CDC® VSOS USER'S GUIDE FOR FORTRAN 200 PROGRAMMERS

FOR USE WITH
CDC® CYBER 200
SYSTEM COMPUTER

USER'S GUIDE

60455390

REVISION RECORD

REVISION	DESCRIPTION
A (12-14-84)	Manual released.
B (12-05-86)	Manual revised to reflect VSOS 2.3 at PSR level 670.
Publication No. 60455390	

REVISION LETTERS I, O, Q, S, X AND Z ARE NOT USED.

© 1984, 1986
by Control Data Corporation
All rights reserved
Printed in the United States of America

Address comments concerning this manual to:

Control Data Corporation
Technology and Publications Division
4201 North Lexington Avenue
St. Paul, Minnesota 55126-6198

or use Comment Sheet in the back of this manual.

LIST OF EFFECTIVE PAGES

New features, as well as changes, deletions, and additions to information in this manual, are indicated by bars in the margins or by a dot near the page number if the entire page is affected. A bar by the page number indicates pagination rather than content has changed.

PAGE	REV
Front Cover	-
Title Page	-
2	B
3/4	B
5	A
6	A
7	A
8	A
9	A
1-1	A
2-1	A
2-2	A
2-3	A
2-4	A
2-5	B
2-6	A
2-7	A
2-8	A
2-9	A
2-10	A
2-11	B
2-12	A
3-1	A
3-2	B
3-3	B
3-4	A
3-5	A
3-6	B
3-7	B
3-8	B
3-9	B
3-10	A
3-11	A
3-12	B
3-13	A
3-14	A
3-15	A
3-16	A
3-17	A
3-18	A
3-19	A
3-20	B
3-21	A
3-22	B
3-23	A
3-24	A
3-25	B
4-1	A
4-2	B
4-3	A
4-4	A
4-5	A
4-6	B
4-7	A
4-8	A
4-9	A
4-10	A
4-11	A
4-12	A
5-1	A
5-2	A
5-3	A
5-4	A
5-5	A
5-6	A
5-7	A
5-8	A
6-1	B
6-2	A
6-3	A
6-4	B
6-5	B
6-6	A
6-7	B
6-8	A
6-9	A
6-10	B
6-11	A
6-12	B
6-13	A
6-14	B
6-15	B
6-16	A
7-1	B
7-2	B
7-3	B
7-4	A
7-5	B
7-6	B
7-7	A
7-8	A
8-1	A
8-2	B
8-3	A
8-4	A
8-5	B
8-6	A
8-7	A
8-8	A
8-9	A
8-10	B
8-11	B
8-12	B
8-13	B
8-14	B
8-15	B
8-16	B
8-17	B
8-18	B
8-19	A
8-20	B
8-21	B
8-22	B
8-23	B
A-1	B
A-2	A
A-3	A
A-4	B
A-5	B
A-6	B
Index-1	B
Index-2	A
Index-3	B
Comment Sheet	B
Back Cover	-

PREFACE

AUDIENCE

This manual provides an overview of the CDC® Virtual Storage Operating System (VSOS) and an introduction to the FORTRAN 200 compiler. The manual is intended for the experienced FORTRAN programmer who is new to VSOS and to FORTRAN 200.

ORGANIZATION

The major topics discussed are as follows:

- CYBER 200 system access
- VSOS file management
- Differences between FORTRAN 5 and FORTRAN 200
- Handling FORTRAN I/O functions
- Abnormal job termination
- Optimizing job performance

RELATED PUBLICATIONS

Related information can be found in the following publications:

Publication	Publication Number
VSOS Version 2 Reference Manual, Volume 1	60459410
VSOS Version 2 Reference Manual, Volume 2	60459420
FORTRAN 200 Version 1 Reference Manual	60480200
CYBER 200 FORTRAN Language Version 2 Reference Manual	60485000
Remote Host Facility Handbook (used with NOS systems)	60459060
Remote Host Facility Handbook for IBM Systems (used with MVS/JES2, MVS/JES3, and MVT/ASP systems)	60459050
CYBER 200 Model 205 Hardware Reference Manual	60256020

DISCLAIMER

This product is intended for use only as described in this document. Control Data cannot be responsible for the proper functioning of undescribed features or parameters.

CONTENTS

1. INTRODUCTION 1-1

2. CYBER 200 SYSTEM ACCESS 2-1

CYBER 200 Access Validation 2-2
Constructing a CYBER 200 Job File 2-2
  Control Statement Group 2-4
    Job Statement 2-4
    USER Statement 2-4
    RESOURCE Statement 2-5
    Subsequent Statements 2-5
  Input Groups 2-5
Submitting the Batch Job 2-6
  Submit Errors 2-6
  Checking Job Status 2-6
  Finding Your Output File 2-7
  Job Dayfile 2-8
Batch Job Submittal 2-8
Interactive Access 2-11

3. FILE MANAGEMENT 3-1

File Types in CYBER 200 Jobs 3-1
File Use in CYBER 200 Jobs 3-2
Connecting Files to Your FTN200 Program 3-3
  Changing File Connections on the Program Execution Statement 3-4
  Job Example That Uses Input Data Twice 3-5
Using the Compile and GO Option 3-6
Drop File Space 3-7
  Specifying Drop File Size With a LOAD Statement 3-8
Using CYBER 200 Mass Storage Files 3-8
  Temporary Files 3-8
  Permanent Files 3-9
    Permanent File Entry Information 3-9
  File Ownership 3-10
    Public Files 3-10
    Private Files 3-10
      Attaching Private Files Belonging to Another User 3-10
      Permitting Other Users to Attach Your Private Files 3-11
      Giving Your Private Files to Another User 3-12
    Pool Files 3-12
Transferring Files to and From a Front-End System 3-12
  Character Data Transfers 3-13
    Character Data Transfer Example 3-14
  Binary Data Transfers 3-14
    CRM Data Delimiters 3-16
    Data Conversion 3-17
    Binary Data Transfer Example 3-19
Using a CYBER 200 File Interactively 3-20
Using Magnetic Tape Files 3-24
  Reserving Tape Drives 3-24
  Requesting a Tape File 3-24

4. FORTRAN PROGRAM CONVERSION 4-1

FORTRAN 5/FORTRAN 200 Syntax Differences 4-1
FORTRAN-Supplied Functions 4-3
  Boolean Functions 4-3
  I/O Functions 4-3
FORTRAN-Supplied Subroutines 4-4
  Subroutines With Functional Equivalents 4-5
  Subroutines Without Functional Equivalents 4-6
Product Interfaces 4-6
Machine-Dependent Differences 4-7
  Integer Representation 4-8
  Floating-Point Representation 4-9
    Converting to Normalized CYBER 200 Real Format 4-9
    Converting From CYBER 200 Normalized Real Format 4-10
    Floating-Point Zero and Floating-Point Indefinite 4-11
  Double-Precision Representation 4-11
  Complex Representation 4-12
  Logical Representation 4-12

5. FORTRAN I/O 5-1

Mixing FORTRAN Runtime and SIL I/O 5-2
Record I/O 5-4
  FORTRAN Runtime Record I/O 5-4
  SIL Record I/O 5-4
  Access Methods 5-5
Data Transfer 5-6
Block I/O 5-7

6. CYBER 200 MEMORY MANAGEMENT 6-1

Memory Paging 6-1
Program Components 6-2
Generating the Executable File 6-3
User-Controllable Page Mapping 6-10
Advising the System of Memory Requirements 6-11
Implicit I/O 6-12
System Shared Library 6-14
Controllee Files 6-15

7. ABNORMAL TERMINATION 7-1

CYBER 200 Job Termination 7-1
TV Control Statement 7-1
CYBER 200 Task Termination 7-2
User Reprieve Processing 7-2
Abnormal Termination Control 7-2
SIL Status Code Processing 7-2
Data Flag Branch Manager 7-4
System Error Processor 7-5
Debugging Tools 7-5
MDUMP Subroutine 7-5
DUMP Control Statement 7-6
LOOK Utility 7-6
DEBUG Utility 7-7

8. PROGRAM OPTIMIZATION 8-1

Scalar Processor 8-2
Register File 8-3
Instruction Stack 8-4
Scalar Optimization 8-4
Automatic Optimization 8-4
DO Loop Modification 8-5
Using Recursive DO Loops 8-5
Merging Short DO Loops 8-5
Unrolling DO Loops 8-6
Splitting DO Loops 8-7
Vector Processor 8-8
Two-Pipe and Four-Pipe Processors 8-9
Storing Arrays 8-9
Vector Optimization 8-9
Automatic Vectorization 8-10
Linked Triads 8-10
Factorizing DO Loops 8-11
Contiguity in Memory 8-12
Maximum Vector Length 8-13
Explicit Vectorization 8-13
Explicit Vector Syntax 8-14
Implicit Vector Syntax 8-15
Other Vector Functions 8-17
Vector Functions 8-17
Control Vectors 8-18
WHERE Statements 8-19
Q8 Functions 8-20
Speeding Up Subroutine Calls 8-21
Register File Swapping 8-22
Parameter Passing 8-23

