values.
112  > CONTINUE
113
114  Breakpoint trap at :SORT10:36
115  > TYPE J, I, C_ARRAY(I), C_ARRAY(I+1)
116  1
117  3
118  "MIKE      "
119  "LISA JEFFR"
120  > %
121  > % I've got a problem!  "MIKE      " is a valid name but "LISA JEFFR" is
122  > %    wrong.  Somehow "LISA      " and "JEFFREY   " have been incorrectly
123  > %    mixed together.  Now I'll display all the elements of <C_ARRAY> to
124  > %    see if there are any other such mixtures.
125  > TYPE C_ARRAY(1), C_ARRAY(2), C_ARRAY(3), C_ARRAY(4)
126  "HENRIETTA "
127  "ENRICO    "
128  "MIKE      "
129  "LISA JEFFR"
130  > TYPE C_ARRAY(5), C_ARRAY(6), C_ARRAY(7), C_ARRAY(8)
131  "EY   BETSY"
132  "     ALICE"
133  "     NORMA"
134  "N         "
135  > %
136  > % The last five elements of <C_ARRAY> are wrong.  I'll quit the debugger
137  > %    and take a close look at main program TEST_SORT10, which is the
138  > %    source of <C_ARRAY>.
139  > BYE
140
141  SWAT TERMINATED
```

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

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

16,19 Marll's instructions in lines 6 and 10 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.

25 Marll set two breakpoints with one statement.

31,45 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 72 through 77 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. The resulting TEST_SORT10.AU that points to the error follows.

```
 1  SWAT REVISION 02.00 ON 07/26/82 AT 11:35:48
 2  PROGRAM -- :UDD:MARLL:F77:TEST_SORT10
 3  > %
 4  > % I'll set a breakpoint where I can display ALL the elements of <NAMES>.
 5  > BREAKPOINT 21
 6  Set at :TEST_SORT10:21
 7  > LIST 15, 21
 8  15 C      PLACE THE 8 INDIVIDUAL FIRST NAMES INTO <NAMES> FROM THE SINGLE
 9  16 C         STRING <ALL_OF_THE_NAMES>.
10  17        DO 10 I = 1, 8
11  18              NAMES(I) = ALL_OF_THE_NAMES(10*I-9 : 10*I)  ! EXAMPLE: IF
12  19 C                        I = 2, THEN <ALL_OF_THE_NAMES(11:20)> IS
13  20 C                        'HENRIETTA ' AND <NAMES(2)> IS ALSO 'HENRIETTA '.
14  21B          10 CONTINUE
15  > %
16  > % Here we go!
17  > CONTINUE
18
19  Breakpoint trap at :TEST_SORT10:21
20  > TYPE I, NAMES(I) ; CONTINUE
21  1
22  "MIKE      "
23
24  Breakpoint trap at :TEST_SORT10:21
25  > TYPE I, NAMES(I) ; CONTINUE
26  2
27  "HENRIETTA "
28
29  Breakpoint trap at :TEST_SORT10:21
30  > TYPE I, NAMES(I) ; CONTINUE
31  3
32  "ENRICO    "
33
34  Breakpoint trap at :TEST_SORT10:21
35  > TYPE I, NAMES(I) ; CONTINUE
36  4
37  "LISA JEFFR"
38
39  Breakpoint trap at :TEST_SORT10:21
40  > TYPE I, NAMES(I) ; CONTINUE
41  5
42  "EY   BETSY"
43
44  Breakpoint trap at :TEST_SORT10:21
45  > TYPE I, NAMES(I) ; CONTINUE
46  6
47  "     ALICE"
48
49  Breakpoint trap at :TEST_SORT10:21
50  > TYPE I, NAMES(I) ; CONTINUE
51  7
52  "     NORMA"
53
```

```
54  Breakpoint trap at :TEST_SORT10:21
55  > TYPE I, NAMES(I)
56  8
57  "N         "
58  > %
59  > % The first three elements of <NAMES> are OK and I can't see any
60  > %    immediate reason for the error (the mixing) in the last five
61  > %    elements.  I'll investigate by going backwards and LISTing the
62  > %    CHARACTER string <ALL_OF_THE_NAMES>, from which <NAMES>
63  > %    obtains its elements.
64  > LIST 9, 13
65  9  C     THE NEXT TWO LINES HELP TO CONSTRUCT <ALL_OF_THE_NAMES>.
66  10 C000000001111111111222222222233333333334444444444555555555566666666667777
67  11 C23456789012345678901234567890123456789012345678901234567890123456789012
68  12        DATA  ALL_OF_THE_NAMES  / 'MIKE      HENRIETTA ENRICO    LISA
69  13       + JEFFREY   BETSY     ALICE     NORMAN    ' /
70  > %
71  > % Rather puzzling.  I can see that "LISAbbbbbb" (b = blank) is in
72  > %    lines 12 and 13.  The first five blanks of "LISAbbbbbb" come
73  > %    from line 12 and the last blank comes from line 13.  <NAMES(4)>
74  > %    is "LISAbJEFFR" with just one blank.  It looks like only the
75  > %    blank in "+ JEFFREY" of line 13 has arrived in the incorrect
76  > %    <NAMES(4)>.  In other words, the five blanks after "LISA" in
77  > %    line 12 have disappeared.  What's going on here?  I'm going to
78  > %    terminate SWAT and think of why the five blanks after "LISA"
79  > %    in line 12 have disappeared.
80  > %
81  > % However, before terminating SWAT I'll display <ALL_OF_THE_NAMES>.
82  > TYPE ALL OF THE NAMES
83  "MIKE      HENRIETTA ENRICO    LISA JEFFREY   BETSY     ALICE     NORMAN
     "
84  > %
85  > % This display also shows that the first five of the necessary six
86  > %    blank characters after "LISA" have disappeared.
87  > BYE
88
89  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:

```
M I K E □□□□□□
H E N R I E T T A □
E N R I C O □□□□
L I S A □J E F F R
E Y □□□B E T S Y
□□□□□A L I C E
□□□□□N O R M A
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 compilers are large and complicated programs. 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

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 an F77 program unit that calls a subroutine subprogram written in assembly language. And, a PL/I 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 PL/I 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.

Calling Conventions

The F77-generated code which implements the CALL statement or references to function subprograms observes the conventions of the Data General AOS Common Language Runtime Environment (CLRE). These conventions are also used by AOS FORTRAN 5, AOS DG/L, and AOS PL/I.

Under the AOS CLRE conventions, each language defines the addressing schemes used for its data types. But the essential elements of

- How arguments are passed to external procedures
- How external procedures are called
- How the caller's stack is left after return from external procedures

are common to the AOS CLRE languages.

The AOS CLRE convention provides the possibility for a given routine to be used in more than one language environment. Data General has taken advantage of the AOS CLRE to develop, for example, mathematical libraries and file system interface routines that can be called from routines written in any of the AOS CLRE languages.

The CLRE Convention

Here are the five principles of the AOS CLRE convention.

1. As the initial step in calling an external procedure, an

    LDA 2,.SP

    instruction is executed. This is done prior to any push instructions, so that the caller's AC2 becomes a "stack marker".

2. The addresses of the to-be-passed arguments are pushed onto the caller's stack in reverse order of their appearance in the argument list.

    Each CLRE language defines the ways its data types are stored and addressed, and so the nature of the addresses pushed will depend on the called routine's source language. F77's storage and addressing rules are given in the next section.

3. Once all the addresses of the arguments are pushed onto the stack, the external procedure is called via the ?RCALL mechanism.

4. Upon return from the called procedure:

    - All fixed-point accumulators contain the values they had prior to the ?RCALL, except AC3, which contains the frame pointer.
    - All floating-point accumulators are undefined.

5. All "pushed" argument addresses are effectively "popped" from the stack by a

    STA 2,.SP

    instruction that the calling program executes upon return from the procedure.

F77 Argument Addressing Conventions

Each pushed address is usually a 16-bit WORD address. The exceptions to this are:

- For CHARACTER variables and character constants, a 16-bit BYTE address is pushed.
- For arguments that have been declared EXTERNAL or INTRINSIC, the WORD address of the external reference is pushed.

If any argument on the CALL line is of type CHARACTER, extra arguments are on the stack. These arguments, known as *dope vectors*, inform the called routine of the actual size of the arguments. The dope vectors are built either at compile-time or runtime. The addresses of all required dope vectors are first pushed onto the stack, followed by the addresses of all of the user's arguments in reverse order.

(Users writing assembler subprograms to interface with F77 routines need to be aware of the existence of these dope vectors on the stack. However, the content and number of these vectors is determined by Data General and may change over time. User routines should not attempt to refer to or use the dope vectors in any way. Instead, the calling routine should use extra arguments to pass length information. The called routine can then obtain the length information via the appropriate argument address and be independent of the dope vectors, if any.)

All arguments passed to subprograms are passed by reference; that is, subprograms perform operations directly on the arguments, not local copies of them. To pass a variable argument by value, enclose it in parentheses in the argument list. This forces the compiler to treat the argument as an expression, and pass its value in a temporary.

Function Results

A function result will be returned in a temporary. The word address of the temporary will have been pushed on the stack by the calling routine, as if it were the first argument in a CALL statement. This address may need to be copied and converted to a byte pointer inside character functions.

Common Return Block

The AOS CLRE *common return block* is the fundamental data structure used for linkage between routines in the F77 runtime environment. The block is built on the stack of the calling routine. The block is used by subroutines, reached by a CALL statement, and by function subprograms, reached by a function reference. The block is constructed in two separate steps:

Step 1. The CALLING routine pushes onto the stack the addresses of the arguments to be passed.

Step 2. The CALLED routine, as its first instruction, executes a SAVE, to both push a return block onto the stack, and allocate its own stack frame, if needed, beyond the common return block.

Upon completion of the subprogram, the CALLED routine executes an RTN instruction, popping both the CALLED routine's stack frame and the common return block from the stack. This instruction then transfers control to the CALLING routine.

NOTE: There is no relationship between the common return block and FORTRAN COMMON storage. The term "common" here refers to the fact that the same return block format is used by other AOS languages, as well.

Figure 6-1 contains a general diagram of the common 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 0

This is a word or a byte pointer (depending upon the data type) that points to the first argument, which is number 0.

Pointer to Arg i

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

REMEMBER — USE THE PARAMETERS FROM AF77SYM.SR AND F77_FMAC.SR!

Old AC0

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

To access this entry in the return block, use the parameter offset FAC0.

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 FAC1.

Old AC2

This is the saved value of AC2 at the time of the call.

It contains the old value of .SP. Don't change this word, since it affects the stack upon return from the subprogram.

To access this entry in the return block, use the parameter offset FAC2.

Old FP

This is the caller's frame pointer.

To access this entry in the return block, use the parameter offset FOFP.

C | Return PC

These are the values of the carry bit and of the program counter. The RTN instruction restores these values.

To access this entry in the return block, use the parameter offset FRTN.

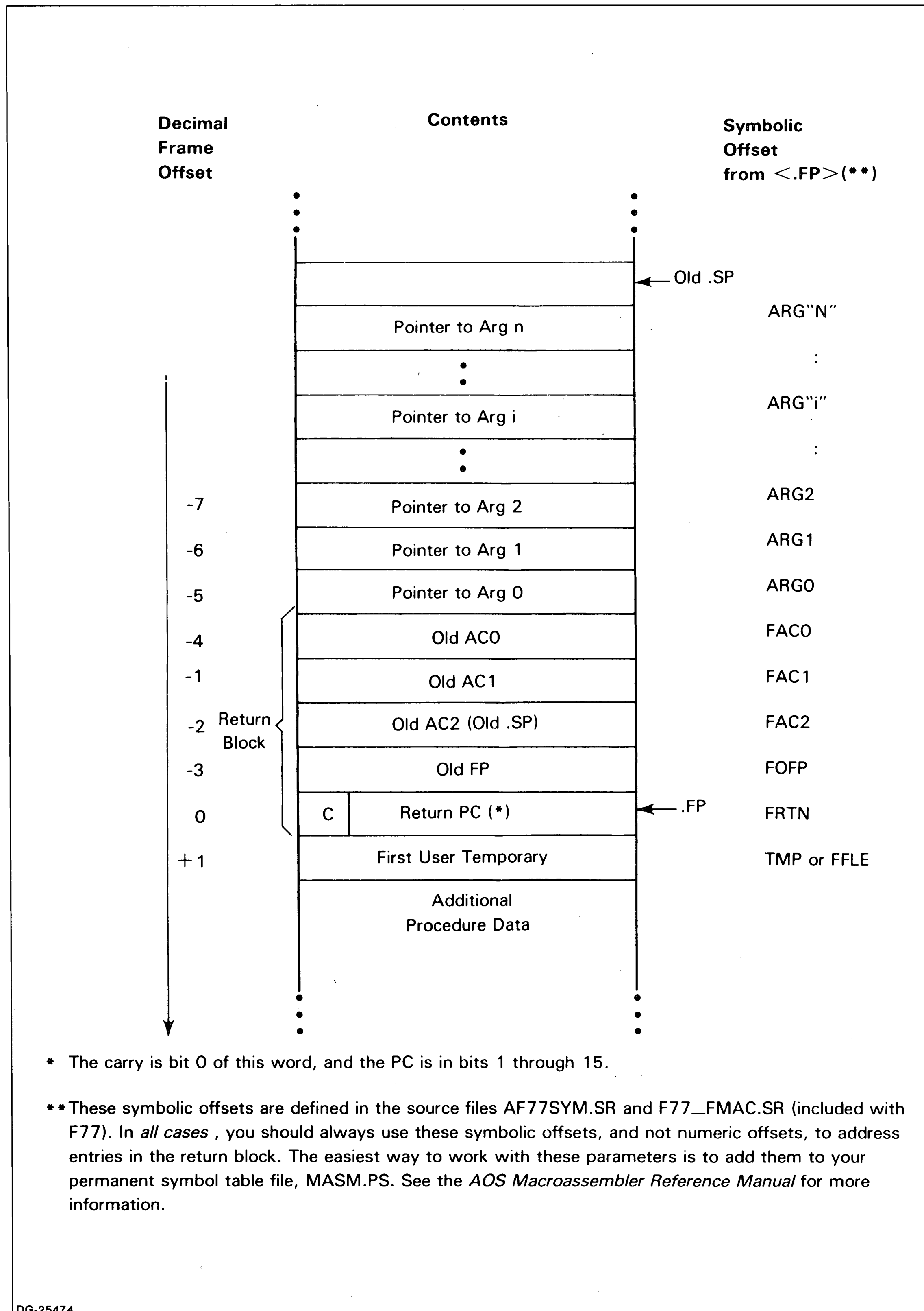


Figure 6-1. The Stack after Execution of a SAVE Instruction

```
Source file: TEST_TYP_SUB.F77
Compiled on 26-Oct-82 at 14:40:08 by AOS F77 Rev 2.10
Options: F77/INTEGER=2/LOGICAL=2/L=TEST_TYP_SUB.LS/CODE

    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 CLRE
    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



                          Code Listing

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


Line 1
              TEST_TYP_SUB:
00000  163710     SAVE     16                  ; 14.