APPENDIX

A. GLOSSARY A-1

INDEX

FIGURES

2-1 CYBER 200 System Access 2-1
2-2 CYBER 200 Job File 2-3
2-3 CYBER 200 Control Statement Group 2-4
2-4 Sample Batch Job Submittal 2-9
2-5 Successful Interactive Login and Logout 2-11
2-6 Unsuccussful CYBER 200 Connection Attempt 2-12
2-7 Unsuccessful CYBER 200 Login Attempt 2-12
3-1 Files Used in Job Execution 3-2
3-2 Job That Uses Job File Data Twice 3-5
3-3 Modified CYBER 200 Job File 3-7
3-4 Character Data Transfer Example 3-15
3-5 TDUMP Octal Dump Example 3-17
3-6 Binary Data Transfer Example 3-18
3-7 Data Dumps From MFLINK Job Example 3-21
3-8 Job File Modified for Interactive I/O 3-22
3-9 Interactive I/O Example 3-23
5-1 FORTRAN I/O Interface 5-1
5-2 Mixed I/O Examples 5-3
5-3 Direct Access Program Example 5-5
5-4 Explicit I/O Data Transfers 5-6
5-5 Block I/O Example 5-7
6-1 SUMMARY Output Example 6-2
6-2 Storage Map Example 6-4
6-3 Load Map Example 6-7
6-4 Task Virtual Space Mapping 6-9
6-5 Mapping Uninitialized Common to the Drop File 6-11
6-6 Implicit I/O Job Example 6-13
6-7 System Shared Library Format 6-14
6-8 Memory Allocation for Dynamic Files 6-16
7-1 DEBUG Session Example 7-8
8-1 Scalar Processor Diagram 8-2
8-2 Vector Processor Diagram 8-8
8-3 Explicit Vector Syntax 8-15
8-4 Use of the WHERE Statement 8-20

TABLES

4-1 FORTRAN 5/FORTRAN 200 Syntax Differences 4-2
4-2 FORTRAN 5 Intrinsic Functions Not Provided by FORTRAN 200 4-4

CYBER 200 SYSTEM ACCESS 2

A CYBER 200 system is always a back-end system. This means the system is accessed only through a smaller front-end computer. Figure 2-1 illustrates CYBER 200 system access.

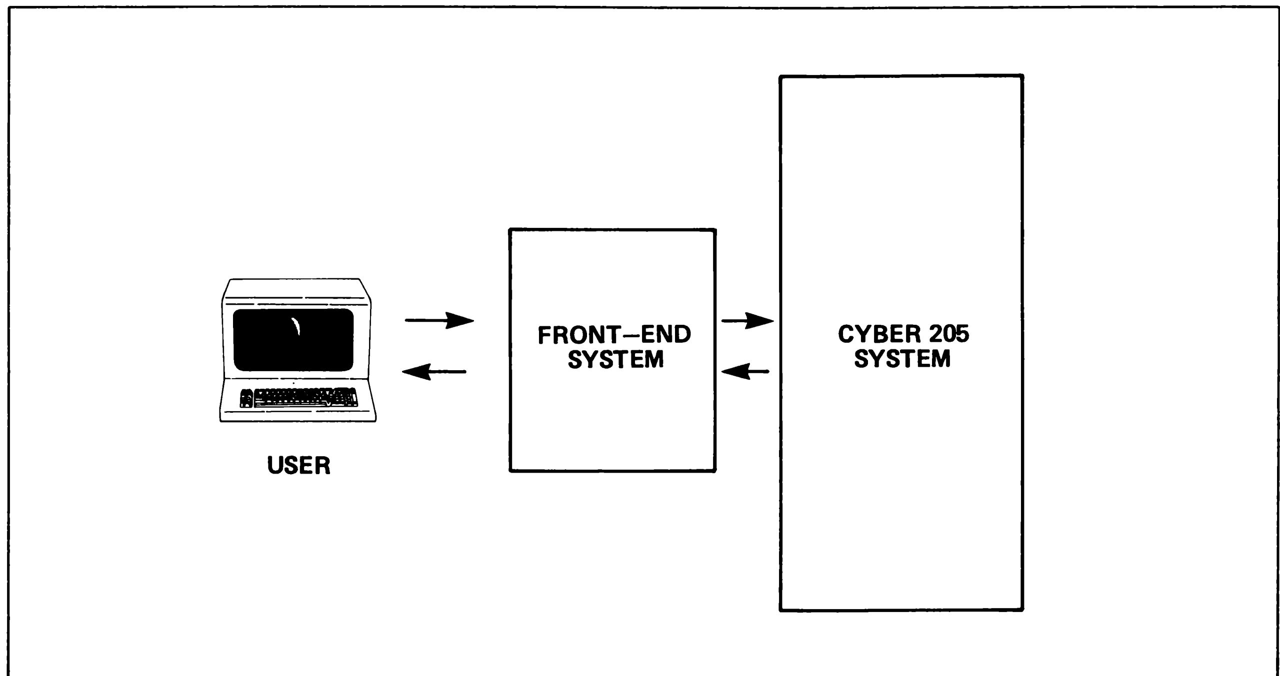


Figure 2-1. CYBER 200 System Access

A front-end computer system keeps the CYBER 200 as free as possible for the primary purpose of large-scale computation. To allow this, the front-end system performs housekeeping tasks such as file editing, job submission, and input/output handling.

A CYBER 200 system can be connected to more than one front-end system. One system frequently used as a front-end system is the CDC CYBER 170 Computer System. The operating system on the CYBER 170 computer can be either the Network Operating System (NOS), the Network Operating System/Batch Environment (NOS/BE), the Network Operating System/Virtual Environment (NOS/VE), or SCOPE. The operating system on the CYBER 200 computer is the Virtual Storage Operating System (VSOS). This manual uses CYBER 170/NOS 2 examples to illustrate front-end system use.

This manual assumes that you already know how to perform the following functions on your site's front-end system:

- Log in and log out
- Text file editing
- File storage and deletion

If you have questions in these areas, refer to the Remote Host Facility Handbook for your front-end system. (See the list of related publications in the preface.) You may also ask site personnel for help.

CYBER 200 ACCESS VALIDATION

To access a CYBER 200 system, ask site personnel for the following information:

- User validation on the front-end system if you do not already have such validation.
- The three-character logical identifier (LID) for the CYBER 200 system.
- A six-digit user number and an account name recognized by VSOS. If your site requires user password entry, you must also ask for your initial password.

This information is required to complete the following actions:

- To submit a CYBER 200 batch job
- To log in interactively to the CYBER 200 system

The job examples in this manual use representative values for these items.

CONSTRUCTING A CYBER 200 JOB FILE

Like a CYBER 170 job, a CYBER 200 job begins as a text file called a job file (figure 2-2).

The sample CYBER 200 job file in figure 2-2 contains three delimiters. These delimiters split the sequence of lines in the job file into groups.

```
ADEY,ST=ABC.
USER,USER=123456,ACCOUNT=ACCT933,PASSWORD=XYZ.
RESOURCE,TL=10.
FTN200.
LOAD.
GO.
(Delimiter)
      PROGRAM LOOP
      K = 0
      DO 10 I=1,5
      READ 100,J
  100 FORMAT(I1)
   10 K = J + K
      PRINT 200
  200 FORMAT(' THE SUM IS')
      PRINT 300,K
  300 FORMAT(1X,I2)
      STOP
      END
(Delimiter)
1
2
3
4
5
(End of file delimiter)
```

Figure 2-2. CYBER 200 Job File

The first group is the control statement group. The groups that follow the control statement group contain input requested by the control statements.

Every job requires a control statement group. Subsequent groups are optional. If the control statements do not request input, no other groups are needed, and the job consists only of the control statement group.

If a job file has more than one group, you must insert delimiters between groups. To insert a delimiter into a CYBER 200 job file, use a text editor on the front-end system to insert an end-of-record indicator at the appropriate point in the text. When you transfer the job file to the CYBER 200 system for execution, each end-of-record indicator is converted to the ASCII GS character (hexadecimal code 1D) that the CYBER 200 system recognizes as an end-of-group delimiter.

CONTROL STATEMENT GROUP

The control statement group is the first group in a job file. All control statements for a job are in this group. The control statement group contains only control statements; it contains no data or directives.

Figure 2-3 lists the control statements from the job file in figure 2-2. Generally, control statements are executed in the order in which they appear in the group.

```
ADEY,ST=ABC.
USER,USER=123456,ACCOUNT=ACCT933,PASSWORD=XYZ.
RESOURCE,TL=10.
FTN200.
LOAD.
GO.
```

Figure 2-3. CYBER 200 Control Statement Group

Job Statement

The first statement in the control statement group is the job statement (ADEY,ST=ABC in figure 2-3). The software that transfers the job file uses the job statement. This statement specifies two items of information: the name you give the job (ADEY) and the logical identifier of the CYBER 200 system to which the job file is transferred (ST=ABC). This logical identifier (LID) is obtained from site administration.

USER Statement

The second statement in the control statement group must be the USER statement. The user statement contains the information VSOS needs to determine if you are a valid user.

The USER statement specifies your user number, your account name, and, if necessary, the password and security level for your job. The user number and the account name are required.

Your site determines whether the USER statement requires a password and a security level specification. The USER statement in figure 2-3 specifies a user number (USER=123456), an account name (ACCOUNT=ACCT933), and a password (PASSWORD=XYZ). The sample USER statement assumes that the default security level (level 1) is appropriate. For more information about security levels, refer to the VSOS Reference Manual, Volume 1.

RESOURCE Statement

You can use the RESOURCE statement to specify resource constraints under which VSOS executes your job. The RESOURCE statement is optional. If you include a RESOURCE statement in your job, it must be the control statement following the USER statement.

The RESOURCE statement can specify time limit, job category, working set size, and the large page limit for a batch job. It can also reserve tape drives for use by a job. The RESOURCE statement in figure 2-3 specifies a job time limit of 10 seconds (TL=10). If you do not specify a time limit, the default job time limit is determined by site administration (it is site dependent).

Subsequent Statements

When VSOS is ready to execute the job, the batch processor portion of VSOS (called BATCHPRO) executes the sequence of tasks indicated by the statements following the RESOURCE statement. The tasks executed for the sample job in figure 2-3 are named FTN200, LOAD, and GO.

When BATCHPRO reads a task name, such as FTN200, it searches for an executable file having that name. When searching for a file, BATCHPRO searches the files attached to the batch job or to the interactive session in order by ownership category. First, BATCHPRO searches private (local, attached permanent) files. Next, it searches pool files in reverse order by which the pools were attached. Finally, BATCHPRO searches the system public files. The system public files may contain assemblers, compilers, and other general purpose routines.

It is assumed that the statement sequence FTN200, LOAD, and GO intends to sequentially execute the FORTRAN 200 compiler in public file FTN200, the program loader in public file LOAD, and finally the object program prepared by the loader on file GO.

INPUT GROUPS

For a batch job, file INPUT is the entire job that is submitted for processing. The system uses the first group of file INPUT, which contains the control statements. The commands use the data contained in the remaining groups.

In the sample job, the FTN200 task, by default, requests input from file INPUT. BATCHPRO provides the task with the source program in the second group of the job file. The next task, which is LOAD, does not require input from file INPUT. However, the object program that is compiled by FTN200 and prepared for execution by LOAD on file GO does require input from the INPUT file. (The program contains a FORTRAN-formatted READ statement that, by default, reads from file INPUT.) Therefore, BATCHPRO provides the object program with the data from the third group in the job file.

Source programs and input data may be read from files other than file INPUT. When no task in a job requires input from file INPUT, the job file contains only the control statement group.

SUBMITTING THE BATCH JOB

When your CYBER 200 job file is ready, you can submit the job for execution using one of the following commands:

- SUBMIT,filename,E
- ROUTE,filename,DC=IN,UN

If you enter the SUBMIT command, NOS submits the job to the remote input queue. When complete, NOS returns the following message:

```
17.08.18 SUBMIT COMPLETE.  JSN IS ABQP.
```

This NOS message gives the time that the job was submitted (17.08.18) and the job sequence name (JSN IS ABQP) that identifies your job in the NOS input queue.

If you use the ROUTE command, NOS routes your files according to the parameters specified on the ROUTE command. When the routing is completed, NOS returns the following message:

```
ROUTE COMPLETE.  JSN IS ABQP.
```

SUBMIT ERRORS

If the logical identifier (LID) specified on the first line of the job file is invalid, a submit attempt fails. In this case, NOS responds to the SUBMIT command with the following message:

```
DSP - INVALID LID
```

To clear this error, correct the logical identifier and resubmit the job.

A submit attempt also fails if the submitted job will exceed your deferred batch jobs limit. If submission of an additional job will exceed this limit, NOS responds to the SUBMIT command with the following message:

```
DSP - TOO MANY DEFERRED BATCH JOBS.
```

To determine your deferred batch job limit, enter a LIMITS command and look for the number listed after the words DEFERRED BATCH FILES. This number is the maximum deferred batch jobs you can execute concurrently.

CHECKING JOB STATUS

To see the status of all your jobs, enter one of the following commands:

- ENQUIRE,JSN
- ENQUIRE,UJN

NOS then displays the job sequence name and the status of all jobs associated with your user name. The display indicates whether the job is in the input queue, is in the print queue, or is executing. (Only NOS jobs can be listed as executing. Executing CYBER 200 jobs are not shown on the NOS job status display.)

Command ENQUIRE,JSN could return the following display:

```
JSN SC CS DS LID STATUS
ABQP.B.  .RB.ABC.INPUT QUEUE
```

Where:

Job ABQP is the job you submitted to logical identifier ABC.

Command ENQUIRE,UJN could return the following display:

```
JSN SC CS DS LID UJN    STATUS       EXECUTING MESSAGE
ABQP.T.ON.BC.ABC.MYJOB  EXECUTING
```

Where:

MYJOB is the name you assigned to your job. This name uniquely identifies your job for input and appears on the banner page of your job's output as UJN=MYJOB.

FINDING YOUR OUTPUT FILE

The following ENQUIRE,JSN display indicates there is a job in the output queue:

```
JSN SC CS DS LID STATUS
ABQZ.R.  .RB.C17.PRINT QUEUE
```

NOTE

The JSN of this job is not the JSN of the job that was in the input queue. Nevertheless, ABQZ is the job output file written by the submitted job.

If the CYBER 200 system has accepted your job from the remote input queue and executed it, the output file contains at least the job dayfile. If the CYBER 200 system is not accepting jobs, however, the output file contains only the following message:

```
USER 12306 - FILE NOT WANTED BY ALL REMOTE HOSTS, QTF.
```

JOB DAYFILE

The last record in the job output file contains the job dayfile. The dayfile lists each control statement executed followed by messages describing the execution of the task.

When looking at your dayfile, you should determine whether any task returned an error. An error message for a task specifies a return code as follows:

- Return code 8 = fatal error
- Return code 4 = nonfatal error
- Return code 0 = no errors

Unless you use a TV control statement to specify otherwise, a fatal error causes control to be passed to an EXIT statement or causes abnormal termination of the job. (The TV control statement is described in section 7, Abnormal Termination. The EXIT statement is explained in section 3, File Management.)

BATCH JOB SUBMITTAL

Figure 2-4 shows an interactive session in which the user submits a CYBER 200 batch job. The job file is on a NOS indirect access file named JOBFILE. The user calls the text editor full screen editor (FSE) to display the job file contents, exits from FSE, and submits the job for execution.

```
84/09/20. 13.06.48. L25T1                          <-- NOS login banner.
(00) CYBER 73 S/N 101  CYBER 200 CLSH.  NOS 2-620/587-12.
FAMILY:     ,987654,xpwxpwx,IAF                    <-- NOS login with IAF request.
JSN:  ABQW, NAMIAF                                 <-- Interactive Access Facility
                                                       (IAF) connection is successful.

/batch                                             <-- Switches NOS to batch mode.
RFL,0.
/get,jobfile                                       <-- Gets the file JOBFILE.
/fse,jobfile                                       <-- Calls full screen editor to
 NOS FULL SCREEN EDITOR                                display the job file contents.
?? pa                                              <-- Prints all of file.
ADEY,ST=ABC.
USER,USER=123456,ACCOUNT=ACCT933,PASSWORD=XYZ.
RESOURCE,TL=5.
FTN200.
LOAD.
GO.
(EOR)
      PROGRAM LOOP
      K = 0
      DO 10 I=1,5
      READ 100,J
  100 FORMAT(I1)
   10 K = J + K
      PRINT 200
  200 FORMAT(' THE SUM IS')
      PRINT 300,K
  300 FORMAT(1X,I2)
      STOP
      END
(EOR)
1
2
3
4
5
(EOF)
 ?? qr                                             <-- Exits from full screen editor.
JOBFILE IS A LOCAL FILE
/submit,jobfile,e                                  <-- Submits the job for execution.
 13.08.17  SUBMIT COMPLETE.  JSN IS ADYF
```