Line 7
00002* 102050     FLDS     0,+16               ; 14. 25.2
00004  162250     FSTS     0,+4,3              ; 4. R1

Line 8
00006* 102050     FLDS     0,+14               ; 12. 16.8
00010  162250     FSTS     0,+6,3              ; 6. R2

Line 9
00012  060041     LEF      0,+41               ; 33.
00013  041410     STA      0,+10,3             ; 8. I1

Line 10
00014  166070     ELEF     1,+1550             ; 872.
00016  045411     STA      1,+11,3             ; 9. I2
```

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

```
Line 14
00017  030040    LDA     2,+40          ; 32.
00020  061412    LEF     0,+12,3        ; 10. SUM4
00021  103110    PSH     0,0
00022  061411    LEF     0,+11,3        ; 9. I2
00023  065410    LEF     1,+10,3        ; 8. I1
00024  107110    PSH     0,1
00025  061406    LEF     0,+6,3         ; 6. R2
00026  065404    LEF     1,+4,3         ; 4. R1
00027  107110    PSH     0,1
00030* 006014    ?RCALL                 ; TYP_SUB
00032  050040    STA     2,+40          ; 32.

Line 16
00033  060006    LEF     0,+6           ; 6.
00034  041414    STA     0,+14,3        ; 12. <temp>
00035  162470    ELEF    0,.+31         ; [66] $.312
00037  065414    LEF     1,+14,3        ; 12. <temp>
00040  107110    PSH     0,1
00041* 162070    ELEF    0,+0           ; 0. <dope>
00043* 166070    ELEF    1,+1           ; 1. <dope>
00045  107110    PSH     0,1
00046* 006000    JSR@    +0             ; 0. LW?EINIT
00047  050040    STA     2,+40          ; 32.
00050  060023    LEF     0,+23          ; 19.
00051  041414    STA     0,+14,3        ; 12. <temp>
00052  061414    LEF     0,+14,3        ; 12. <temp>
00053* 166070    ELEF    1,+2           ; 2.
00055  125120    MOVZL   1,1
00056  107110    PSH     0,1            ; THE OVERALL SUM IS
00057* 006000    JSR@    +0             ; 0. LW?SCH
00060  050040    STA     2,+40          ; 32.
00061  061412    LEF     0,+12,3        ; 10. SUM4
00062  103110    PSH     0,0
00063* 006000    JSR@    +0             ; 0. LW?SR4
00064  050040    STA     2,+40          ; 32.
00065* 006000    JSR@    +0             ; 0. LC?ETERM

Line 18
              $.312:
00066  126400    SUB     1,1
00067* 006000    JSR@    +0             ; 0. F.STOP
```

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

```
Source file: TYP_SUB.F77
Compiled on 26-Oct-82 at 14:40:40 by AOS F77 Rev 2.10
Options: F77/INTEGER=2/LOGICAL=2/L=TYP_SUB.LS/CODE

    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


                    Code Listing

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

Line 1
              TYP_SUB:
00000  163710     SAVE    10                ; 8.
00002  102400     SUB     0,0
00003  041403     STA     0,+3,3            ; 3. $.306

Line 3
00004  162050     FLDS    0,@-5,3           ; -5. REAL_1
00006  161050     FAMS    0,@-6,3           ; -6. REAL_2
00010  162250     FSTS    0,+4,3            ; 4. SUM_REALS

Line 5
00012  027771     LDA     1,@-7,3           ; -7. INT_1
00013  033770     LDA     2,@-10,3          ; -8. INT_2
00014  147000     ADD     2,1
00015  122450     FLAS    1,0
00016  162250     FSTS    0,+6,3            ; 6. SUM_INTS

Line 7
00020  162050     FLDS    0,+6,3            ; 6. SUM_INTS
00022  161050     FAMS    0,+4,3            ; 4. SUM_REALS
00024  162250     FSTS    0,@-11,3          ; -9. OVERALL

Line 9
00026  127710     RTN
```

Figure 6-3. A Listing of TYP_SUB.F77 and Its Generated Code

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. Observe that the compiler has generated code that places this *fourth* argument on the stack *after* it has placed the *fifth* argument there.

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 AF77SYM.SR, F77_FMAC.SR, and LITMACS.SR. This section describes the use of the FORTRAN CALL macro set contained in the first two of these files. These macros are

TITLE

S?ATTR

DEFARGS

DEFTMPS

DEF

FENTRY

FCALL

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.

If your routine will call another routine, you must indicate this by using the **S?ATTR** macro. The calling sequence for S?ATTR is

S?ATTR 7[NO]7 FCALL ! 7[NO]7 RCALL

Here, the symbol "!" carries the meaning of "inclusive OR". Specifying S?ATTR **FCALL** indicates that your routine will call another F77 routine. This specification is necessary to reserve extra space on the stack for "bookkeeping." Specifying S?ATTR RCALL indicates that your routine will use ?RCALL directly (not via the FCALL macro). The RCALL attribute must be set to have the FENTRY macro reserve two words of stack space for the ?RCALL manager. Under AOS, FCALL will actually be performed by an ?RCALL, but you need not specify S?ATTR RCALL if you have specified S?ATTR FCALL. The optional NO argument to the S?ATTR macro is for documentation and completeness. If S?ATTR is not used, the effect is the same as specifying

S?ATTR NO FCALL ! NO RCALL .

DEFARGS immediately follows TITLE, and S?ATTR if used. 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 DEFs (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 one 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 (.FP), which is 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 SAVE 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 — they will be restored as required when your routine completes. If your routine calls out, and uses the CLRE calling convention, you will need to set up AC2 as the stack marker and push argument addresses on the stack, as described earlier in this chapter.

FRET returns control to the calling routine. This macro generates an RTN 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 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 AF77SYM.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 AF77SYM.SR and F77_FMAC.SR; and it uses FORTRAN 77 CALL macros from these files.

NOTE: The first pages of both versions of RUNTM.SR are identical.

```
      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) + ALOG(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
```

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 AOS CLRE 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:
;
;       1.  Be sure a MASM.PS file exists that contains the parameter
;           offsets and macros for interfacing to F77.  To create one,
;           give the CLI command
;
;               X MASM/8/S/N EBID SYSID PARU AF77SYM PARF77 F77_FMAC LITMACS
;
;       2.  Give the CLI command
;
;               X MASM/8 RUNTM
;
; To link this routine with F77 programs:
;
;           F77LINK main-program-name RUNTM
;
```

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,
        PCKTLEN = ?GRLTH
                                        ;  and calculate the maximum number of
                                        ;  words on the stack that will
                                        ;  be needed to build the packet.

RUNTM:                                  ; Routine entry:
        SAVE    PCKTLEN                 ; Save the state, and enough stack
                                        ;  space for the packet, and put
                                        ;  AC3 <== my FRAME POINTER.

                                        ; Make system call:
        ADC    0,0                      ; AC0 <== -1 to indicate this process
        ELEF    2,PACKET,3              ; AC2 <== address of packet
        ?RUNTM                          ; Get runtime stats
        JMP     RUNTERR                 ; Error on system call
                                        ; Good return:
                                        ;  move values into caller's arguments
        FLMD    0,PACKET+?GRRH,3        ; Get elapsed time in seconds
        FFMD    0,@ARG0,3               ; Put into 0th argument via pointer
        FLMD    0,PACKET+?GRCH,3        ; Get CPU time in milliseconds
        FFMD    0,@ARG1,3               ; Put into 1st argument via pointer
        FLMD    0,PACKET+?GRIH,3        ; Get I/O blocks read or written
        FFMD    0,@ARG2,3               ; Put into 2nd argument via pointer
        FLMD    0,PACKET+?GRPH,3        ; Get # page/milliseconds
        FFMD    0,@ARG3,3               ; Put into 3rd argument via pointer

        SUB     0,0                     ; Zero AC0 to show good return

RUNTERR:                                ; Enter here if error.  Common path
                                        ;  for setting error return variable:
        FLAS    0,0                     ; Float the value in AC0 (into FPAC0)
                                        ;  as a single precision number.
        FFMD    0,@ARG4,3               ; Put (FPAC0) into 4th argument
                                        ;  as a 4-byte integer.

        RTN                             ; Go back to F77 caller.

        .END
```

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 AOS CLRE 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:
;
;        1.  Be sure a MASM.PS file exists that contains the parameter
;            offsets and macros for interfacing to F77.  To create one,
;            give the CLI command
;
;                X MASM/8/S/N EBID SYSID PARU AF77SYM PARF77 F77_FMAC LITMACS
;
;        2.  Give the CLI command
;
;                X MASM/8 RUNTM
;
; To link this routine with F77 programs:
;
;            F77LINK main-program-name RUNTM
;
```

Figure 6-6. Subroutine RUNTM.SR, Version 2 (continues)

Licensed Material-Property of Data General Corporation

```
; ****************************   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___BLKS                 ; (Use two underscores since these are  @FMAC
  DEF IP___MS                    ;   arguments to a macro that removes   @FMAC
                                ;   one of them: "IP___MS" becomes
                                ;   "IP__MS" as desired.)
  DEF IER                       ;                                      @FMAC

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

        ADC     0,0             ; AC0 <== -1 to indicate this process
        ELEF    2,PACKET,3      ; AC2 <== address of packet
        ?RUNTM                  ; Get runtime stats
        JMP     RUNTERR         ; Error on system call
                                ; Good return:
                                 ;  move values into caller's arguments
        FLMD  0,PACKET+?GRRH,3   ; Get elapsed time in seconds
        FFMD  0,@IELAPSED,3      ; Put into IELAPSED via address on stack
        FLMD  0,PACKET+?GRCH,3   ; Get CPU time in milliseconds
        FFMD  0,@ICPU,3          ; Put into ICPU via address on stack
        FLMD  0,PACKET+?GRIH,3   ; Get I/O blocks read or written
        FFMD  0,@IIO__BLKS,3      ; Put into IIO__BLKS via address on stack
        FLMD  0,PACKET+?GRPH,3   ; Get # page/milliseconds
        FFMD  0,@IP__MS,3         ; Put into IP__MS via address on stack

        SUB    0,0               ; Zero AC0 to show good return

RUNTERR:                        ; Enter here if error.  Common path
                                ;  for setting error return variable:
        FLAS  0,0               ; Float the value in AC0 (into FPAC0)
                                ;  as a single precision number.
        FFMD  0,@IER,3          ; Put (FPAC0) into IER via argument
                                ;  address, as a 4-byte integer.

        FRET                    ; Go back to F77 caller.               @FMAC

        END                     ;                                      @FMAC
```

Figure 6-6. Subroutine RUNTM.SR, Version 2 (concluded)

The following commands assemble RUNTM.SR (in either Figure 6-5 or Figure 6-6), compile TEST_RUNTM.F77, and create TEST_RUNTM.PR.

```
X MASM/8/O=RUNTM.OB RUNTM
F77 TEST_RUNTM
F77LINK TEST_RUNTM RUNTM
```

Let's look at the results of executing TEST_RUNTM.PR:

```
) X TEST_RUNTM ↲
```

PROCESS ELAPSED TIME IN SECONDS:	*4*
PROCESS CPU TIME IN MILLISECONDS:	*2916*
NUMBER OF I/O BLOCKS:	*20*
NUMBER OF PAGE/MILLISECONDS:	*98946*

*** *END OF JOB* ***

The results usually vary slightly each time TEST_RUNTM.PR executes.

Macro F77_FMAC.SR

Beginning with Revision 2.10 of AOS F77, the file F77_FMAC.SR is supplied with F77 instead of FMAC.SR. The files are different mainly in the content of their ISA.NORM and ISA.ERR macros.

The ISA.NORM and ISA.ERR macros have been changed because of a side effect caused by the presence of character datatype 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 referred to. 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 pushed (arguments are pushed in reverse order) and would be as useless as extra leading zeros when writing numbers.

The AOS CLRE languages (including FORTRAN 5 and FORTRAN 77) mark the stack (by loading the stack pointer into AC2) before pushing the addresses of arguments. The ISA.NORM and ISA.ERR macros with FMAC.SR assumed that the last argument in the list (the one whose address is first 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.

The new version of FMAC.SR (called F77_FMAC.SR) has 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 error code that is in AC0. The "*" symbol is a placeholder, which allows you to specify a nondefault *ier_pos* and to supply the error code 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 languages that also use the Common Language Runtime Environment (CLRE). The languages using the CLRE are DG/L™, FORTRAN 5, FORTRAN 77, 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 DG/L 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 directly support any data type that corresponds to the DG/L language's POINTER.
- The DG/L language 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 CLRE 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.

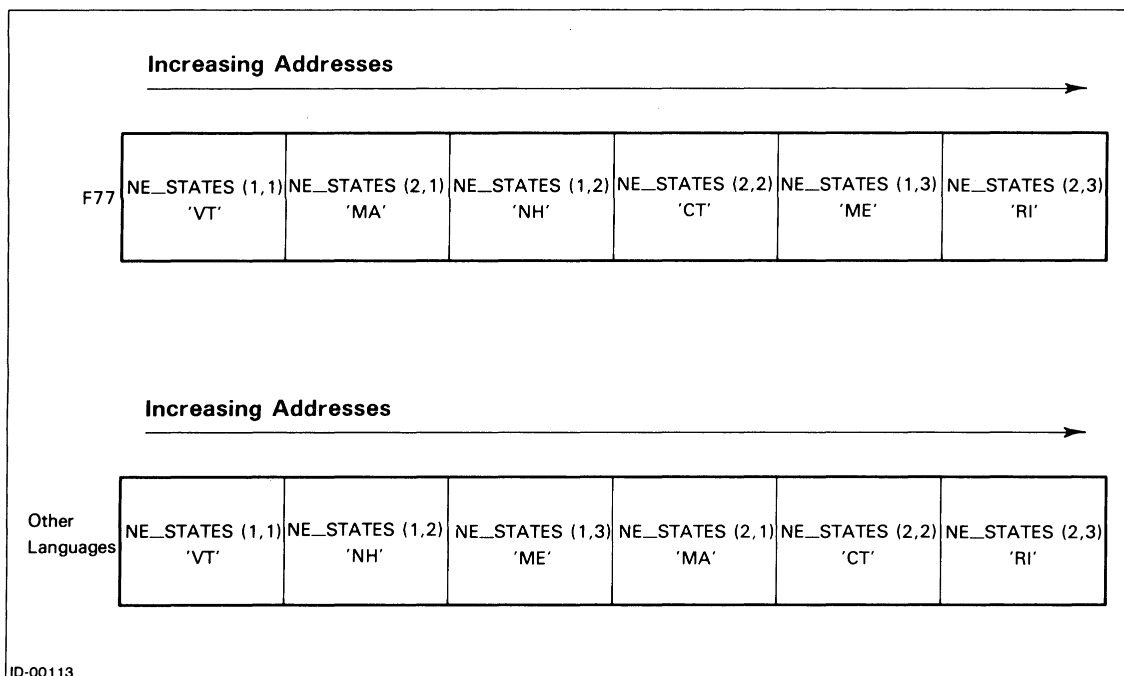


Figure 6-7. An Example of Storage of Multidimension Arrays by F77 and Other Languages

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 other than 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 is case-insensitive because it maps 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.