Figure 2-4. Sample Batch Job Submittal (Sheet 1 of 2)

```
/enquire,jsn                                           ←— Requests job status.
 JSN SC CS DS LID STATUS        JSN SC CS DS LID STATUS
 ADYF.B.  .RB.ABC.INPUT QUEUE   ABQW.T.ON.BC.    .EXECUTING
/enquire,jsn
 JSN SC CS DS LID STATUS        JSN SC CS DS LID STATUS
 ABQW.T.ON.BC.   .EXECUTING                            ←— Job sent to CYBER 205; thus,
 /enquire,jsn                                              does not display on NOS queue.
 JSN SC CS DS LID STATUS        JSN SC CS DS LID STATUS
 ADYZ.B.  .RB.ABC.PRINT QUEUE   ABQW.T.ON.BC.    .EXECUTING
/qget,adyz,pr                                          ←— Output file becomes local file.
 QGET COMPLETE.
/fse,adyz                                              ←— Calls a text editor to display
 NOS FULL SCREEN EDITOR                                    the output file.
?? l/adey                                              ←— Locates the job name.
(EOR)                                                      at the top of dayfile.

1 13.06.31  RSYSL620       VSYSL620       987654  PUBLIC    G    ADEY
   09/20/84
?? pa                                                  ←— Prints entire dayfile.
1 13.06.31  RSYSL620       VSYSL620       987654  PUBLIC    G    ADEY
   09/20/84
  13.06.32 RESOURCE,TL=5.
  13.06.32 FTN200.
  13.06.35 FORTRAN 200 CYCLE L607     BUILT 03/21/84 14:31
  13.06.37 COMPILING LOOP
  13.06.37       NO ERRORS
  13.06.46     0.049 SECONDS COMPILATION TIME
  13.06.50  ALL DONE
  13.06.52 LOAD.
  13.06.55  LOAD R2.1 CYCLE L592
  13.07.03  ALL DONE
  13.07.05 GO.
  13.07.10 STOP
  13.07.13  ALL DONE
  13.07.14 SYSTEM TIME UNITS (STU)                 5.188
  13.07.14 $$COMPLETE$$
(EOF)
?? l/the sum is/2                                      ←— Locates the two occurences of
    00008      200 FORMAT(' THE SUM IS')                   the text string.
                                    0001/00008
(EOR)
(EOR)
?? pa                                                  ←— Prints entire output.
 THE SUM IS
 15
?? qr                                                  ←— Exits full screen editor.
ADYZ IS A LOCAL FILE
/route,adyz,dc=lp                                      ←— Routes the output file to a
 ROUTE COMPLETE.                                           printer.
/bye                                                   ←— Logout request.

UN=987654    LOG OFF    13.23.10.
JSN=ABQW      SRU-S        1.693

IAF     CONNECT TIME:  00.15.46.                       ←— Logout response.
LOGGED OUT.
```

Figure 2-4. Sample Batch Job Submittal (Sheet 2 of 2)

INTERACTIVE ACCESS

This manual assumes that your primary access method to the CYBER 200 system is through batch job submission. If permitted by your site, however, you can also log in to the CYBER 200 system from a front-end system and execute tasks interactively. Most of the control statements described in this manual for use in a batch job can also be used interactively. The VSOS Reference Manual, Volume 1 fully describes control statement use in batch and interactive jobs.

Figure 2-5 shows an interactive session where the user first logs in to the NOS front-end system, then logs in to the CYBER 200 system, and finally logs out. Figure 2-6 shows an interactive session where the user enters an invalid CYBER 200 mainframe identifier (LID). Figure 2-7 shows an interactive session where the user enters two invalid login lines.

```
84/09/10. 13.06.48. L25T1                        <-- NOS login banner.
(00) CYBER 73 S/N 101  CYBER 200 CLSH.  NOS 2-602/587-12.
FAMILY:    ,987654,xpwxpwx                       <-- NOS login.
JSN:  ABQW, NAMIAF                               <-- IAF connection is successful.

/hello,itf                                       <-- Requests Interactive Transfer
                                                     Facility (ITF) connection.

UN=987654    LOG OFF    13.23.10.
JSN=ABQW      SRU-S          1.693
IAF     CONNECT TIME:  00.15.46.                 <-- Disconnect from IAF.

ITF 1.0 - 596
Terminal T1335, Connection 3

Enter LID (or ?): abc                            <-- Valid logical ID entry.
[ITF,connected to host ABC on ACN 1/TCN4.]
 PLEASE ENTER CY200 LOGIN                        <-- Successful link to VSOS.
login,123456,acct933,xyz                         <-- CYBER 200 login line.
VSOS 2.3   RSYS620   VSYS620   G 09977.0207 ACTIVE JOBS   0   <-- Banner for
                                                     CYBER 200 operating system.
NIL                                              <-- Indicates no files for user.
 bye                                             <-- CYBER 200 logout request.
BYE                                              <-- Logout response.
[ITF, terminal connection ended by host ABC.]
Enter LID (or ?): bye                            <-- ITF logout request.

ITF     CONNECT TIME:  00.02.52.                 <-- ITF logout response.
LOGGED OUT.
```

Figure 2-5. Successful Interactive Login and Logout

```
/hello,itf                                         ←— Command entered within a NOS
                                                       interactive session.

UN=987654    LOG OFF     13.23.10.
JSN=ABQW      SRU-S          1.693
IAF     CONNECT TIME:  00.15.46.                   ←— Disconnect from IAF.

ITF 1.0 - 596
Terminal T1335, Connection 3

Enter LID (or ?): bcd                              ←— Invalid logical ID entry.
[ITF, BCD is not a defined Lid.]
Enter LID (or ?): ?                                ←— User enters a ?.
   Enter the 3-character Logical IDentifier of the
   remote host to which you wish to connect,
or enter BYE     to LOGOUT,
or enter BYE,IAF to return to IAF,
or enter (CR)    to connect to host ABC.
ENTER LID (or ?): bye                              ←— ITF logout request.
ITF     CONNECT TIME:  00.02.52.                   ←— ITF logout response.
LOGGED OUT.
```

Figure 2-6. Unsuccessful CYBER 200 Connection Attempt

```
/hello,itf                                         ←— ITF connection request.

UN=987654    LOG OFF     13.23.10.
JSN=ABQW      SRU-S          1.693
IAF     CONNECT TIME:  00.15.46.                   ←— Disconnect from IAF.

ITF 1.0 - 596
Terminal T1335, Connection 3

Enter LID (or ?): abc                              ←— Valid logical ID entry.
[ITF,connected to host ABC on ACN 1/TCN4.]
 PLEASE ENTER CY200 LOGIN                          ←— CYBER 200 login request.
login                                              ←— Login line without all
LOGIN FORMAT ERROR                                     required parameters.
[ITF, terminal connection ended by host ABC.]
Enter LID (or ?): abc                              ←— Valid logical ID entry.
[ITF,connected to host ABC on ACN 1/TCN4.]
 PLEASE ENTER CY200 LOGIN                          ←— CYBER 200 login request.
login,m12345,acct933,xyz                           ←— Invalid login information.
BAD LOGIN
[ITF, terminal connection ended by host ABC.]
Enter LID (or ?): bye                              ←— ITF logout request.

 ITF     CONNECT TIME:  00.02.52.                  ←— ITF logout response.
LOGGED OUT.
```

Figure 2-7. Unsuccessful CYBER 200 Login Attempt

FILE MANAGEMENT 3

This section describes file types and explains how you can use files within your CYBER 200 job. The chapter uses a simple CYBER 200 job example to outline file use. The section then focuses on the following topics:

- Connecting files to your FTN200 program
- Managing drop file space
- Using CYBER 200 mass storage files
- Transferring files to and from a NOS front-end system
- Using CYBER 200 files interactively

FILE TYPES IN CYBER 200 JOBS

VSOS recognizes the following four types of files:

- Controllee files
- Data files
- Drop files
- Output files

A controllee file is an executable file. The controllee file contains all the information needed to execute the object code within a file. The LOAD utility generates the controllee file. (For FORTRAN 200 programs, you may also use the compile and GO feature to completely bypass the LOAD statement. This option is discussed later in this section.)

A data file is any CYBER 200 file that is not a controllee file and not a drop file. A data file is not executable.

A drop file is created by the system. When a task references virtual address space that is not associated with the controllee file or another mass storage file, the system associates the virtual address space with the task drop file. Modified pages of the controllee file are also written to the drop file.

Output files contain data to be processed by an output device.

FILE USE IN CYBER 200 JOBS

The CYBER 200 operating system, VSOS, is a task-oriented system. A task is defined as an execution of a controllee file for a user number. A job is simply a sequence of tasks. The primary means of communication between tasks and jobs is via file I/O.

In general, when a task ends, the results of the task processing are contained in the files the task wrote. Subsequent tasks in the job can read those files. When the job terminates, the results of job processing are either in the output file printed for the job or stored in permanent files.

To illustrate this, figure 3-1 shows the sequence of files used to execute a job designed to execute the following task sequence:

```
FTN200.
LOAD.
GO.
```

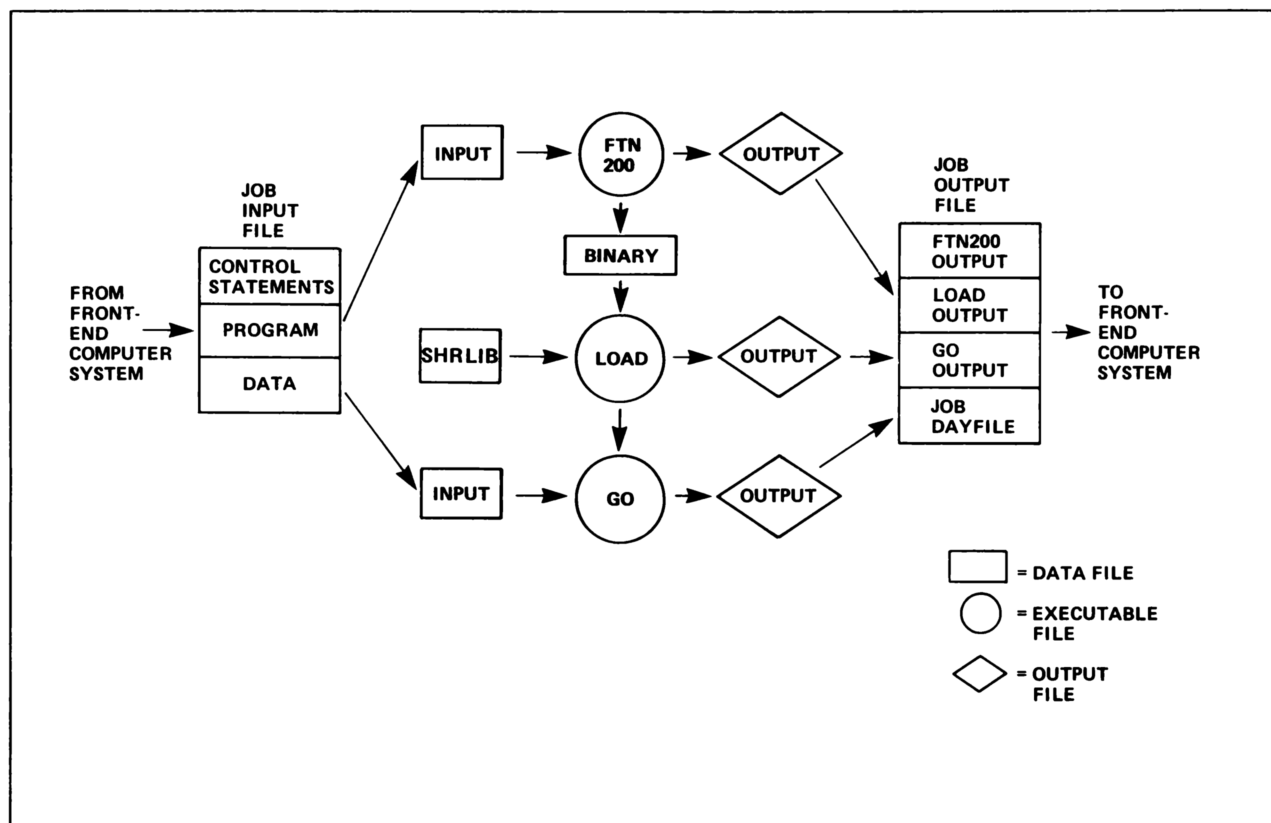


Figure 3-1. Files Used in Job Execution

The FTN200 task, by default, reads its input from file INPUT. By default, the task writes two forms of output: a binary file on file BINARY and a listing and error file on file OUTPUT.

The LOAD task, by default, processes the input from file BINARY. LOAD also reads SYSLIB to satisfy unsatisfied external references in the object modules. The LOAD task writes two forms of output: an executable program on file GO and a load map on file OUTPUT.

The GO task reads and writes files as specified in the FTN200 source program. If no other file names are specified, the GO task also, by default, reads its input from file INPUT and writes its output on file OUTPUT.

As the task terminates, BATCHPRO discards the file named INPUT. The file named OUTPUT is copied to the job output file and is then discarded. This does not mean that a task cannot read from the same input file as a previous task or read the output file of an earlier task. Figure 3-2 contains an example of a job that uses input data twice.

When a job terminates, the job dayfile is copied to the job output file which is then routed, by default, to the front-end system from which the job came.

CONNECTING FILES TO YOUR FTN200 PROGRAM

A file is useful only when a task can read from or write to the file. For your FTN200 program to read or write a file, the program must declare the file name and connect it to a unit identifier.

As an example, the following PROGRAM statement first declares the file name to be MYFILE and then associates unit identifier 1 with the file named MYFILE. A subsequent READ statement specifies unit identifier 1.

```
PROGRAM LOOP(MYFILE, UNIT1=MYFILE)
          :
          :
READ(UNIT=1,FMT=100,END=20) J
```

A file name can be connected to more than one unit identifier, but a unit identifier can be connected to only one file name. For example, a PROGRAM statement can contain both UNIT1=MYFILE and UNIT2=MYFILE. However, the statement PROGRAM cannot contain both UNIT1=MYFILE and UNIT1=HERFILE.

The unit identifiers UNITn and TAPEn are special because they connect a file name to more than one unit identifier. For example, the connection UNIT1=MYFILE allows subsequent READ statements to reference MYFILE as unit 1, unit 5HUNIT1, and unit 5HTAPE1. The connection TAPE1=MYFILE allows subsequent READ statements to reference the file as unit 1 and unit 5HTAPE1.

Initially, the system uses the file names and the unit identifier connections specified on the PROGRAM statement of your program. The control statement that executes your program however, may change or add to these file names and connections. Also, during task execution, an OPEN statement may perform additional file connections.

CHANGING FILE CONNECTIONS ON THE PROGRAM EXECUTION STATEMENT

Using the PROGRAM statement to connect files allows you change file connections when a program is executed.

NOTE

File connections specified on OPEN statements cannot be changed at execution time.

For example, a program contains the following PROGRAM statement associating the file name MYFILE with the unit identifier 1:

```
PROGRAM LOOP(MYFILE,UNIT1=MYFILE)
```

First, assume that when the program executes, you want the program to read from file INPUT1 instead of file MYFILE. If LOAD writes the program on file GO, the following control statement executes the program and changes the file connection:

```
GO,**INPUT1,UNIT1=INPUT1.
```

The two asterisks (**) before the file specification indicate that what follows replaces the entire PROGRAM statement connection specifier list (MYFILE,UNIT1=MYFILE). Each statement that reads from unit identifier 1 (or 5HUNIT1 or 5HTAPE1) reads from file INPUT1 instead of file MYFILE.

Next, assume that program LOOP writes to file OUTPUT. (Files INPUT and OUTPUT need not be declared or connected.) If you want the program to read from file MYFILE as specified on the PROGRAM statement, but you want the program to write to file LISTFIL instead of file OUTPUT, you concatenate the new file declaration and connection to the PROGRAM statement as follows:

```
GO,LISTFIL,OUTPUT=LISTFIL.
```

Without the two asterisks (**), the file specification LISTFIL,OUTPUT=LISTFIL is concatenated to the PROGRAM statement specifications.

JOB EXAMPLE THAT USES INPUT DATA TWICE

Earlier, it was stated that generally no two tasks can read the same input group from file INPUT. The job listed in figure 3-2, however, shows how two tasks can read the same job file data.

Except for files named OUTPUT, your program can read data from a file written by an earlier task in the same job. A file written by an earlier task remains assigned to the job until the file is explicitly returned or the job terminates.

To use the data in an input group twice, copy the data to a temporary file before executing the first program. Assuming both programs are written to read from the same data in file INPUT, the job file appears as shown in figure 3-2.

Note that the sequence of the input groups must match the sequence of the control statements that read from file INPUT; that is, the two FTN200 statements and the COPY statement.

```
ADEY,ST=ABC.
USER,USER=123456,ACCOUNT=ACCT933,PASSWORD=XYZ.
RESOURCE,TL=20.               <-- Longer time limit needed for two programs.

FTN200.                       <-- Compiles program in first input group.

LOAD.                         <-- Loads first program on file GO.

COPY,INPUT,DATA1.             <-- Copies data in the second input group
                                  to a temporary file named DATA1.

GO,DATA1,INPUT=DATA1.         <-- Executes GO with data from DATA1.

FTN200,B=BINARY2.             <-- Compiles program in third input group.

LOAD,BINARY2,CN=GO2.          <-- Loads the second program on file GO2.

GO2,DATA1,INPUT=DATA1.        <-- Executes GO2 with data from DATA1.

(Delimiter)

      Source for first FORTRAN program

(Delimiter)

      Data for both programs

(Delimiter)

      Source for second FORTRAN program

(End of file)
```

Figure 3-2. Job That Uses Job File Data Twice

The following summarizes the files (other than public files) read and written by the job in figure 3-2:

Task	Files Read	Files Written
First FTN200	INPUT (containing first program)	OUTPUT, BINARY
First LOAD	BINARY	OUTPUT, GO
COPY	INPUT (containing data)	DATA1
GO	DATA1	OUTPUT
Second FTN200	INPUT (containing second program)	OUTPUT, BINARY2
Second LOAD	BINARY2	OUTPUT, GO2
GO2	DATA1	OUTPUT

Note that, unlike a similar NOS job, the CYBER 200 job does not require a REWIND statement to reuse the DATA1 file. When each task begins, all disk and interactive files are positioned at the beginning of information.