Interlanguage Conflicts

Each CLRE language 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.

```
Source file: GENERAL.F77
Compiled on 10-Nov-82 at 13:36:58 by AOS F77 Rev 2.10
Options: F77/INTEGER=2/LOGICAL=2/L=GENERAL.LS

    1            SUBROUTINE GENERAL (REAL_ARRAY, REAL_SIZE, INT_ARRAY, INT_SIZE,
    2         +     ANGLE, LARGEST_REAL, SMALLEST_INT, SINE_ANGLE, ERROR)
    3
    4            INTEGER*2 REAL_SIZE
    5            REAL*4 REAL_ARRAY(REAL_SIZE)
    6            INTEGER*2 INT_SIZE
    7            INTEGER*2 INT_ARRAY(INT_SIZE)
    8            REAL*4 ANGLE
    9            REAL*4 LARGEST_REAL
   10            INTEGER*2 SMALLEST_INT
   11            REAL*4 SINE_ANGLE
   12            INTEGER*2 ERROR
   13
   14            ERROR = 0         ! Assume there's no error in the array sizes.
   15    C                              But, check the sizes and RETURN with the
   16    C                              error variable set if there is an error.
   17            IF ( REAL_SIZE .LT. 1  .OR.  INT_SIZE .LT. 1 ) THEN
   18                    ERROR = 1
   19                    RETURN
   20            ENDIF
   21
   22    C       Find the largest element in <REAL_ARRAY> and place it in
   23    C          <LARGEST_REAL>.
   24            LARGEST_REAL = REAL_ARRAY(1)
   25            DO 10 I = 2, REAL_SIZE
   26                    IF ( REAL_ARRAY(I) .GT. LARGEST_REAL )
   27         1               LARGEST_REAL = REAL_ARRAY(I)
   28       10   CONTINUE
   29
   30    C       Find the smallest element in <INT_ARRAY> and place it in
   31    C          <SMALLEST_INT>.
   32            SMALLEST_INT = INT_ARRAY(1)
   33            DO 20 I = 2, INT_SIZE
   34                    IF ( INT_ARRAY(I) .LT. SMALLEST_INT )
   35         1               SMALLEST_INT = INT_ARRAY(I)
   36       20   CONTINUE
   37
   38    C       Compute the sine of <ANGLE> after converting <ANGLE> from degrees
   39    C          to radians.
   40            SINE_ANGLE = SIN(3.141593*ANGLE/180.0) ! PI radians = 180 degrees.
   41
   42    C       Done!
   43            RETURN
   44            END
```

Figure 6-8. Subroutine Subprogram GENERAL.F77

Subroutine GENERAL.F77 exists so that the other CLRE languages can call it to process their data. You will soon see sample programs that call GENERAL; they are written in the FORTRAN 5, DG/L, and PL/I languages (as well as in 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.

```
Source file: TEST_GENERAL.F77
Compiled on 10-Nov-82 at 13:40:12 by AOS F77 Rev 2.10
Options: F77/INTEGER=2/LOGICAL=2/L=TEST_GENERAL.LS

    1           PROGRAM TEST_GENERAL   ! to test subroutine GENERAL
    2
    3           REAL*4 REALS(10)   /  3.40,  8.61, -6.00,  8.94,  4.18,
    4        1                        7.56, -9.57,  0.00, -1.24,  0.52 /
    5           INTEGER*2 R_SIZE   / 10 /
    6
    7           INTEGER*2 INTS(5)  /   386, -2846,  3091,   -33,  5104 /
    8           INTEGER*2 I_SIZE   /     5 /
    9
   10           REAL*4 ANGLE       /  30.0 /
   11
   12           REAL*4 BIG_REAL
   13           INTEGER*2 SMALL_INT
   14           REAL*4 SINE_ANGLE
   15           INTEGER*2 IER
   16
   17   C       Here we go ...
   18
   19           CALL GENERAL (REALS, R_SIZE, INTS, I_SIZE, ANGLE,
   20        1     BIG_REAL, SMALL_INT, SINE_ANGLE, IER)
   21
   22           IF ( IER .EQ. 0 ) THEN
   23
   24                   PRINT *
   25                   PRINT *, 'THE LARGEST REAL*4 NUMBER IS:         ', BIG_REAL
   26                   PRINT *, 'THE SMALLEST INTEGER*2 NUMBER IS:     ', SMALL_INT
   27                   PRINT *, 'THE SINE OF ', ANGLE, ' DEGREES IS: ', SINE_ANGLE
   28                   PRINT *
   29           ELSE
   30                   PRINT *
   31                   PRINT *, 'ERROR OCCURRED IN SUBROUTINE GENERAL.'
   32                   PRINT *
   33           ENDIF
   34
   35           STOP
   36           END
```

Figure 6-9. Main Program TEST_GENERAL.F77

Note that all the variables in GENERAL.F77 and TEST_GENERAL.F77 are either REAL*4 or INTEGER*2. Each of the CLRE 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_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
```

High-Level Languages and F77 Subroutines

The DG/L, FORTRAN 5, FORTRAN 77, and PL/I languages follow the CLRE. 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.
- An explanation of any peculiarities of the language that affect F77 subroutines.

FORTRAN 5 and F77

This section lists F77 data types and their FORTRAN 5 correspondents. It also shows the FORTRAN 5 program, TEST_GENERAL.FR, that calls subroutine GENERAL.F77.

F77 and FORTRAN 5 Data Types

F77	FORTRAN 56
INTEGER*2	INTEGER
INTEGER*4	None
REAL*4	REAL
REAL*8 and DOUBLE PRECISION	DOUBLE PRECISION
COMPLEX	COMPLEX
COMPLEX*16 and DOUBLE PRECISION COMPLEX	DOUBLE PRECISION COMPLEX
LOGICAL*2	LOGICAL
LOGICAL*4	None
CHARACTER*N ("N" is a constant.)	None

Sample Program

Program TEST_GENERAL.FR calls subroutine GENERAL. This program's listing is shown in Figure 6-10.

```
AOS FORTRAN 5, Version 6.11 -- Wednesday, November 10, 1982  2:01:33 PM

                        TEST_GENERAL.FR

 1:   C       PROGRAM TEST_GENERAL     to test subroutine GENERAL
 2:
 3:           REAL REALS(10)   /  3.40,  8.61, -6.00,  8.94,  4.18,
 4:        1                      7.56, -9.57,  0.00, -1.24,  0.52 /
 5:           INTEGER R_SIZE   / 10 /
 6:
 7:           INTEGER INTS(5)  /   386, -2846,  3091,   -33,  5104 /
 8:           INTEGER I_SIZE   /     5 /
 9:
10:           REAL ANGLE       /  30.0 /
11:
12:           REAL BIG_REAL
13:           INTEGER SMALL_INT
14:           REAL SINE_ANGLE
15:           INTEGER IER
16:
17:   C       Here we go ...
18:
19:           CALL GENERAL (REALS, R_SIZE, INTS, I_SIZE, ANGLE,
20:        1     BIG_REAL, SMALL_INT, SINE_ANGLE, IER)
21:
22:           IF ( IER .NE. 0 ) GO TO 10
23:
24:           TYPE
25:           TYPE 'THE LARGEST REAL NUMBER IS:        ', BIG_REAL
26:           TYPE 'THE SMALLEST INTEGER NUMBER IS:    ', SMALL_INT
27:           TYPE 'THE SINE OF ', ANGLE, ' DEGREES IS: ', SINE_ANGLE
28:           TYPE
29:           GO TO 20
30:
31:      10   TYPE
32:           TYPE 'ERROR OCCURRED IN SUBROUTINE GENERAL.'
33:           TYPE
34:
35:      20   STOP
36:           END
```

Figure 6-10. Program TEST_GENERAL.FR

Assume that you have the directory with the FORTRAN 5 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.FR.

```
F5 TEST_GENERAL
F5LD TEST_GENERAL GENERAL F77MATH2.LB
XEQ TEST_GENERAL
```

The output from the execution of TEST_GENERAL.PR is

THE LARGEST REAL NUMBER IS: 8.94000
THE SMALLEST INTEGER NUMBER IS: -2846
THE SINE OF 30.0000 DEGREES IS: .500000

STOP

DG/L and F77 Languages

This section lists F77 data types and their DG/L correspondents. It also shows the DG/L program, TEST_GENERAL.DG, that calls subroutine GENERAL.F77.

F77 and DG/L Data Types

F77	DG/L
INTEGER*2	INTEGER(1)
INTEGER*4	INTEGER(2)
REAL*4	REAL(2)
REAL*8 and DOUBLE PRECISION	REAL(4)
COMPLEX	None
COMPLEX*16 and DOUBLE PRECISION COMPLEX	None
LOGICAL*2	None — But, a DG/L BOOLEAN variable whose value is 0 is the same as an F77 LOGICAL*2 variable whose value is .FALSE.. The DG/L language represents "TRUE" with a binary 1 in a BOOLEAN variable; F77 represents ".TRUE." with a -1 (all bits on) in a LOGICAL*2 variable.
LOGICAL*4	None
CHARACTER*1	STRING(1)
CHARACTER*N ("N" is a constant.)	STRING(N)
CHARACTER*(*)	None

Sample Program

Program TEST_GENERAL.DG calls subroutine GENERAL. This program's listing is shown in Figure 6-11.

```
      AOS DG/L, VERSION 2.10  --  WEDNESDAY, NOVEMBER 10, 1982  3:11 PM

                DGL/L=TEST_GENERAL.LS  TEST_GENERAL.DG

  1     BEGIN
  2
  3     COMMENT      SAMPLE PROGRAM   TEST_GENERAL   TO TEST SUBROUTINE
  4          <GENERAL>;
  5
  6     EXTERNAL PROCEDURE GENERAL;
  7     EXTERNAL STRING PROCEDURE GETCOUTPUT;
  8
  9          REAL(2) ARRAY           REALS [1:10];
 10          INTEGER(1)              R_SIZE;
 11
 12          INTEGER(1) ARRAY        INTS [1:5];
 13          INTEGER(1)              I_SIZE;
 14
 15          REAL(2)                 ANGLE;
 16
 17          REAL(2)                 BIG_REAL;
 18          INTEGER(1)              SMALL_INT;
 19          REAL(2)                 SINE_ANGLE;
 20          INTEGER(1)              IER;
 21
 22
 23          REALS[1]   :=  3.40;
 24          REALS[2]   :=  8.61;
 25          REALS[3]   := -6.00;
 26          REALS[4]   :=  8.94;
 27          REALS[5]   :=  4.18;
 28          REALS[6]   :=  7.56;
 29          REALS[7]   := -9.57;
 30          REALS[8]   :=  0.00;
 31          REALS[9]   := -1.24;
 32          REALS[10]  :=  0.52;
 33          R_SIZE     := 10    ;
 34
 35          INTS[1]    :=    386;
 36          INTS[2]    :=  -2846;
 37          INTS[3]    :=   3091;
 38          INTS[4]    :=    -33;
 39          INTS[5]    :=   5104;
 40          I_SIZE     :=      5;
 41
 42          ANGLE      := 30.0;
 43
 44          OPEN (1,(GETCOUTPUT));
 45
 46     COMMENT      Here we go ... ;
 47
 48          GENERAL (REALS, R_SIZE, INTS, I_SIZE, ANGLE,
 49                   BIG_REAL, SMALL_INT, SINE_ANGLE, IER);
 50
```

Figure 6-11. Program TEST_GENERAL.DG (continues)

```
51          IF IER = 0 THEN
52                BEGIN
53                WRITE (1, "<NL>");
54                WRITE (1, "THE LARGEST REAL(2) NUMBER IS:       ",
55                          BIG_REAL, "<NL>");
56                WRITE (1, "THE SMALLEST INTEGER(1) NUMBER IS:  ",
57                          SMALL_INT, "<NL>");
58                WRITE (1, "THE SINE OF ", ANGLE, " DEGREES IS: ",
59                          SINE_ANGLE, "<NL>");
60                WRITE (1, "<NL>");
61                END
62          ELSE
63                BEGIN
64                WRITE (1, "<NL>");
65                WRITE (1, "ERROR OCCURRED IN SUBROUTINE GENERAL.<NL>");
66                WRITE (1, "<NL>");
67                END;
68
69    END;
```

Figure 6-11. Program TEST_GENERAL.DG (concluded)

Assume that you have the directory with the DG/L 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.DG.

```
X DGL TEST_GENERAL
X LINK TEST_GENERAL GENERAL F77MATH2.LB [DGLIB]
X TEST_GENERAL
```

The output from the execution of TEST_GENERAL.PR is

THE LARGEST REAL(2) NUMBER IS:	*8.94*
THE SMALLEST INTEGER(1) NUMBER IS:	*-2846*
THE SINE OF 30. DEGREES IS:	*.5*

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*4	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 6-12.