USING THE COMPILE AND GO OPTION

Up to this point, the examples have used the LOAD statement, but there is another option. The FORTRAN 200 compiler allows you to skip the LOAD statement by using the compile and GO option. To alter the job shown in the previous example, consider the following control statement:

```
FTN200,GO=1.
```

When you specify GO=1, your FORTRAN program executes upon completion of a nonfatal compilation. Your program executes using the system shared library and the LINKER utility.

The system shared library is a file that contains directories, shared utilities, a shared SYSLIB, and a dynamic LINKER utility. The system shared library file allows users to share the same pages of virtual memory.

If you choose the compile and GO option when working interactively, you must create two files (INPUT and Q5INPUT) before beginning the compile of your program. INPUT is used by the compiler, and Q5INPUT contains runtime data. When working interactively, the compile and GO option sends the compiler listing to file Q5OUTPUT and your program output to file OUTPUT.

For batch jobs, the GO task reads and writes files as specified in the FTN200 source program. File INPUT is used for your program source. File Q5INPUT contains the program data, if any. After the program is compiled, the FORTRAN 200 compiler returns the INPUT file and renames the Q5INPUT file as INPUT. As each of the tasks terminates, BATCHPRO discards the INPUT file. For batch jobs, BATCHPRO then copies the OUTPUT and the Q5OUTPUT files to the job output file.

Figure 3-3 illustrates the CYBER 200 job file example shown in figure 2-2 modified for use with the compile and GO option.

```
ADEY,ST=ABC.
USER,USER=123456,ACCOUNT=ACCT933,PASSWORD=XYZ.
RESOURCE,TL=10.
FTN200,GO=1.
(Delimiter)
      PROGRAM LOOP
      K = 0
      DO 10 I=1,5
      READ 100,J
  100 FORMAT (I1)
   10 K = J + K
      PRINT 200
  200 FORMAT (' THE SUM IS')
      PRINT 300,K
  300 FORMAT (1X,I2)
      STOP
      END
(Delimiter)
1
2
3
4
5
(End of file delimiter)
```

Figure 3-3. Modified CYBER 200 Job File

DROP FILE SPACE

A drop file is space used to store all modified pages that are not already assigned space in a disk file. Because the CYBER 200 computer is a virtual memory machine, all memory assigned to a task must have corresponding disk file space assigned to it. This is required because as VSOS reassigns physical memory, VSOS stores the current contents of the physical memory on disk. When the task later references this stored data, VSOS automatically recopies the data into physical memory and assigns that memory to the task.

VSOS creates a drop file when it begins task execution and extends the drop file as more space is needed during task execution. These extensions are automatic, and no messages are printed indicating an extension has occurred.

The name the system gives to the drop file is made up of the following two parts:

- A single decimal digit (1 - 9)
- The file name

SPECIFYING DROP FILE SIZE WITH A LOAD STATEMENT

If you know your job uses only a very small drop file, you can specify an initial drop file size on the LOAD control statement. This specification minimizes the size of the extensions and reduces system overhead. You may use the DFL parameter to specify drop file size. The following LOAD statement, for example, specifies an initial drop file size of 200 blocks:

```
LOAD,DFL=200.
```

USING CYBER 200 MASS STORAGE FILES

The CYBER 200 system allows users to create and to access mass storage files. A mass storage file consists of space allocated on CYBER 200 disk units. Space is assigned to a file in segments. A segment is contiguous file space. The first segment is allocated when the file is created. The other segments are allocated as needed when the file is extended as it is written.

In general, a task creates the file to which it writes if the file does not already exist as an attached permanent file. For example, a FTN200 task creates the file on which it writes the object program if the file does not already exist.

TEMPORARY FILES

You can create both temporary and permanent files within a job. Temporary files are those files that are no longer accessible to you or to another user after the job that created them is finished.

To create a temporary mass storage file, use the REQUEST statement. For example, the following control statement creates a temporary mass storage file named TEMPFIL whose initial length is 50 blocks.

```
REQUEST,TEMPFIL/50.
```

The REQUEST statement has many other optional parameters. For their descriptions, refer to the VSOS Reference Manual, Volume 1.

After a task has been completed successfully and has written the results to file TEMPFIL; your job can change the temporary file, TEMPFIL, to a permanent file so that the task results are available to other jobs. The following DEFINE statement changes TEMPFIL to a permanent file:

```
DEFINE,TEMPFIL.
```

If the task that uses the data written on TEMPFIL is in the same job as the task that created TEMPFIL and if that task completes successfully, there may be no need to keep TEMPFIL file space.

```
RETURN,TEMPFIL.
```

A job is not required to discard its temporary files explicitly. If a job does not discard a temporary file, VSOS discards the file when the job terminates.

PERMANENT FILES

A permanent file is a mass storage file that continues to exist until the user explicitly destroys it. The file remains stored after the job that created it terminates. A permanent file is entered in the disk's permanent file directory.

Each disk unit has a directory that describes all permanent files on that disk unit. If a file has an entry in the permanent file directory, its space and characteristics remain defined as long as the entry exists. Because this entry is independent from any job and independent from system execution, the entry continues to exist after the job using the file terminates and even after system execution terminates.

A job can use a permanent file only if the file is accessible to the job. Attaching a permanent file makes the file accessible and provides the job with the information contained in the permanent file entry. A permanent file created by a job is implicitly attached to the job. A permanent file remains attached to a job until the job returns the file or the job terminates.

Permanent File Entry Information

The AUDIT control statement lists information about permanent mass storage files. Suppose a control statement group ends with this control statement sequence:

```
GO.
EXIT.
DEFINE,GO.
AUDIT,GO.
```

Assuming that GO is a local file and that the job terminates abnormally, VSOS executes the DEFINE and AUDIT statements that follow the EXIT statement. The DEFINE statement saves the executable file (GO) as a permanent file. The AUDIT statement produces the following output:

```
NAME   OWNER   TY  FC  RT  BT  ACS  EXT  SL  DEVICE  DSET     FLEN
 GO    123456  VC   U  U    C   RW   SX   1  PACK01  DVST01    98
```

The AUDIT produces a partial listing of the contents of the permanent file entry for file GO. For a full listing, specify LO=F on the AUDIT statement. Refer to the VSOS Reference Manual, Volume 1, for a complete description of the AUDIT listing.

In addition to the AUDIT statement, you may also use the FILES control statement to list information about a file. Using the previous example, FILES produces the following output:

```
NAME   ACS   LEN     JDN  ORI.DATE   OWNER    TYPE  DT  FC  BT  RT  FO
 GO    XR   002656-  124  08/30/84   *123456  VC    MS   U   C   U   S
```

Like AUDIT, FILES also produces a partial listing of the contents of the permanent file entry for file GO.

FILE OWNERSHIP

A permanent file entry records the owner of a file. Each file has an owner who controls file use.

There are three file ownership categories in the CYBER 200 permanent file system. These categories are as follows:

- Public files
- Private files
- Pool files

Public Files

Public files are the set of system files accessible to all users. These files are automatically attached to each executing job. Site personnel determine which files are public files. All public files belong to user number 000000, which indicates system ownership. Compilers, loaders, and other system utilities are examples of public files.

Private Files

A private file is a permanent file that belongs to the user number of the job that created the file. By default, owners can access their private files; however, no other user can access these files unless the owner grants access.

A private file is created with a DEFINE statement. The file is accessed with an ATTACH statement, detached with a RETURN statement, and destroyed with a PURGE statement.

A RETURN statement ends the attachment of the permanent file to the job, but the file continues to exist. A PURGE statement deletes the permanent file entry and releases the file space when the file is detached. Only a job executing for the user number that created the file can purge a private file.

Attaching Private Files Belonging to Another User

If a task in your job reads from a private file that belongs to another user, the other user must grant your user number access to the file. Once you have access to another user's private file, you can attach it to your program with an ATTACH statement.