```
SOURCE FILE: TEST_GENERAL.PL1
COMPILED ON 11/10/82   AT 15:41:28   BY PL/I REV  2.31
OPTIONS: PL1/L=TEST_GENERAL.LS,TEST_GENERAL

    1   TEST_GENERAL:
    2           PROCEDURE;
    3           DECLARE REALS(10)       FLOAT BINARY(15) STATIC INIT (
    4                                        3.40,  8.61, -6.00,  8.94, 4.18,
    5                                        7.56, -9.57,  0.00, -1.24, 0.52 ),
    6
    7                   R_SIZE           FIXED BINARY(15) STATIC INIT(10),
    8
    9                   INTS(5)         FIXED BINARY(15) STATIC INIT (
   10                                        386, -2846, 3091, -33, 5104 ),
   11
   12                   I_SIZE           FIXED BINARY(15) STATIC INIT(5),
   13
   14                   ANGLE           FLOAT BINARY(15) STATIC INIT(30),
   15
   16                   BIG_REAL         FLOAT BINARY(15),
   17                   SMALL_INT        FIXED BINARY(15),
   18                   SINE_ANGLE       FLOAT BINARY(15),
   19                   IER             FIXED BINARY(15),
   20
   21                   GENERAL         ENTRY((10) FLOAT BIN(15), /* REALS     */
   22                                              FIXED BIN(15), /* R_SIZE    */
   23                                          (5) FIXED BIN(15), /* INTS      */
   24                                              FIXED BIN(15), /* I_SIZE    */
   25                                              FLOAT BIN(15), /* ANGLE     */
   26                                              FLOAT BIN(15), /* BIG_REAL  */
   27                                              FIXED BIN(15), /* SMALL_INT*/
   28                                              FLOAT BIN(15), /*SINE_ANGLE*/
   29                                              FIXED BIN(15)  ),   /* IER */
   30
   31                   @OUTPUT         FILE;
   32
   33           OPEN FILE(@OUTPUT) STREAM OUTPUT PRINT;
   34
   35           /* Here we go ... */
   36
   37           CALL GENERAL( REALS, R_SIZE, INTS, I_SIZE, ANGLE,
   38                         BIG_REAL, SMALL_INT, SINE_ANGLE, IER);
   39
```

Figure 6-12. Program TEST_GENERAL.PL1 (continues)

```
40          IF IER = 0 THEN
41                  DO;
42                          PUT FILE(@OUTPUT) SKIP LIST (" ");
43                          PUT FILE(@OUTPUT) SKIP EDIT(
44                                  "THE LARGEST REAL*4 NUMBER IS:      ",
45                                  BIG_REAL ) (A, F(5,2));
46                          PUT FILE(@OUTPUT) SKIP EDIT(
47                                  "THE SMALLEST INTEGER*2 NUMBER IS: ",
48                                  SMALL_INT ) (A, F(5));
49                          PUT FILE(@OUTPUT) SKIP EDIT(
50                                  "THE SINE OF ", ANGLE, " DEGREES IS: ",
51                                  SINE_ANGLE ) (A, F(5,1), A, F(7,4));
52                          PUT FILE(@OUTPUT) SKIP LIST (" ");
53                  END;
54                  ELSE
55                  DO;
56                          PUT FILE(@OUTPUT) SKIP LIST (" ");
57                          PUT FILE(@OUTPUT) SKIP LIST (
58                                  "ERROR OCCURRED IN SUBROUTINE GENERAL.");
59                          PUT FILE(@OUTPUT) SKIP LIST (" ");
60
61                  END;
62
63          STOP;
64
65          END;    /* OF PROGRAM TEST_GENERAL */
```

Figure 6-12. 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 F77MATH2.LB
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
- Reducing Memory and Disk Usage by Program Files

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 that is slightly more efficient than /DOTRIP=0. Be certain that the program logic will work correctly with this switch before using 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 forces uninitialized variables to 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, you improve compilation and execution speed by running on an idle system with lots of physical memory and a large working set.

Third, scan each Release Notice for hints. Also, your Data General Systems Engineer has access to the two following documents:

- The Systems Engineering NewsLetter (SENL).
- The FORTRAN Product Support Manual.

Ask him or her about the latest F77 programming suggestions that appear in these publications.

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.	2.94
Read the file as a fixed file into a character variable using the data descriptor "A100" for each record.	3.25
Read the file as a fixed file into a character variable with unformatted I/O for each record.	5.21
Read the file as a dynamic file into a character variable with unformatted I/O for each record and with the default BLOCKSIZE (512).	6.52
Read the file as a dynamic file into a character variable with unformatted I/O for each record and with a BLOCKSIZE value of 2048.	6.91

NOTE: These numbers reflect operation with a particular ECLIPSE® computer, operating system, peripherals, and revision of F77. Use them as guidelines 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 includes a form-feed character (whose octal value is <014>). When you print the file via a QPRINT command, XLPT.PR (as part of AOS) 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 (AOS) Operator's Guide* and the *Command Line Interpreter (CLI) User's Manual (AOS & AOS/VS).*

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 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

Frank 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 address labels, and MEMBERS.CARDS — for printing 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
```

Figure 7-2. Program PRINT_LABELS

Frank 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 Frank 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:FRANK:MEMBERS.LABELS] ? ↵

Command ? BYE ↵

FCU terminating ...

Frank verifies that the VFU specifications file exists with the CLI command

```
FILESTATUS/UDA MEMBERS.LABELS
```

AOS responds with

MEMBERS.LABELS UDA

Frank 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:FRANK: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).
- Frank wants printing to begin on the second line of the form.
- Frank 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
```

Figure 7-4. Program PRINT_CARDS

Frank 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 Frank 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:FRANK:MEMBERS.CARDS] ?  ↓

Command ?  BYE ↓

FCU terminating ...
```

Frank 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:FRANK: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 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.

Reducing Memory and Disk Usage of Program Files

This section explains a way to reduce your program files' runtime memory space and disk space.

Consider the following CLI dialog.

```
) CREATE/I SMALL_LARGE.F77 ↲
))      PROGRAM SMALL_LARGE ! SMALL PROGRAM, LARGE MEMORY/DISK USAGE ↲
))      SUM = 2.0 + 3.0 ↲
))      PRINT *, 'SUM IS ', SUM, '<NL>' ↲
))      STOP ↲
))      END ↲
))) ↲
) F77 SMALL_LARGE; F77LINK SMALL_LARGE; F/LEN SMALL_LARGE.<F77,PR> ↲
SMALL_LARGE.F77     124         SMALL_LARGE.PR   553
```

You can see that this small source program requires very little disk space, yet the resulting .PR file is much larger. When SMALL_LARGE.PR executes, it requires a full 32K words of memory. It may well slow the execution of other programs because of memory contention. Is there any way to reduce SMALL_LARGE.PR? The answer is "yes"; continue reading for the details.

Link — A Closer Look

F77LINK.CLI invokes LINK.PR to create a .PR file from at least one .OB file and various .LB files. By default, Link creates a .PR file that requires 65,536 bytes of memory at runtime and an identical amount of disk space. Link places a program's shared area at the end of a 32K-word block. Consequently, the program's unshared area of the 32K words is often much larger than it has to be.

See Figure 7-5 for a simplified view of a .PR file that results from a .F77 file. SMALL_LARGE.PR is an example; recall that it is created without special F77LINK switches that can alter the addresses in the figure. The file exists this way on disk; its exact image is in memory immediately after it is copied there. Symbols such as NTOP in Figure 7-5 appear in a Link-created load map file if you give the F77LINK command with the /MAP switch.

The /KTOP=n Link Switch

Link uses the value of its /KTOP=n switch to place the shared portion of the program in the .PR file. n is the memory page number where the shared area ends. (A page of memory has 1024=2000K words.) By default, n is 32, and consequently there is typically much unusued memory between the bottom of the shared area (location SBOT) and the top of the stack. In Figure 7-5, /KTOP=32.

If /KTOP is less than 32 then, with all other things being equal:

- The shared area moves down, along with the value of SBOT.
- The size of the shared area remains the same.
- The overall size of the .PR file becomes smaller, both on the disk and then later in memory.
- The unused portion of the unshared area becomes smaller.
- The area from NMAX down to location 0 remains unchanged in size.

Thus, one key to producing smaller .PR files is the proper use of the /KTOP=n switch in your F77LINK command. F77LINK passes this switch intact to Link.

An Example of Reducing a .PR File

Figure 7-6 contains a listing file from a sample program, MEMPAGE.F77. The program uses the ISYS function (from Chapter 3) to make a ?PSTAT call to get process statistics, and then displays some of them. Chapter 3 also explains the creation of %INCLUDE files such as MEMPAGE_SYMBOLS.F77.IN.

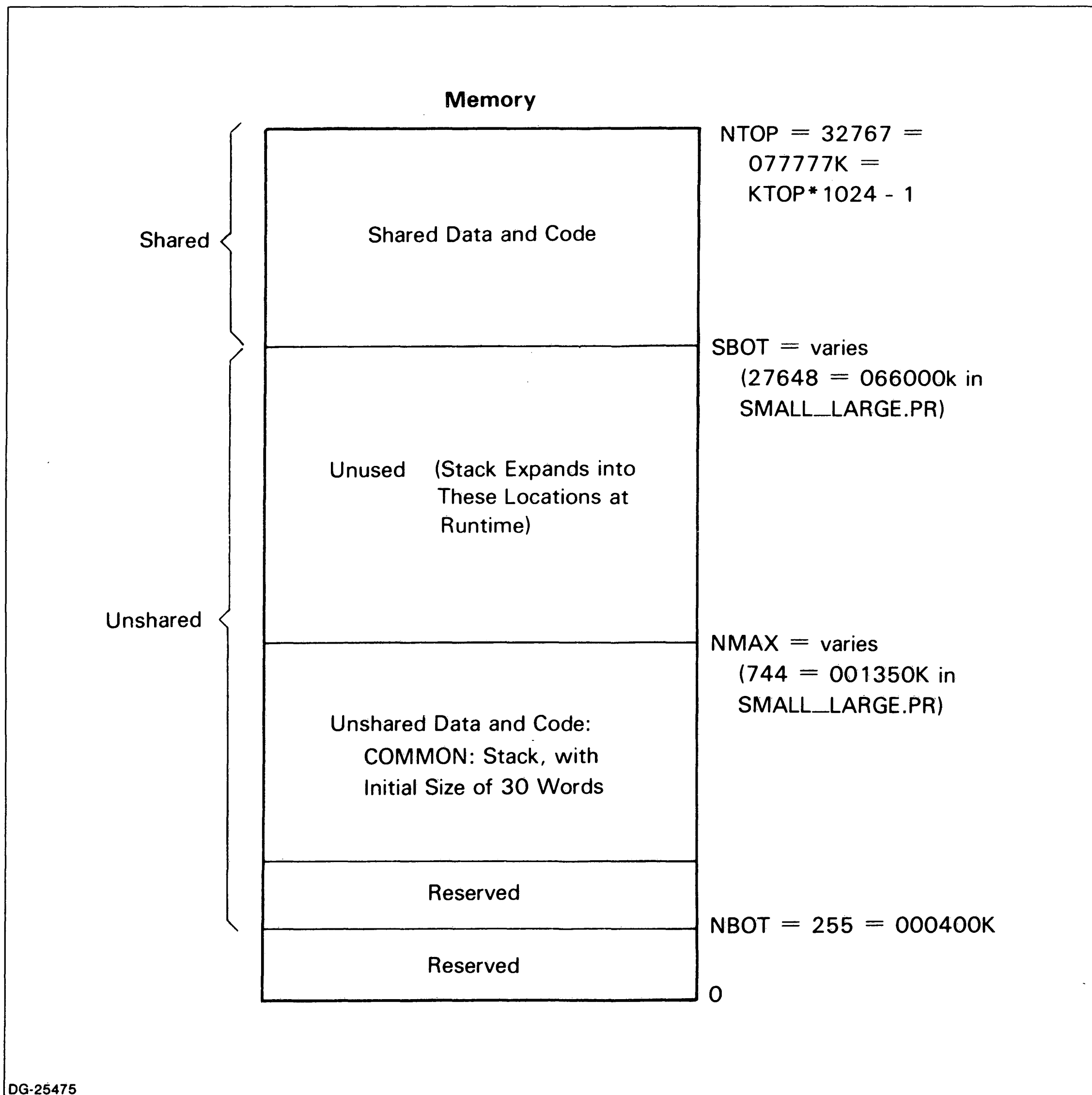


Figure 7-5. A Memory Model for an F77 Program File

```
Source file: MEMPAGE.F77
Compiled on 16-Nov-82 at 16:18:11 by AOS F77 Rev 2.10
Options: F77/INTEGER=2/LOGICAL=2/L=MEMPAGE.LS

    1          PROGRAM MEMPAGE   ! TO DETERMINE THE NUMBER OF MEMORY PAGES
    2   C                          THIS PROGRAM REQUIRES AT RUNTIME.
    3
    4          INTEGER*2 IER
    5          INTEGER*2 AC0, AC1, AC2
    6
    7   %INCLUDE 'MEMPAGE_SYMBOLS.F77.IN'
    8   ****  F77 INCLUDE file for system parameters ****
    9
   10   ****   Parameters for SYSID  ****
   11
   12          INTEGER*2  ISYS_PSTAT
   13          PARAMETER (ISYS_PSTAT = 5)      ! ?PSTAT = 5K
   14
   15   ****    Parameters for PARU   ****
   16
   17
   18          INTEGER*2  ISYS_PSBK
   19          PARAMETER (ISYS_PSBK = 25)      ! ?PSBK = 31K
   20
   21          INTEGER*2  ISYS_PSPS
   22          PARAMETER (ISYS_PSPS = 26)      ! ?PSPS = 32K
   23
   24          INTEGER*2  ISYS_PSSF
   25          PARAMETER (ISYS_PSSF = 27)      ! ?PSSF = 33K
   26
   27          INTEGER*2  ISYS_PSLTH
   28          PARAMETER (ISYS_PSLTH = 46)     ! ?PSLTH = 56K
   29
   30
   31   ****  END of F77 INCLUDE file for system parameters ****
   32
   33          INTEGER*2 PSTAT_PACKET(0:ISYS_PSLTH)
   34
   35          AC0 = -1
   36          AC1 =  0
   37          AC2 =  WORDADDR(PSTAT_PACKET)
   38
   39          IER = ISYS(ISYS_PSTAT, AC0, AC1, AC2)
   40          IF (IER .NE. 0) CALL ERRCODE(IER, 0)
   41
   42          PRINT *, '<NL> User Unshared Pages: ', PSTAT_PACKET(ISYS_PSBK)
   43          PRINT *, '<NL> User Shared Pages:   ', PSTAT_PACKET(ISYS_PSPS)
   44          PRINT *, '<NL> Start User Shared:   ', PSTAT_PACKET(ISYS_PSSF)
   45
   46          STOP
   47          END
```

Figure 7-6. A Listing of Program MEMPAGE.LS

The command to create MEMPAGE.OB is

```
F77 MEMPAGE
```

The general command to create MEMPAGE.PR is

```
DELETE/2=IGNORE MEMPAGE.LS
F77LINK/L=MEMPAGE.MAP/MAP/KTOP=n MEMPAGE
```

Below are the results of linking MEMPAGE for various values of /KTOP=n and then executing the program. Note how the value of n (Column 1) directly affects the size of MEMPAGE.PR, both on the disk (Column 2) and in memory (the sum of Columns 3 and 4).