For example, a task in your job reads a private file named COMDATA, which belongs to user 987654. If user 987654 grants your user number access to file COMDATA, the following statement attaches the file to your job:

```
ATTACH,COMDATA,USER=987654.
```

If you are not sure whether you have access to file COMDATA, you can list the files belonging to user 987654 that you can attach by executing this statement:

```
FILES,USER=987654.
```

The FILES statement lists the files belonging to user 987654 that you can access. If user 987654 has not granted you access to any files, the FILES output message is as follows:

```
FILES NOT FOUND.
```

Permitting Other Users to Attach Your Private Files

To permit other users to attach your private files, you must define one or more access permission sets for a file. The PERMIT statement defines an access permission set. A general access permission set is an access permission set that applies to all users. An individual access permission set applies to a single user. You can also define the type of access you want to give to another user: read (R), write (W), execute (X), append (A), modify (M), or none (NONE).

To permit all users to read your private file TESTDATA, execute the following statement:

```
PERMIT,TESTDATA,USER=GENERAL,ACCESS=R.
```

To permit user 987654 to read file TESTDATA, execute the following statement:

```
PERMIT,TESTDATA,USER=987654,ACCESS=R.
```

To deny access to user 776655 after having previously granted access, execute the following statement:

```
PERMIT,TESTDATA,USER=776655,ACCESS=NONE.
```

To list the results of the three PERMIT statements, execute a LISTAC statement to list all access permission sets that apply to file TESTDATA:

```
LISTAC,TESTDATA,USER=*.
```

The * requests all access permission sets, including the access permission set for the file owner. The LISTAC output appears as follows:

```
NAME       OWNER    USER      ACCESS

TESTDATA   012306   012306    XMARW
                    987654    R
                    776655    NONE
                    GENERAL   R
```

Giving Your Private Files to Another User

The owner of a private file can change file ownership by giving the file to another user or to a pool. The other user or the pool becomes the new owner, and the former owner cannot access the file unless granted access by the new owner.

The GIVE control statement changes the file ownership. The following statement, for example, gives file FILE1 to user 987654.

```
GIVE,FILE1,U=987654.
```

Pool Files

A pool is a group of files that can be accessed through a single command. Once a pool has been created, you can attach all files in the pool through a single control statement.

If your user name has access to a pool named DATAPOOL, for example, your job can attach all files in DATAPOOL by executing the following control statement:

```
PATTACH,DATAPOOL.
```

To list the files in the attached pool, execute the following statement:

```
FILES,POOL=DATAPOOL.
```

These pool files remain attached to your job until your job terminates unless your job executes the following statement:

```
PDETACH,DATAPOOL.
```

For information on creating pools and granting pool access to other users, refer to the VSOS Reference Manual, Volume 1.

TRANSFERRING FILES TO AND FROM A FRONT-END SYSTEM

Your CYBER 200 job can transfer data to and from a front-end system. Your job can request that data is copied from a front-end file to a temporary or permanent CYBER 200 file. Similarly, your job can request that data is copied from a temporary or permanent CYBER 200 file to a front-end file.

Data is transferred between the CYBER 200 system and its front-end systems over hardware called the loosely coupled network (LCN) using the software called the remote host facility (RHF). The RHF applications that process data file transfers are called the permanent file transfer facility (PTF) and the permanent file transfer facility servicer (PTFS). For a more complete description of RHF transfers, refer to the RHF handbook for your system.

The control statement that requests a data file transfer is the MFLINK statement. An MFLINK can be initiated on either a front-end or back-end system. This statement specifies the following information:

- The name of the CYBER 200 data file
- The identifier of the front-end system
- A data declaration indicating the format of the data to be transferred
- Directives to be passed to the front-end system

For example, the following MFLINK statement copies a NOS file to a CYBER 200 file:

```
MFLINK,INFILE1,ST=ADB,DD=UU,
JCS="USER,12345,XPWXPWX.","CHARGE,PROJ,ACCOUNT.",
"GET,OTFILE1.".
```

The CYBER 200 file name is INFILE1. If INFILE1 is already an attached permanent file, MFLINK uses the existing file. If INFILE1 does not exist, however, MFLINK creates a local file named INFILE1. If INFILE1 exists as a local file, MFLINK discards the old local file and creates a new local file named INFILE1.

The ST parameter specifies the front-end system identifier ADB.

The DD parameter specifies the data declaration keyword UU. RHF recognizes four data declaration keywords. There are two keywords for character data (C6 and C8) and two keywords for binary data (US and UU). Use of the C6 and C8 keywords is described under the Character Data Transfers heading. Use of the US and UU keywords is described under the Binary Data Transfers heading.

The JCS parameter specifies three directives to be passed to the front-end system (in this example, NOS). The USER and the CHARGE directives specify the required NOS validation information. The GET directive specifies an indirect file to be copied to the CYBER 205.

CHARACTER DATA TRANSFERS

For character data transfers, the data declaration keyword, C6 or C8, is required only if the front-end system uses more than one character set.

CYBER 170 systems use both 6-bit display code (the default display code) and 8/12 ASCII code. You may specify C6 if the file uses or will use display code. You must specify C8 if the file uses or will use 8/12 ASCII code. In either case, RHF performs all required character code conversion between the character set of the front-end system and the ASCII character set used on the CYBER 200 system.

Character Data Transfer Example

Figure 3-4 shows a job that performs a character data transfer. The figure lists the CYBER 200 job file with its NOS input and output files. The job performs the following steps:

1. Copies the data in the NOS file INFILE1 to the CYBER 200 file INFILE1. The NOS file contains four records of 6-bit display code characters that MFLINK converts to four records of 8-bit ASCII characters.

2. Compiles and loads the program TRIANG.

3. Executes the TRIANG program, which iterates a loop that performs the following steps:

   a. Reads a line using formatted I/O. The READ statement reads three fields of seven characters each and converts each field to a CYBER 200 floating-point number.

   b. Determines if the values are valid lengths for the three sides of a triangle. If so, the program computes the area of the triangle.

   c. Writes one or two lines of character data to file OTFILE2. The first line contains the three input values reconverted to ASCII character data. The second line, if written, contains the computed area value that the CYBER 200 file OTFILE2 transferred to the NOS file OTFILE2. MFLINK converts five records of 8-bit ASCII character code to five records of 6-bit display code.

4. Copies tha data in the CYBER 200 file OTFILE to the NOS file OTFILE2. MFLINK converts five records of 8-bit ASCII character code to five records of 6-bit display code.

BINARY DATA TRANSFERS

The binary data keywords, UU and US, indicate that no character code conversion is performed. Data is transferred as a bit string. Use of the UU or the US keyword depends on whether the data transferred requires the use of the logical structure indicators (EOR, EOF, and EOI).

For binary data transfers to the CYBER 200 system, specifying UU discards the EOR and EOF indicators and converts the EOI to the end-of-file indicator. In this case, the CYBER 200 file must use the system interface language (SIL) record type U. When keyword US is specified, the CYBER 200 file must use the SIL record type W. Use of the US keyword keeps the logical structure indicators, however, these indicators are converted to their record type W equivalents.

Job File:

```
ADEY,ST=ABC.
USER,USER=123456,ACCOUNT=ACCT933,PASSWORD=XYZ.
RESOURCE,TL=20.
MFLINK,INFILE1,ST=ADB,DD=C6,JCS="USER,12345,XPWXPWX.",
"CHARGE,PROJ,ACCOUNT.","GET,INFILE1.".
FTN200.
LOAD.
GO.
MFLINK,OTFILE1,ST=ADB,DD=C6,JCS="USER,12345,XPWXPWX.",
"CHARGE,PROJ,ACCOUNT.","SAVE,OTFILE1.".
(EOR)
      PROGRAM TRIANG(INFILE1,OTFILE1,UNIT1=INFILE1,UNIT2=OTFILE1)
C
   10 READ(1,100) A, B, C
  100 FORMAT(3F7.0)
      IF (A .EQ. 0.0) STOP
      S = (A + B + C) / 2.0
      RDCL = S * (S-A) * (S-B) * (S-C)
      IF (RDCL .LT. 0.0) GO TO 20
      AREA = SQRT(RDCL)
      WRITE(2,200) A, B, C, AREA
  200 FORMAT('SIDES ARE ', 3F9.4, /,' AREA IS ', F11.4/)
      GO TO 10
C
   20 CONTINUE
      WRITE(2,300) A, B, C
  300 FORMAT('SIDES OF ', 3F9.4, ' FORM AN INVALID TRIANGLE',/)
      GO TO 10
      END
(EOR)
```

Input File INFILE1:

```
3.0000 4.0000 5.0000
2.0000 1.0000 5.0000
1.5197 1.5197 1.5197
0 0 0
```

Output File OTFILE1:

```
SIDES ARE    3.0000   4.0000   5.0000
AREA IS      6.0000
SIDES OF    2.0000   1.0000   5.0000 FORM AN INVALID TRIANGLE
SIDES ARE    1.5197   1.5197   1.5197
AREA IS      1.0000
```

Figure 3-4. Character Data Transfer Example

For binary data transfers from the CYBER 200 system, the keyword used depends on the SIL record type of the CYBER 200 file. The US keyword is valid only if the SIL record type is W. The transfer converts the logical structure indicated by the W control words to the logical structure indicators (EOR, EOF, and EOI) of the front-end system. The UU keyword causes MFLINK to transfer the file as a bit string with no logical structure indicators. If the file uses the SIL record type R or W, the record marks or control words are not converted. They are transferred as data.