--------- Result of XEQ MEMPAGE -----------

n	Size of MEMPAGE.PR	User Unshared Pages	User Shared Pages	First User Shared Page
5	Link error: FILE TOO LARGE FOR ADDRESS SPACE			
6	12,288	1	5	1
7	14,336	2	5	2
10	20,480	5	5	5
15	30,720	10	5	10
27	55,296	22	5	22
32	65,536	27	5	27
33	67,584	ERROR: ILLOGICAL PROCESS ADDRESS SPACE DEFINITION		

Cautions about Specifying /KTOP=n

There are no convenient rules for specifying the smallest value of /KTOP=n as F77LINK creates your program files. The execution of MEMPAGE shows that n should be at least 6 and at most 32.

Remember that "ordinary" (i.e., non-COMMON and non-SAVEd) variables and arrays require space on the stack at runtime. A program with such variables and arrays could compile and link with no indicated errors, but then abort at runtime because of a stack overflow. In this case, the stack had to expand to accommodate the variables and arrays, but its growth attempted to expand into the program's shared area to cause the overflow and process termination.

For example, consider program MEMPAGE.F77. The commands

```
F77 MEMPAGE
DEL/2=IGNORE MEMPAGE.MAP
F77LINK/L=MEMPAGE.MAP/MAP/KTOP=6 MEMPAGE
XEQ MEMPAGE
```

successfully compile, link, and execute the program. Suppose you change line six of MEMPAGE.F77 to

```
      INTEGER*2 TABLE(1024)
```

The same four commands successfully compile and link the program, yet it aborts at runtime. Why? The compiler does *not* allocate 2,048 bytes (one page) for array TABLE. Instead, at runtime the program tries to allocate 2,048 bytes on the stack for TABLE; they simply don't exist, and the program aborts. If you change "/KTOP=6" to "/KTOP=7" in the F77LINK command, then MEMPAGE.PR executes successfully.

We have added letters A, B, C, D, and E to identify five lines. Compare Figure 7-5 and 7-7, and make the following observations about program MEMPAGE:

- Lines A and B show that the shared area of MEMPAGE.PR is from 002000K to 13777K. These locations aren't necessarily entirely occupied by shared data and shared code.
- Line B shows that the last location in MEMPAGE.PR is 13777K=6143, where 6143 = 6*1024-1; this number is the direct result of specifying /KTOP=6 in the F77LINK command that created MEMPAGE.PR.
- Lines C and D show that the stack occupies at least locations 001361K to 001416K; at runtime it can expand into locations 001417K to 001777K inclusive, which is an increase of 361K=241 words. You have already seen that addition of a 1024-word array causes a runtime stack overflow error.
- Lines D and E show that the shared data and shared code reside in locations 002000K through 012415K.

In conclusion, you may have to use a certain amount of trial and error to select the smallest possible value of /KTOP=n in your F77LINK commands. The load map can help with your selection. The difference between SBOT and NMAX is a *rough estimate* of the amount of memory the stack can expand into.

Other Ways

So far in this section ("Reducing Memory and Disk Usage of Program Files") we have explained the use of the /KTOP=n Link switch to reduce the size of the .PR file. Two other ways of reducing memory and disk usage are available: eliminating the symbol table file and executing a program via the PROCESS command.

Eliminating the Symbol Table File

By default, Link creates *two* files: a program (.PR) file and a symbol table (.ST) file. You usually don't need the .ST file unless you're going to debug the program using the AOS debugger (DEBUG). The easiest way to eliminate a program's .ST file is not to create it in the first place. To do this, give the /SUPST (SUPpress Symbol Table) switch to F77LINK. F77LINK then passes this switch intact to Link.

The PROCESS Command

You usually execute a program by typing XEQ programname. You can also execute a program by giving the PROCESS command. This latter method lets you exercise rather fine control over the process, including its memory usage.

For example, recall program SMALL_LARGE. The command

```
XEQ SMALL_LARGE
```

executes SMALL_LARGE.PR, and SMALL_LARGE.PR requires a full 32K words of memory because Link created it without the /KTOP=n switch. However, the command

```
PROCESS/BLOCK/DEFAULT/IOC/MEMORY=7 SMALL_LARGE
```

also executes SMALL_LARGE.PR. Then, SMALL_LARGE.PR only requires 7K words of memory as a result of this PROCESS command. It doesn't matter if Link created SMALL_LARGE.PR with a /KTOP= value of 8 or more.

In general, you may have to experiment to find the smallest possible value for /MEMORY if you give the /PROCESS command to execute a program. Too small a value gives a runtime error message (ILLEGAL MAXIMUM PROCESS SIZE) and aborts the program.

End of Chapter

Using the Load Map

Sometimes you can

- Invoke F77LINK without a value of /KTOP (same as /KTOP=32), and with the /MAP and /L=<map_filename> switches.
- Examine the Link-created load map.
- Select a value of /KTOP for the next F77LINK command.

For example, Figure 7-7 contains a portion of MEMPAGE.MAP as created by Link when KTOP=6.

```
LINK REVISION 04.10 ON 11/17/82 AT 11:56:34

  MEMPAGE 02.10
  F77DGPCT
  F77IOREV 02.10
  CO.ERLOP
  LC?EINIT
  CO.EINIT

  .
  .
  .

  FINIT
  DU.MAIN.
  CO.ERLOG
  DU.TFINT
  CO.PTRDF
  II.ENDLI

ZBOT:           000050
ZMAX:           000073
NBOT:           000400
USTA:           000447
NMAX:           001417
SBOT:           002000          <-- (A)
NTOP:           013777          <-- (B)
STACK SIZE:     000036 (OCTAL)

  TYPE     NAME               ADDRESS      LENGTH       END
  --------------------------------------------------------
  COMM UC  AB                 000000       000046       000045
  PART UC  ZR                 000050       000023       000072
  COMM UC  UST                000400       000023       000422
  COMM UC  TCB                000423       000024       000446
  COMM UD  SCR?11.            000447       000105       000553
  COMM UD  SCR?5.             000554       000105       000660
  PART UD  UD                 000661       000000
  PART UC  UC                 000661       000500       001360
  COMM UC  STACK              001361       000036       001416  <-- (C)
  PART SD  SD                 002000       000050       002047  <-- (D)
  PART SC  SC                 002050       010346       012415  <-- (E)
=MEMPAGE.PR CREATED
```

Figure 7-7. A Portion of MEMPAGE.MAP

Chapter 8
Overlays

This chapter gives a general explanation of overlays. Then, it explains the construction of two sample F77 programs that contain overlays.

Introduction

In many cases, an application program is too large to reside in main memory at one time. If you have written your program in modules (for example, using subroutines and functions), you can often use a software technique known as *overlaying* to overcome this size difficulty. An *overlay area* is a reserved portion in main memory that various modules can share.

Several modules (subroutines and functions, but not the main program) reside in a special file on the disk. These modules are the individual overlays. When the main program needs such a module, it requests AOS to copy the module from the disk file to the overlay area. Then, the newly arrived code executes. While a given subprogram is in the overlay area, the other subroutines not in use remain on the disk.

NOTE: Depending on Link's construction of the program, AOS might copy additional modules along with the needed one from the disk file.

A main program and its overlay file that contains several modules are like a doctor who must deal with several patients. Only one patient at a time is in the office while the others wait outside. A patient moves from the waiting room to the office to receive one part of a treatment and then back to the waiting room until the doctor is ready to have him or her return to the office for the next step of the treatment. (In contrast, an overlay "moves" — i.e., is copied — only from the disk to memory, where it overwrites the memory it occupies.)

Licensed Material-Property of Data General Corporation

Example 1 — A Program Using Overlays

For example, consider an AOS F77 programmer named Peggy who works for a college. Her job is to write a program to analyze each prospective graduate's records and determine if the student's courses satisfy various department and college-wide requirements. There are 30 departments, and all students must take specific courses such as English composition and physical education. The program functions fall fairly naturally into these steps:

1. Read each student's course records into a two-dimension array, where the fields in each row of the array represent one completed course. For example:

- ENGLISH	101	FALL 79	B-	3.0	PEARL
- MATH	110	FALL 79	C+	4.0	HEITZ
- COMP SCI	101	FALL 79	A-	3.0	ZETTERHOLM
- HISTORY	131	FALL 79	B	3.0	MCDONALD

   The next-to-last field contains the number of semester hours in the course. The last field is the instructor's name.
2. Determine whether or not English composition was successfully completed.
3. Determine whether or not physical education was successfully completed.
4. Depending on the student's major, determine whether or not the department requirements were met.
5. Determine whether or not the student has passed 128 semester hours.

Peggy has no problems with the organization of the program. The course records are in a file sorted by student number, so the main program follows these major steps for each student:

1. Read all the course records and place them in COMMON in the two-dimension array RECORDS.
2. CALL Subroutine CHECK_ENG, which also declares RECORDS in COMMON, to examine RECORDS for English composition records, flag them if found so that certain future subroutines will skip them, and print the result.
3. CALL subroutine CHECK_PED to examine array RECORDS in COMMON for physical education records and similarly process them.
4. Depending on the student's major (art, biology, chemistry, ..., zoology), CALL a subroutine (CHECK_ART, CHECK_BIOLOGY, CHECK_CHEMISTRY, ..., CHECK_ZOOLOGY) to examine RECORDS in COMMON that meet the major's requirements and similarly process them. Perhaps all these subroutines aren't required, but this choice of one subroutine for each of the 30 majors is one way to introduce overlays.
5. CALL subroutine COUNT_HOURS to examine array RECORDS in COMMON and count the number of hours earned from all the course records, then print this number.

Based on these functions and steps, Peggy finds the coding to be straightforward. She writes the main program and its (1+1+30+1)=33 subroutines, and links them to create GRADUATION_CHECK.PR. However, GRADUATION_CHECK.PR exceeds 32K words; it can't execute. What can she do?

She begins by constructing a diagram of the desired program file and its accompanying overlay file. The program file contains the main program, subroutine CHECK_ENG, subroutine CHECK_PED, subroutine COUNT_HOURS, and *space for just one of the 30 subroutines CHECK_ART through CHECK_ZOOLOGY*. The overlay file contains all of the 30 subroutines CHECK_ART through CHECK_ZOOLOGY. See Figure 8-1.

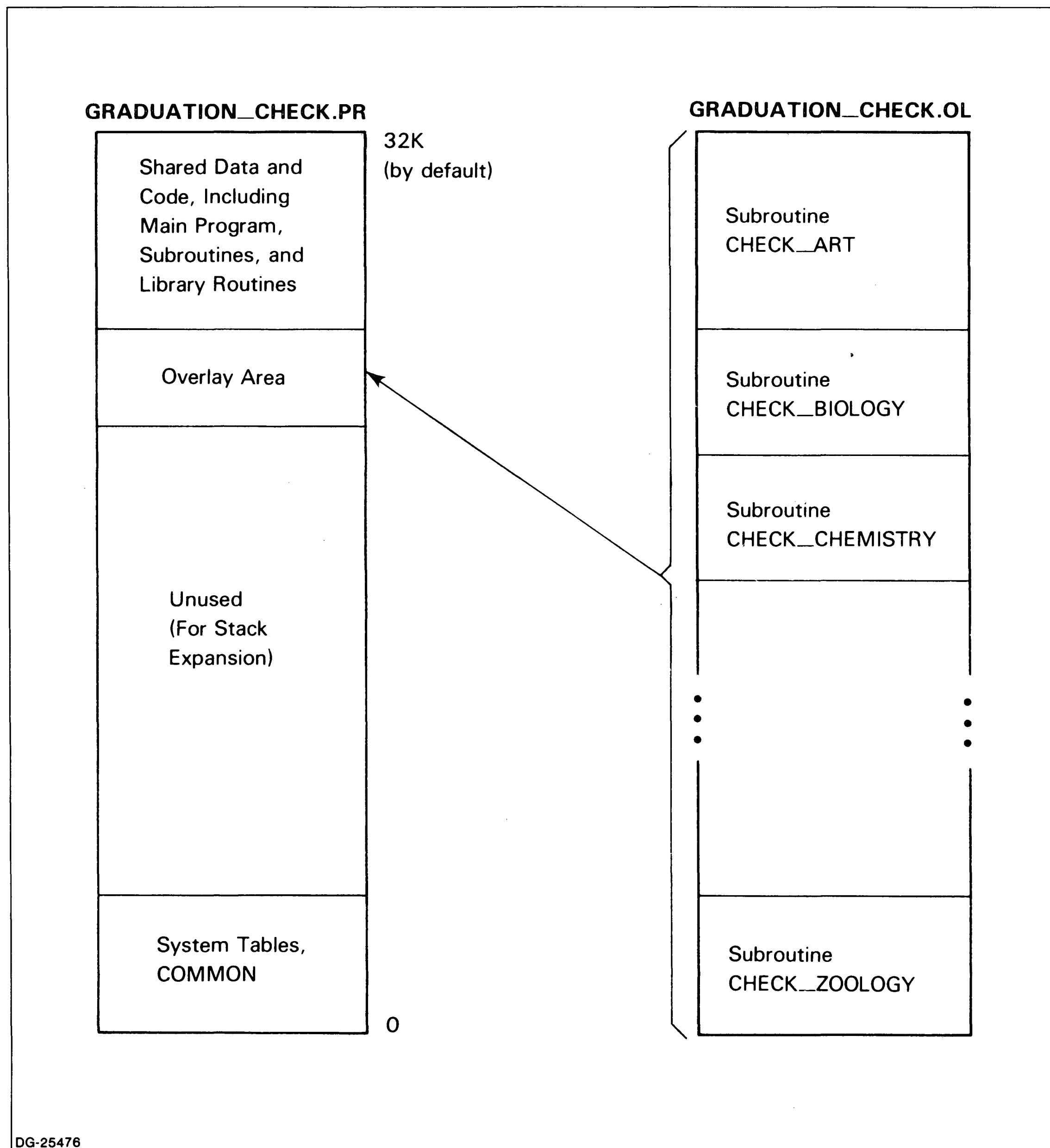


Figure 8-1. The Desired Organization of GRADUATION_CHECK.PR and Its Overlay File

At runtime, a CALL to any of the subroutines that reside in GRADUATION_CHECK.OL results in the movement (i.e., duplication) of the subroutine into the overlay area of GRADUATION_CHECK.PR. Then, the instructions from the newly arrived subroutine execute just as if they had always been in GRADUATION_CHECK.PR.

The key to creating GRADUATION_CHECK.PR and GRADUATION_CHECK.OL is the command to F77LINK. The *AOS Link User's Manual* has a general explanation of giving commands to Link to create related program and overlay files. Basically, Peggy uses the symbols !*, !, and *! as part of her F77LINK command. These symbols specify the construction of the overlay areas in the .PR file and the placement of the subroutines in the .OL file as follows.

!* Start an overlay area.

! Subprogram separator in the overlay file.

*! End an overlay area.

Examine the F77LINK skeleton commands next. The first one creates GRADUATION_CHECK.PR and GRADUATION.OL according to Figure 8-1. For comparison, the second command creates a GRADUATION_CHECK.PR that contains all the subroutines (even though the program file is too large to execute).

```
F77LINK GRADUATION_CHECK &      F77LINK GRADUATION_CHECK &
CHECK_ENG CHECK_PED &           CHECK_ENG CHECK_PED &
!* CHECK_ART ! &                CHECK_ART &
CHECK_BIOLOGY ! &               CHECK_BIOLOGY &
CHECK_CHEMISTRY ! &             CHECK_CHEMISTRY &
...                             ...
CHECK_ZOOLOGY *! &              CHECK_ZOOLOGY &
COUNT_HOURS                     COUNT_HOURS
```

The movement of subroutines from the overlay file into the overlay area and their subsequent execution requires much more time than the CALLing of subroutines such as CHECK_ENG that are always in memory. But, creating overlay files lets Peggy and you meet memory requirements such as those presented by the graduation program with its many subprograms.

Example 2 — A Program Implementing Overlays

Consider the following two-dimension array named ARRAY with two rows and three columns.

```
811.0   812.0   813.0
821.0   822.0   823.0
```

We wish to construct a main program and five subroutine subprograms that function as follows:

- Main program SAMPLE_OVERLAY declares ARRAY, places it in a named COMMON area, and assigns the above values to it.
- Subroutine SUB_00_00 finds the sum of the first row of ARRAY. The subroutine resides in the first (number 00) of the overlay areas; it is the first (number 00) of the two subroutines in this overlay area.
- Subroutine SUB_00_01 finds the sum of the second row of ARRAY. The subroutine resides in the first (number 00) of the overlay areas; it is the second (number 01) of the two subroutines in this overlay area.
- Subroutine SUB_01_00 finds the sum of the first column of ARRAY. The subroutine resides in the second (number 01) of the overlay areas; it is the first (number 00) of the three subroutines in this overlay area.
- Subroutine SUB_01_01 finds the sum of the second column of ARRAY. The subroutine resides in the second (number 01) of the overlay areas; it is the second (number 01) of the three subroutines in this overlay area.
- Subroutine SUB_01_02 finds the sum of the third column of ARRAY. The subroutine resides in the second (number 01) of the overlay areas; it is the third (number 02) of the three subroutines in this overlay area.

It isn't at all necessary to use overlays in this situation, but doing so outlines a way to construct a program file with more than one overlay area.

Figure 8-2 shows the desired construction of SAMPLE_OVERLAY.PR and of SAMPLE_OVERLAY.OL.

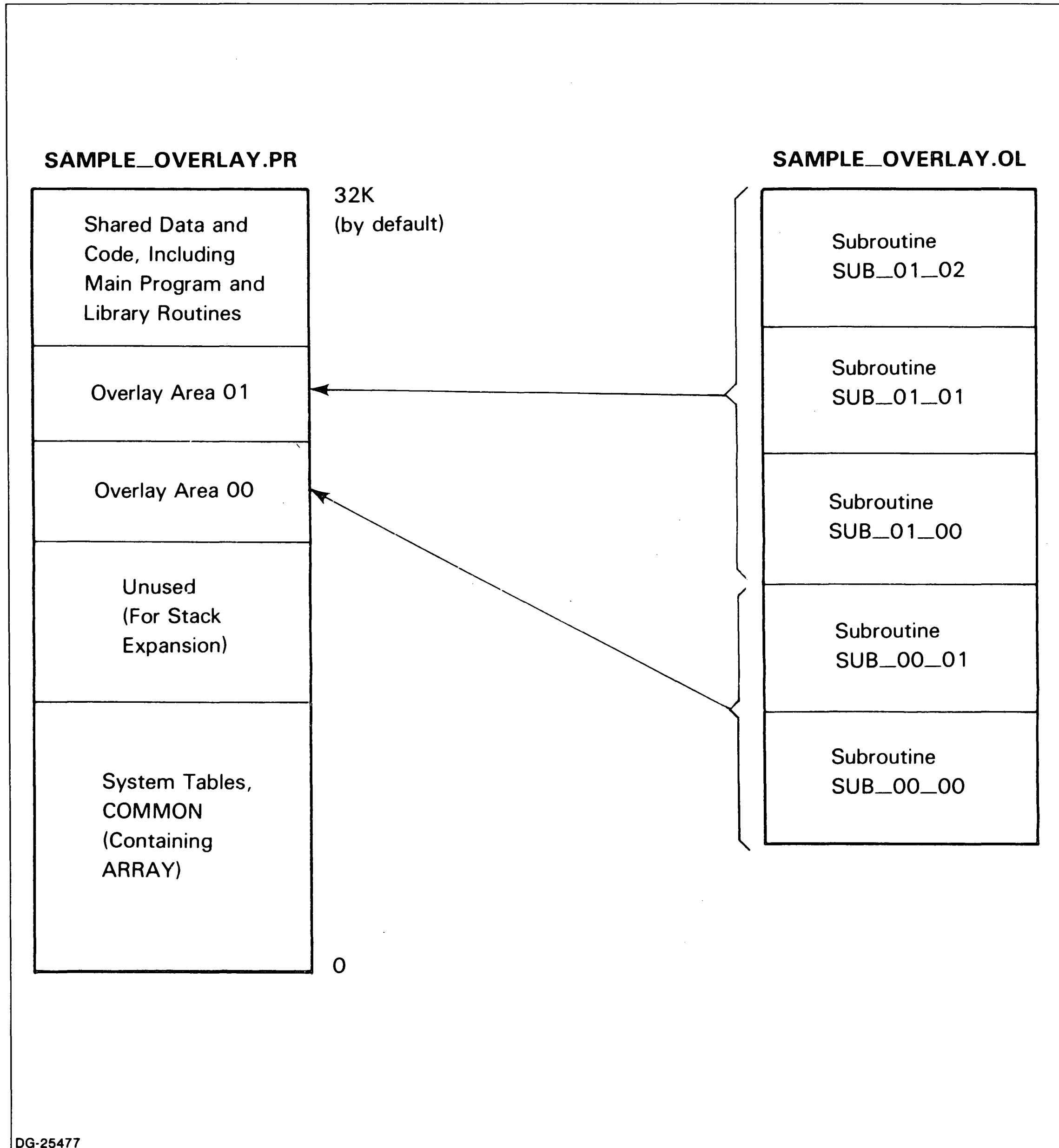


Figure 8-2. The Desired Organization of SAMPLE_OVERLAY.PR and Its Overlay File

Link allows a maximum of 63 overlay areas in a program file with up to 511 overlays for each overlay area. It stores all the program modules sequentially in the overlay file and does not create distinct overlay areas there. We have two overlay areas in SAMPLE_OVERLAY.PR instead of one just to indicate how to create more than one overlay area in a program file.

The coding of the six program units is straightforward. We make few particular allowances for overlays because their construction occurs during linking and their execution occurs at runtime. However, we have placed ARRAY in COMMON because different subroutines, not all of which are in memory at the same time, access it at runtime.

Figures 8-3, 8-4, 8-5, 8-6, 8-7, and 8-8 respectively contain program units SAMPLE_OVERLAY.F77, SUB_00_00.F77, SUB_00_01.F77, SUB_01_00.F77, SUB_01_01.F77, and SUB_01_02.F77.

```
PROGRAM SAMPLE_OVERLAY  ! TO DEMONSTRATE OVERLAYS

REAL ARRAY(2,3)
COMMON /COLD/ ARRAY
DATA ARRAY / 811.0, 821.0, 812.0, 822.0, 813.0, 823.0 /


PRINT *, '<NL>*** SUBROUTINE CALLING BEGINS NOW ***<NL><NL>'

CALL SUB_00_00(2, 3, SUM)
PRINT *, 'SUM OF THE ELEMENTS IN ROW 1    IS ', SUM, '<NL>'

CALL SUB_00_01(2, 3, SUM)
PRINT *, 'SUM OF THE ELEMENTS IN ROW 2    IS ', SUM, '<NL>'

CALL SUB_01_00(2, 3, SUM)
PRINT *, 'SUM OF THE ELEMENTS IN COLUMN 1 IS ', SUM, '<NL>'

CALL SUB_01_01(2, 3, SUM)
PRINT *, 'SUM OF THE ELEMENTS IN COLUMN 2 IS ', SUM, '<NL>'

CALL SUB_01_02(2, 3, SUM)
PRINT *, 'SUM OF THE ELEMENTS IN COLUMN 3 IS ', SUM, '<NL>'

PRINT *, '<NL>*** SUBROUTINE CALLING ENDS NOW ***<NL>'

STOP
END
```

Figure 8-3. Main Program SAMPLE_OVERLAY.F77

```
      SUBROUTINE SUB_01_00 (ROWS, COLUMNS, SUM)

C     THIS SUBROUTINE FINDS THE SUM OF THE ELEMENTS IN THE
C        1ST COLUMN OF <ARRAY>.  IT IS IN OVERLAY AREA 01, AND
C        IS THE FIRST INDIVIDUAL OVERLAY (NUMBER 00) IN THE AREA.

      INTEGER ROWS    ! NUMBER OF ROWS IN <ARRAY> FOR POSSIBLE ADDITION
      INTEGER COLUMNS ! NUMBER OF COLUMNS IN <ARRAY> FOR POSSIBLE ADDITION
      REAL ARRAY (2, 3)

      COMMON /COLD/ ARRAY

      SUM = 0.0

      DO 10 I = 1, ROWS
              SUM = SUM + ARRAY(I,1)
   10 CONTINUE

      RETURN
      END
```

Figure 8-6. Subprogram SUB_01_00.F77

```
      SUBROUTINE SUB_01_01 (ROWS, COLUMNS, SUM)

C     THIS SUBROUTINE FINDS THE SUM OF THE ELEMENTS IN THE
C        2ND COLUMN OF <ARRAY>.  IT IS IN OVERLAY AREA 01, AND
C        IS THE SECOND INDIVIDUAL OVERLAY (NUMBER 01) IN THE AREA.

      INTEGER ROWS    ! NUMBER OF ROWS IN <ARRAY> FOR POSSIBLE ADDITION
      INTEGER COLUMNS ! NUMBER OF COLUMNS IN <ARRAY> FOR POSSIBLE ADDITION
      REAL ARRAY (2, 3)

      COMMON /COLD/ ARRAY

      SUM = 0.0

      DO 10 I = 1, ROWS
              SUM = SUM + ARRAY(I,2)
   10 CONTINUE

      RETURN
      END
```

Figure 8-7. Subprogram SUB_01_01.F77

```
      SUBROUTINE SUB_00_00 (ROWS, COLUMNS, SUM)

C     THIS SUBROUTINE FINDS THE SUM OF THE ELEMENTS IN THE
C        1ST ROW OF <ARRAY>.  IT IS IN OVERLAY AREA 00, AND
C        IS THE FIRST INDIVIDUAL OVERLAY (NUMBER 00) IN THE AREA.

      INTEGER ROWS    ! NUMBER OF ROWS IN <ARRAY> FOR POSSIBLE ADDITION
      INTEGER COLUMNS ! NUMBER OF COLUMNS IN <ARRAY> FOR POSSIBLE ADDITION
      REAL ARRAY (2, 3)

      COMMON /COLD/ ARRAY

      SUM = 0.0

      DO 10 J = 1, COLUMNS
              SUM = SUM + ARRAY(1,J)
   10 CONTINUE

      RETURN
      END
```

Figure 8-4. Subprogram SUB_00_00.F77

```
      SUBROUTINE SUB_00_01 (ROWS, COLUMNS, SUM)

C     THIS SUBROUTINE FINDS THE SUM OF THE ELEMENTS IN THE
C        2ND ROW OF <ARRAY>.  IT IS IN OVERLAY AREA 00, AND
C        IS THE SECOND INDIVIDUAL OVERLAY (NUMBER 01) IN THE AREA.

      INTEGER ROWS    ! NUMBER OF ROWS IN <ARRAY> FOR POSSIBLE ADDITION
      INTEGER COLUMNS ! NUMBER OF COLUMNS IN <ARRAY> FOR POSSIBLE ADDITION
      REAL ARRAY (2, 3)

      COMMON /COLD/ ARRAY

      SUM = 0.0

      DO 10 J = 1, COLUMNS
              SUM = SUM + ARRAY(2,J)
   10 CONTINUE

      RETURN
      END
```

Figure 8-5. Subprogram SUB_00_01.F77

```
      SUBROUTINE SUB_01_02 (ROWS, COLUMNS, SUM)

C     THIS SUBROUTINE FINDS THE SUM OF THE ELEMENTS IN THE
C        3RD COLUMN OF <ARRAY>.  IT IS IN OVERLAY AREA 01, AND
C        IS THE THIRD INDIVIDUAL OVERLAY (NUMBER 02) IN THE AREA.

      INTEGER ROWS    ! NUMBER OF ROWS IN <ARRAY> FOR POSSIBLE ADDITION
      INTEGER COLUMNS ! NUMBER OF COLUMNS IN <ARRAY> FOR POSSIBLE ADDITION
      REAL ARRAY (2, 3)

      COMMON /COLD/ ARRAY

      SUM = 0.0

      DO 10 I = 1, ROWS
              SUM = SUM + ARRAY(I,3)
   10 CONTINUE

      RETURN
      END
```

Figure 8-8. Subprogram SUB_01_02.F77

The command to compile the six program units is

```
F77 (SAMPLE_OVERLAY SUB_00_00 SUB_00_01 SUB_01_00 SUB_01_01 SUB_01_02)
```

The all-important command to F77LINK that results in the creation of SAMPLE_OVERLAY.PR and SAMPLE_OVERLAY.OL is

```
DELETE/2=IGNORE SAMPLE_OVERLAY.MAP;
F77LINK/L=SAMPLE_OVERLAY.MAP/MAP SAMPLE_OVERLAY &
                                   !* SUB_00_00 ! SUB_00_01 *! &
                                   !* SUB_01_00 ! SUB_01_01 ! SUB_01_02 *!
```

Figure 8-9 contains a portion of file SAMPLE_OVERLAY.MAP.

```
LINK REVISION 04.10 ON 11/12/82 AT 10:57:14

  SAMPLE_OVERLAY 02.10
  SUB_00_00 02.10
  SUB_00_01 02.10
  SUB_01_00 02.10
  SUB_01_01 02.10
  SUB_01_02 02.10
  F77DGPCT
  F77IOREV 02.10
  CO.ERLOP
  LC?EINIT
  CO.EINIT

  .
  .
  .

  FINIT
  DU.MAIN.
  CO.ERLOG
  DU.TFINT
  CO.PTRDF
  II.ENDLI

ZBOT:           000050
ZMAX:           000077
NBOT:           000400
USTA:           000504
NMAX:           001473
SBOT:           056000
NTOP:           077777
STACK SIZE:     000036 (OCTAL)

  TYPE     NAME                ADDRESS     LENGTH      END
  ---------------------------------------------------------
  COMM UC  AB                  000000      000046      000045
  PART UC  ZR                  000050      000027      000076
  COMM UC  UST                 000400      000023      000422
  COMM UC  TCB                 000423      000024      000446
  COMM UC  OLDIR               000447      000023      000471
  COMM UC  RHT                 000472      000012      000503
  COMM UD  COLD                000504      000014      000517
  COMM UD  SCR?11.             000520      000105      000624
  COMM UD  SCR?5.              000625      000105      000731
  PART UD  UD                  000732      000000
  PART UC  UC                  000732      000503      001434
  COMM UC  STACK               001435      000036      001472
  COMM SC  AREA 00             056000      002000      057777
  COMM SC  AREA 01             060000      002000      061777
  PART SD  SD                  062000      000233      062232
  PART SC  SC                  062233      013615      076047
```

Figure 8-9. A Portion of SAMPLE_OVERLAY.MAP (continues)

```
  TYPE     NAME               ADDRESS     LENGTH      END
  -----------------------------------------------------
AREA 00
  PART SC  OVERLAY 00         056000      000040      056037
  PART SC  OVERLAY 01         056000      000040      056037

  TYPE     NAME               ADDRESS     LENGTH      END
  -----------------------------------------------------
AREA 01
  PART SC  OVERLAY 00         060000      000037      060036
  PART SC  OVERLAY 01         060000      000037      060036
  PART SC  OVERLAY 02         060000      000037      060036
=SAMPLE_OVERLAY.PR CREATED
```

Figure 8-9. A Portion of SAMPLE_OVERLAY.MAP (concluded)

The results of the command

XEQ SAMPLE_OVERLAY

are as follows.

**** SUBROUTINE CALLING BEGINS NOW ****

SUM OF THE ELEMENTS IN ROW 1 IS 2436.

SUM OF THE ELEMENTS IN ROW 2 IS 2466.

SUM OF THE ELEMENTS IN COLUMN 1 IS 1632.

SUM OF THE ELEMENTS IN COLUMN 2 IS 1634.

SUM OF THE ELEMENTS IN COLUMN 3 IS 1636.

**** SUBROUTINE CALLING ENDS NOW ****

STOP

Examine Figure 8-9 and note how Link creates identifiers for the subroutines in the overlay file it builds. For example, our subroutine SUB_00_01 is known to Link as AREA 00, OVERLAY 01. It might be more natural to call this subroutine SUM_ROW_02, but Link would label it AREA 00, OVERLAY 01. The existence of SUM_ROW_02 would appear in SAMPLE_OVERLAY.MAP at the beginning and not at the end as we might like. We chose "SUB_00_01" to have it agree with the Link-created name.

This chapter does not explain all possible ways to construct overlay files. Much of this construction depends on Link, thus, you should read its documentation carefully.

End of Chapter

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., BYTEADDR); all others are lowercase.

A

Access Control List (ACL) 3-2
address
  byte 6-2
  word 6-2
AF77SYM.SR 6-5, 6-9f
AOS 1-3, 1-1, 2-2, 2-6, 2-18, 3-1, 3-3, 4-1, 4-3, 4-5, 4-22, 4-27, 4-45, 4-53, 7-5, 8-1
array storage, multidimension 6-17ff
assembly language/multitasking interface 4-24f
assembly language subprograms 6-1ff

B

block, common return 6-3ff
byte address 6-2
BYTEADDR 3-2f

C

call, system 1-3, 4-10
carriage control tape 7-5
case sensitivity 6-20
chi-square 2-15
CLI (special subroutine) 3-15ff
CLI.PR 3-15ff
CLRE 6-2ff, 2-3, 2-7, 6-1, 6-10, 6-16f, 6-20, 6-22
CLREERMES.SR 4-29
COBOL 1-2
code
  in-line 1-4
  re-entrant 4-20ff, 4-1
command format conventions iv
Command Line Interpreter (CLI) 1-3
common return block 6-3ff
compiler programs, F77 1-3, 1-5ff
conflicts, interlanguage 6-20
contacting Data General v
conventions
  command format iv
  documentation iv, v
count, protect 4-45

D

Data General systems engineering v
Data General, contacting v
DATE 2-2
/DEBUG F77.CLI switch 7-2
/DEBUG F77LINK.CLI switch 1-5
debugger, SWAT 5-1ff
debugging 5-1ff
DEF macro 6-9f, 6-15
DEFARGS macro 6-9f, 6-15
DEFTMPS macro 6-9f, 6-15
DG/L 6-1, 6-17, 6-23
DG/L and F77 6-25ff
disk usage by program files 7-12ff
documentation conventions iv, v
documentation, related iv
documentation remarks form v
dope vector 6-2f, 6-16
dormant task 4-11
/DOTRIP F77.CLI switch 7-2f
.DUSR symbols 3-4

E

EJSR 4-24
END macro 6-9f, 6-15
enhancements, program 7-1ff
:ERMES 2-3, 2-6
ERRCODE 2-3ff, 2-7, 4-29
ERR.F77.IN 2-3, 2-6, 4-29, 7-1
error message file 2-3
ERRORLOG specifier 2-3, 2-7
ERRTEXT 2-7ff, 2-3
EXEC 7-5
executing task 4-11
EXIT 2-10, 4-7, 4-9
.EXTN 4-24

F

F5_MT.LB 4-27
F77 compiler programs 1-3, 1-5ff
F77BUILD_SYM 3-4ff
F77.CLI 1-5, 4-27, 4-64, 5-1
F77DGPCT.OB 1-5
F77_DOCUMENTATION 3-2
F77ENV.LB 1-4f
F77ENV_MT.LB 4-27

F77ERMES.SR 4-29
F77_FMAC.SR 6-16f, 6-5, 6-9f, 6-15
F77IO.LB 1-5
F77LINK.CLI 1-5f, 2-1, 4-7, 4-9, 4-25ff, 4-28, 4-39, 4-64, 4-67, 5-1, 7-12, 8-3f
F77MT.LB 4-24f, 4-27
F77STACK macro 4-65ff
faster programs 7-3f
FCALL macro 6-9
FENTRY macro 6-9f, 6-15
FMAC.SR 6-16
form
  documentation remarks v
  TIPS order v
format conventions, command iv
Forms Control Utility (FCU) 7-5
forms, printing special 7-5ff
FORTRAN 5 6-1, 6-16f, 6-23
FORTRAN 5 and F77 6-23ff
FORTRAN 5 multitasking programs 4-25
frame pointer 6-2, 6-10
FRET macro 6-9f, 6-15

H

high-level/F77 programs 6-23ff

I

?IDGOTO 4-24
?IDKIL 4-24
?IDPRI 4-24
?IFPU 4-24
initial task 4-28
in-line code 1-4
interface, assembly language/multitasking 4-24f
interlanguage conflicts 6-20
/IOCONFLICT F77LINK.CLI switch 4-27f, 4-64
IOSTAT variable 2-3, 2-6
IO_CHAN function 3-20f, 3-1
ISA.ERR macro 6-16f
ISA.NORM macro 6-16f
ISYS and multitasking 3-20, 4-27
ISYS function 3-1ff, 4-27, 6-1

K

KILL 4-7
/KTOP F77LINK.CLI switch 7-14, 7-16ff

L

Link 1-3ff, 1-2, 4-7, 4-28, 5-1, 7-12
LITMACS.SR 6-9

M

MAINSTACK macro 4-66ff
manuals, related iv
MASM 1-2, 3-5, 3-7, 4-66
MASM.PS 6-5, 6-12, 6-14
memory usage by program files 7-12ff
message file, error 2-3
multidimension array storage 6-17ff
multitask stack definition 4-65ff
multitasking 4-1ff
multitasking and ISYS 3-20, 4-27
multitasking interface, assembly language 4-24f

N

Notice
  Release iv, 1-6, 2-3, 2-6f, 5-13, 7-3
  Update iv, 1-6

O

operating system updating 3-10
/OPT F77.CLI switch 7-2
order form, TIPS v
overlay area 8-1
overlays 8-1ff

P

PARU.32.SR 3-3
PARU.LS 3-4ff
PARU.SR 3-4ff, 3-2f, 4-29
pass by reference 6-3
pass by value 6-3
per task area 4-65
PL/I 1-2, 6-1, 6-17, 6-23
PL/I and F77 6-28ff
pointer, frame 6-2, 6-10
printing special forms 7-5ff
?PROC 3-15ff
Product Support Manual 7-3
program enhancements 7-1ff
program files
  reducing disk usage 7-12ff
  reducing memory usage 7-12ff
programs
  F77 compiler 1-3, 1-5ff
  faster 7-3f
  high-level/F77 6-23ff
  F77/assembly 6-1ff
protect count 4-45

Q

QPRINT 7-5
QSYM.F77.IN 3-4ff

R

RANDOM 2-11ff
?RCALL 6-2
ready-to-run task 4-11
reducing program disk usage 7-12ff
reducing program memory usage 7-12ff

re-entrant code 4-20ff, 4-1
reference, pass by 6-3
related documentation iv
related manuals iv
Release Notice iv, 1-6, 2-3, 2-6f, 5-13, 7-3
remarks form, documentation v
return block, common 6-3ff
routines
  runtime 1-3ff
  specific runtime 2-1ff
runtime routines 1-3ff
runtime routines, specific 2-1ff

S

?SACL 3-2ff
S?ATTR macro 6-9
/SAVEVARS F77.CLI switch 7-2f
scheduler, task 4-5, 4-11
SED text editor 3-15
sensitivity, case 6-20
Software Trouble Report (STR) 5-13
special forms, printing 7-5ff
specific runtime routines 2-1ff
stack definition, multitask 4-65ff
states, task 4-7ff
storage, multidimension array 6-17ff
/SUB F77.CLI switch 7-2
subprograms, assembly language 6-1ff
suspended task 4-11
SWAT debugger 1-2, 1-5, 5-1ff
SWATI.OB 1-5
SYSID.LS 3-4ff, 3-5ff
SYSID.SR 3-4ff, 3-1, 3-2f
system call 1-3, 4-10
system interface, see ISYS function
system updating, operating 3-10
systems engineering, Data General v
Systems Engineering Newsletter (SENL) 7-3

T

tape, carriage control 7-5
task
  dormant 4-11
  executing 4-11
  initial 4-28
  ready-to-run 4-11
  suspended 4-11
task control block (TCB) 4-20ff, 4-37
/TASKS F77LINK.CLI switch 4-7, 4-9, 4-27f, 4-39, 4-64f, 4-67
task scheduler 4-5, 4-11
task states 4-7ff
task transitions 4-11
T?DQTSK 4-25
T?DRSCH 4-25
T?ERSCH 4-25
text editor, SED 3-15
T?IDKIL 4-24
T?IDPRI 4-25, 4-27
T?IDRDY 4-25
T?IDSUS 4-25
TIME 2-18
TIPS order form v
T?IQTSK 4-25
TITLE macro 6-9, 6-15
T?KILAD 4-25
T?KILL 4-25
T?MYTID 4-25
T?PRI 4-25
T?PRKIL 4-25
T?PROT 4-25
T?PRRDY 4-25
T?PRSUS 4-25
TQDQTSK 4-30ff, 4-23, 4-48
TQDRSCH 4-32, 4-23
TQERSCH 4-33, 4-23
TQIDKIL 4-34, 4-12ff, 4-23, 4-29, 4-45
TQIDPRI 4-35, 4-12f, 4-23f, 4-27
TQIDRDY 4-36, 4-12f, 4-23
TQIDSTAT 4-37, 4-23
TQIDSUS 4-38, 4-12f, 4-23, 4-45
TQIQTSK 4-39, 4-23, 4-30
TQKILAD 4-40, 4-23
TQKILL 4-41, 4-11ff, 4-12f, 4-23, 4-40
TQMYTID 4-42, 4-23
TQPRI 4-43, 4-12f, 4-23
TQPRKIL 4-44, 4-23, 4-45
TQPROT 4-45, 4-23ff, 4-53
TQPRRDY 4-46, 4-12f, 4-23
TQPRSUS 4-47, 4-12f, 4-23, 4-45
TQQTASK 4-48, 4-12f, 4-23f, 4-28, 4-30, 4-66f
TQREC 4-49, 4-12ff, 4-23
TQRECNW 4-50, 4-12, 4-23
TQSTASK 4-51, 4-7, 4-9f, 4-12ff, 4-23, 4-28, 4-66f
TQSUS 4-52, 4-12f, 4-23, 4-26
T?QTASK 4-25
TQUNPROT 4-53, 4-23ff, 4-45
TQXMT 4-54, 4-12, 4-23, 4-26
TQXMTW 4-55, 4-12ff, 4-23
TRACE option 5-1
transitions, task 4-11
?TRCON 4-24
T?REC 4-25
T?RECNW 4-25
T?STASK 4-25
T?SUS 4-25
T?TIDSTAT 4-25
T?UNPROT 4-25
T?XMT 4-25
T?XMTW 4-25

U

Update Notice iv, 1-6
updating, operating system 3-10
usage
  reducing disk 7-12ff
  reducing memory 7-12ff
User Runtime Library (URT.LB) 1-5, 3-1, 4-11, 4-27

V

value, pass by 6-3
vector, dope 6-2f, 6-16
vertical forms unit (VFU) 7-5

W

word address 6-2
WORDADDR 3-2f, 4-48

X

XLPT.PR 7-5

 Licensed Material-Property of Data General Corporation

Data General

TIPS ORDERING PROCEDURE:

Technical literature may be ordered through the Customer Education Service's Technical Information and Publications Service (TIPS).

1. Turn to the TIPS Order Form.
2. Fill in the requested information. If you need more space to list the items you are ordering, use an additional form. Transfer the subtotal from any additional sheet to the space marked "subtotal" on the form.
3. Do not forget to include your MAIL ORDER ONLY discount. (See discount schedules on the back of the TIPS Order Form.)
4. Total your order. (MINIMUM ORDER/CHARGE after discounts of $50.00.)

   If your order totals less than 100.00, enclose a certified check or money order for the total (include sales tax, or your tax exempt number, if applicable) plus $5.00 for shipping and handling.
5. Please indicate on the Order Form if you have any special shipping requirements. Unless specified, orders are normally shipped U.P.S.
6. Read carefully the terms and conditions of the TIPS program on the reverse side of the Order Form.
7. Sign on the line provided on the form and enclose with payment. Mail to:

   TIPS
   Educational Services – M.S. F019
   Data General Corporation
   4400 Computer Drive
   Westboro, MA 01580
8. We'll take care of the rest!

educational services

Data General

users group

Installation Membership Form

Name ______________________ Position ______________________ Date __________

Company, Organization or School ____

Address ______________________ City ______________ State __________ Zip __________

Telephone: Area Code __________ No. ______________ Ext. ______________

1. Account Category

- ☐ OEM
- ☐ End User
- ☐ System House
- ☐ Government

2. Hardware

	Qty. Installed	Qty. On Order
M/600		
MV/Series ECLIPSE®		
Commercial ECLIPSE		
Scientific ECLIPSE		
Array Processors		
CS Series		
NOVA®4 Family		
Other NOVAs		
microNOVA® Family		
MPT Family		

Other __________
(Specify) __________
____

3. Software

- ☐ AOS
- ☐ AOS/VS
- ☐ AOS/RT32
- ☐ MP/OS
- ☐ MP/AOS
- ☐ RDOS
- ☐ DOS
- ☐ RTOS
- ☐ Other

Specify __________

4. Languages

- ☐ ALGOL
- ☐ DG/L
- ☐ COBOL
- ☐ Interactive COBOL
- ☐ PASCAL
- ☐ Business BASIC
- ☐ BASIC
- ☐ Assembler
- ☐ FORTRAN 77
- ☐ FORTRAN 5
- ☐ RPG II
- ☐ PL/1
- ☐ APL
- ☐ Other

Specify __________
____

5. Mode of Operation

- ☐ Batch (Central)
- ☐ Batch (Via RJE)
- ☐ On-Line Interactive

6. Communication

- ☐ HASP
- ☐ HASP II
- ☐ RJE80
- ☐ RCX 70
- ☐ RSTCP
- ☐ 4025
- ☐ X.25
- ☐ SAM
- ☐ CAM
- ☐ XODIAC™
- ☐ DG/SNA
- ☐ 3270
- ☐ Other

Specify __________
____

7. Application Description

○ __________
____
____
____

8. Purchase

From whom was your machine(s) purchased?

- ☐ **Data General Corp.**
- ☐ Other

Specify __________
____

9. Users Group

Are you interested in joining a special interest or regional Data General Users Group?

○ __________
____
____

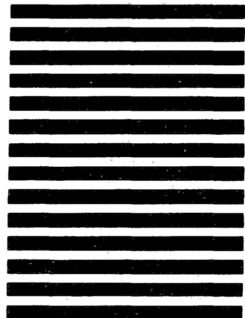
Data General Corporation, Westboro, Massachusetts 01580, (617) 366-8911

CUT ALONG DOTTED LINE

FOLD FOLD

TAPE TAPE

FOLD FOLD



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA. 01772

Postage will be paid by addressee:

ATTN: Users Group Coordinator (C-228)
4400 Computer Drive
Westboro, MA 01581

Data General

TP________

TIPS ORDER FORM
Technical Information & Publications Service

BILL TO:
COMPANY NAME________
ADDRESS ________
CITY ________
STATE________ ZIP ________
ATTN: ________

SHIP TO: (if different)
COMPANY NAME________
ADDRESS ________
CITY ________
STATE________ ZIP ________
ATTN: ________

QTY	MODEL #	DESCRIPTION	UNIT PRICE	LINE DISC	TOTAL PRICE

(Additional items can be included on second order form) [Minimum order is $50.00]

Tax Exempt #________
or Sales Tax (if applicable)

TOTAL	
Sales Tax	
Shipping	
TOTAL	

METHOD OF PAYMENT

☐ Check or money order enclosed
For orders less than $100.00

☐ Charge my ☐ Visa ☐ MasterCard
Acc't No.________ Expiration Date________

☐ Purchase Order Number:________

SHIP VIA

☐ DGC will select best way (U.P.S or Postal)

☐ Other:
☐ U.P.S. Blue Label
☐ Air Freight
☐ Other ________

NOTE: ORDERS LESS THAN $100, INCLUDE $5.00 FOR SHIPPING AND HANDLING.

Person to contact about this order ________ Phone ________ Extension ________

Mail Orders to:

Data General Corporation
Attn: Educational Services/TIPS F019
4400 Computer Drive
Westboro, MA 01580
Tel. (617) 366-8911 ext. 4032

____
Buyer's Authorized Signature Date
(agrees to terms & conditions on reverse side)

____
Title

____
DGC Sales Representative (If Known) Badge #

DISCOUNTS APPLY TO MAIL ORDERS ONLY

012-1780

educational services

CUT ALONG DOTTED LINE

DATA GENERAL CORPORATION
TECHNICAL INFORMATION AND PUBLICATIONS SERVICE
TERMS AND CONDITIONS

Data General Corporation ("DGC") provides its Technical Information and Publications Service (TIPS) solely in accordance with the following terms and conditions and more specifically to the Customer signing the Educational Services TIPS Order Form shown on the reverse hereof which is accepted by DGC.

1. PRICES
Prices for DGC publications will be as stated in the Educational Services Literature Catalog in effect at the time DGC accepts Buyer's order or as specified on an authorized DGC quotation in force at the time of receipt by DGC of the Order Form shown on the reverse hereof. Prices are exclusive of all excise, sales, use or similar taxes and, therefore are subject to an increase equal in amount to any tax DGC may be required to collect or pay on the sale, license or delivery of the materials provided hereunder.

2. PAYMENT
Terms are net cash on or prior to delivery except where satisfactory open account credit is established, in which case terms are net thirty (30) days from date of invoice.

3. SHIPMENT
Shipment will be made F.O.B. Point of Origin. DGC normally ships either by UPS or U.S. Mail or other appropriate method depending upon weight, unless Customer designates a specific method and/or carrier on the Order Form. In any case, DGC assumes no liability with regard to loss, damage or delay during shipment.

4. TERM
Upon execution by Buyer and acceptance by DGC, this agreement shall continue to remain in effect until terminated by either party upon thirty (30) days prior written notice. It is the intent of the parties to leave this Agreement in effect so that all subsequent orders for DGC publications will be governed by the terms and conditions of this Agreement.

5. CUSTOMER CERTIFICATION
Customer hereby certifies that it is the owner or lessee of the DGC equipment and/or licensee/sub-licensee of the software which is the subject matter of the publication(s) ordered hereunder.

6. DATA AND PROPRIETARY RIGHTS
Portions of the publications and materials supplied under this Agreement are proprietary and will be so marked. Customer shall abide by such markings. DGC retains for itself exclusively all proprietary rights (including manufacturing rights) in and to all designs, engineering details and other data pertaining to the products described in such publication. Licensed software materials are provided pursuant to the terms and conditions of the Program License Agreement (PLA) between the Customer and DGC and such PLA is made a part of and incorporated into this Agreement by reference. A copyright notice on any data by itself does not constitute or evidence a publication or public disclosure.

7. DISCLAIMER OF WARRANTY
DGC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE ON ANY OF THE PUBLICATIONS SUPPLIED HEREUNDER.

8. LIMITATIONS OF LIABILITY
IN NO EVENT SHALL DGC BE LIABLE FOR (I) ANY COSTS, DAMAGES OR EXPENSES ARISING OUT OF OR IN CONNECTION WITH ANY CLAIM BY ANY PERSON THAT USE OF THE PUBLICATION OF INFORMATION CONTAINED THEREIN INFRINGES ANY COPYRIGHT OR TRADE SECRET RIGHT OR (II) ANY INCIDENTIAL, SPECIAL, DIRECT OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOSS OF DATA, PROGRAMS OR LOST PROFITS.

9. GENERAL
A valid contract binding upon DGC will come into being only at the time of DGC's acceptance of the referenced Educational Services Order Form. Such contract is governed by the laws of the Commonwealth of Massachusetts. Such contract is not assignable. These terms and conditions constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes all prior oral or written communications, agreements and understandings. These terms and conditions shall prevail notwithstanding any different, conflicting or additional terms and conditions which may appear on any order submitted by Customer.

DISCOUNT SCHEDULES

DISCOUNTS APPLY TO MAIL ORDERS ONLY.

LINE ITEM DISCOUNT

5-14 manuals of the same part number - 20%
15 or more manuals of the same part number - 30%

DISCOUNTS APPLY TO PRICES SHOWN IN THE CURRENT TIPS CATALOG ONLY.

User Documentation Remarks Form

Your Name ______________________ Your Title ______________________

Company ______________________

Street ______________________

City ______________________ State __________ Zip __________

We wrote this book for you, and we made certain assumptions about who you are and how you would use it. Your comments will help us correct our assumptions and improve the manual. Please take a few minutes to respond. Thank you.

Manual Title *FORTRAN 77 Environment Manual (AOS)* Manual No. *093-000273-00*

Who are you? ☐EDP Manager ☐Analyst/Programmer ☐Other ______________________
☐Senior Systems Analyst ☐Operator ______________________

What programming language(s) do you use? ______________________

How do you use this manual? *(List in order: 1 = Primary Use)* ______________________

___ Introduction to the product ___ Tutorial Text ___ Other ______________________
___ Reference ___ Operating Guide

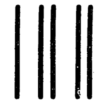
About the manual:

	Yes	Somewhat	No
Is it easy to read?	☐	☐	☐
Is it easy to understand?	☐	☐	☐
Are the topics logically organized?	☐	☐	☐
Is the technical information accurate?	☐	☐	☐
Can you easily find what you want?	☐	☐	☐
Does it tell you everything you need to know	☐	☐	☐
Do the illustrations help you?	☐	☐	☐

If you have any comments on the software itself, please contact Data General Systems Engineering.
If you wish to order manuals, use the enclosed TIPS Order Form (USA only).

Remarks:

Date

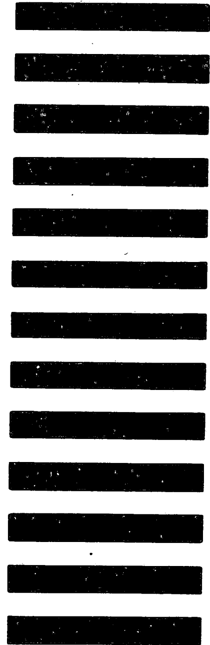


NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

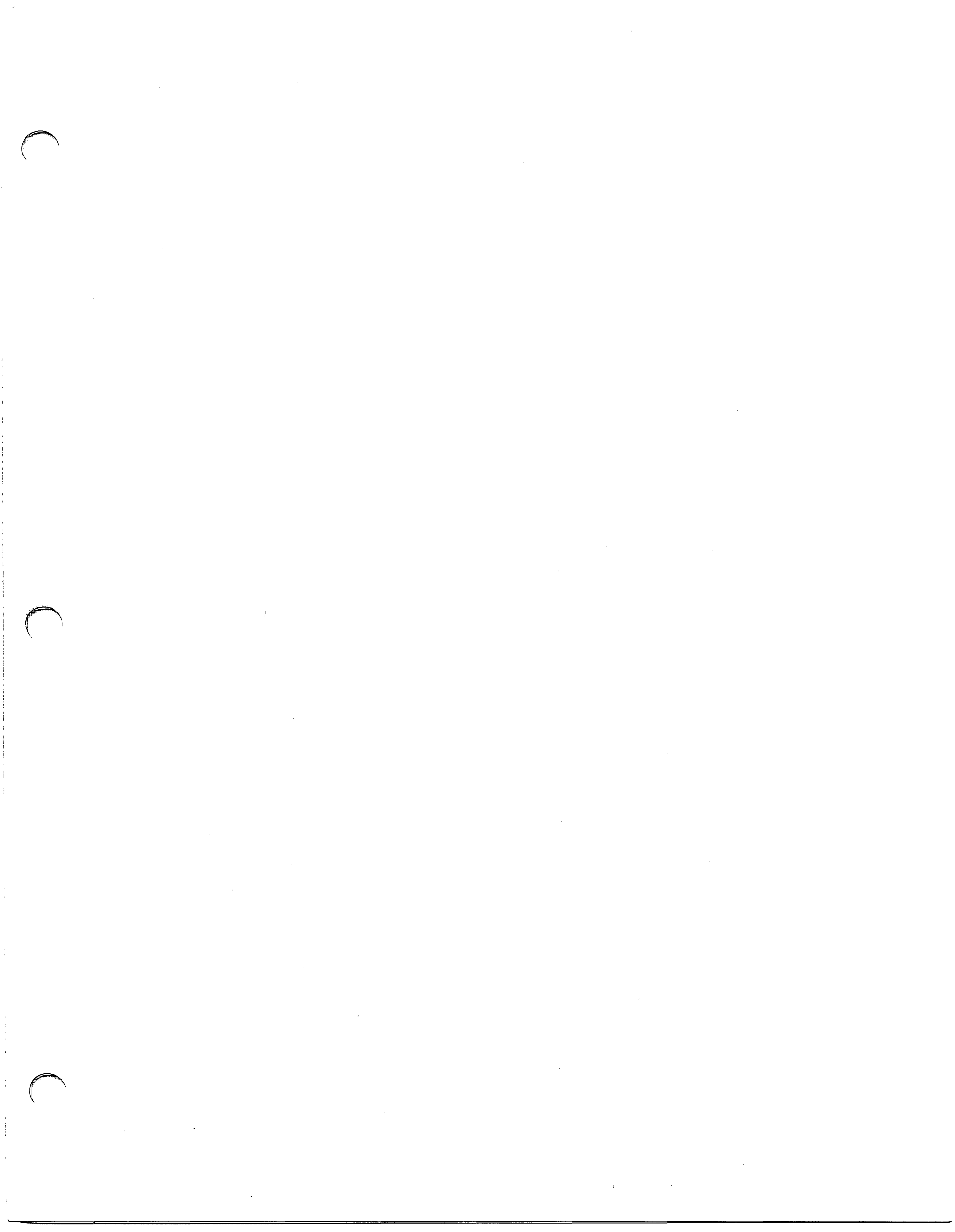
BUSINESS REPLY MAIL

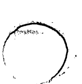
FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA. 01772

POSTAGE WILL BE PAID BY ADDRESSEE



User Documentation, M.S. E-111
4400 Computer Drive
Westborough, Massachusetts 01581





Data General Corporation, Westboro, MA 01580

093-000273-00