In summary, use of the UU or US keyword depends on whether the data requires a logical structure.

- The US keyword maintains the logical structure. The CYBER 200 file must use the W record type.
- The UU keyword does not maintain the logical structure. When transferring data to the CYBER 200, VSOS discards the logical structure indicators. The CYBER 200 file must use the U record type.

NOTE

If you intend to transfer an output file from the FORTRAN 200 program to the front-end system using the UU keyword, explicitly request the output file to be record type U. Otherwise, the FORTRAN 200 program creates an output file having record type W.

For example, in figure 3-6 all operands are read into an array and converted by a single call. In this case, no logical structure is needed, and the UU keyboard can transfer the file. If the program contains more than one READ statement so that more than one sequence of operands is read and converted, however, a logical structure is required to delimit the sequences of operands.

CRM Data Delimiters

A binary data transfer on a CYBER 170 front-end system transfers all the CYBER record manager (CRM) delimiters embedded in the data. For example, if the data being transferred is written using the CRM record type W (the FORTRAN 5 default), the control words that delimit records and groups are transferred embedded in the data. It is recommended, therefore, that you write all binary data to be transferred using the CRM record type S which does not use embedded delimiters.

To specify the CRM record type S for a file, execute a NOS FILE control statement before executing the program that writes the file. For example, before executing the program that writes file INFILE1, execute this statement:

```
FILE,INFILE1,RT=S.
```

On NOS you can check that a file contains no CRM delimiters. Use a TDUMP statement to look at the file contents. For example, assume that you specify CRM record type S for file INFILE1 and then execute a FORTRAN 5 program in which an unformatted WRITE statement writes an array of integers on file INFILE1. Figure 3-5 shows an octal dump of the file performed by a NOS TDUMP statement.

```
/tdump,i=infile1,o
F  1 R   1 W    0- 0000 0000 0000 0000 0001   0000 0000 0000 0000 0002
F  1 R   1 W    2- 0000 0000 0000 0000 0003   0000 0000 0000 0000 0004
F  1 R   1 W    4- 0000 0000 0000 0000 0005   0000 0000 0000 0000 0006
F  1 R   1 W    6- 0000 0000 0000 0000 0007   0000 0000 0000 0000 0010
F  1 R   1 W    8- 0000 0000 0000 0000 0011   0000 0000 0000 0000 0012
F  1 R   1 W   12- 0000 0000 0000 0000 0013   0000 0000 0000 0000 0014
F  1 R   1 W   14- 0000 0000 0000 0000 0015   0000 0000 0000 0000 0016
F  1 R   1 W   16- 0000 0000 0000 0000 0017   0000 0000 0000 0000 0020
```

Figure 3-5. TDUMP Octal Dump Example

Data Conversion

The TDUMP output in figure 3-5 shows a sequence of CYBER 170 integers in octal notation. To interpret the contents of a dump, you need to know the data representations used. Section 4 describes the data representations used for FORTRAN 5 and for FORTRAN 200.

Because data representations differ between the CYBER 200 and its front-end systems, the FORTRAN 200 program must convert all transferred integer or floating-point numbers to the corresponding CYBER 200 formats before using the numbers as operands. Similarly, before transferring binary results to the front-end system, the FORTRAN 200 program must convert the data to the appropriate front-end format.

VSOS provides FORTRAN data conversion routines to perform integer and floating-point conversion. These data conversion routines have names that begin with Q9 and are described in an appendix of the VSOS Reference Manual, Volume 1. The job file example in figure 3-6 uses the Q9LCI and Q9CLI subroutines.

NOTE

To be converted by a data conversion subroutine, operands must be read into an array as a packed string of operands. The entire array must then be read by a single READ statement. This READ statement reads one record containing the entire string of operands.

```
ADEY,ST=ABC.
USER,USER=123456,ACCOUNT=ACCT933,PASSWORD=XYZ.
RESOURCE,TL=20.
DEFINE,INFILE1,RT=W.
MFLINK,INFILE1,ST=ADB,DD=US,JCS="USER,12306,XPWXPWX.",
"CHARGE,PROJ,ACCOUNT.","GET,INFILE1.".
LOOK,INFILE1.
FTN200.
LOAD
GO.
LOOK,OTFILE2.
MFLINK,OTFILE2,ST=ADB,DD=US,JCS="USER,12306,XPWXPWX.",
"CHARGE,PROJ,ACCOUNT.","SAVE,OTFILE2.".
PURGE,INFILE1.
(EOR)
HEX,0,10
END
(EOR)
      PROGRAM LOOP(INFILE1,OTFILE2,UNIT1=INFILE1,UNIT2=OTFILE2)
      INTEGER CMPARAY, OUTARAY
      DIMENSION INARAY(15), CMPARAY(16), OUTARAY(17)
C
      READ(UNIT=1,END=10) INARAY
   10 CALL MDUMP(INARAY,15,'Z',6HOUTPUT)
      CALL Q9LCI(INARAY,CMPARAY,16,ISTAT)
      IF (ISTAT .NE. 0) THEN
      PRINT 100, ISTAT
  100 FORMAT(1X,I8)
      END IF
      CALL MDUMP(CMPARAY,16,'Z',6HOUTPUT)
C
      CMPARAY(16) = 0
      DO 20 I=1,15
   20 CMPARAY(16) = CMPARAY(16) + CMPARAY(I)
C
      CALL Q9CLI(CMPARAY,OUTARAY,17,ISTAT)
      IF (ISTAT .NE. 0) THEN
      PRINT 100, ISTAT
      END IF
      CALL MDUMP(OUTARAY,16,'Z',6HOUTPUT)
      WRITE(UNIT=2) OUTARAY
      STOP
      END
(EOR)
HEX,0,11
END
```

Figure 3-6. Binary Data Transfer Example

Binary Data Transfer Example

When the CYBER 200 job shown in figure 3-6 is executed, it performs the following steps:

1. Defines a file named INFILE1 having the CYBER 200 System Interface Language (SIL) record type W.

2. Copies the data from NOS file INFILE1 to CYBER 200 file INFILE1. MFLINK requests the temporary file INFILE1 with SIL record type W.

3. Compiles and loads the program LOOP.

4. Executes the LOOP program that performs the following steps:

   a. Reads a record of data from file INFILE1 into an array.

   b. Calls the Q9LCI subroutine to convert the data from CYBER 170 integer format to CYBER 200 integer format.

   c. Adds the integers in the array and stores the sum as the last element in the array.

   d. Calls the Q9CLI subroutine to convert the data from CYBER 200 integer format to CYBER 170 integer format.

   e. Writes the converted data as a record on file OTFILE2. By default, it has requested the temporary file OTFILE2 with SIL record type W.

5. Copies the data from the CYBER 200 file OTFILE2 to the NOS indirect access file OTFILE2.

6. Purges the INFILE1 file defined by the job.

The LOOK statements and MDUMP subroutine calls are not required for a job. They are inserted so you can trace changes in the data.

The LOOK control statements dump the data in hexadecimal notation before and after the program uses the data. Each LOOK statement requires an input group that contains LOOK directives. The VSOS Reference Manual, Volume 1, fully describes the LOOK utility.

The MDUMP calls within program LOOP dump the data in hexadecimal notation before and after the dump is converted to CYBER 200 representation. MDUMP also provides a third dump of data after the data is reconverted to CYBER 170 representation. The FORTRAN 200 Reference Manual fully describes the MDUMP subroutine.

Using the data in NOS file INFILE1 in figure 3-6, the output from the LOOK statements and MDUMP calls is shown in figure 3-7.

- The first LOOK dump in figure 3-7 indicates that the data shown in figure 3-6 has been transferred as a single record type W on the CYBER 200 file.

- The first MDUMP dump shows that the FORTRAN READ statement read the data, but discarded the SIL control words.

- The second MDUMP dump shows the data converted to CYBER 200 integer format.
- The third MDUMP dump shows the data and its sum converted back to CYBER 170 integer format.
- The second LOOK dump shows that the FORTRAN WRITE statement wrote the data as a single record type W.

The job saves the output file on the front-end system. If you execute a TDUMP statement on the front-end system, the dump shows that the contents of the OTFILE2 file are the same as the contents of the input file data shown in figure 3-5 plus the following three words:

```
0000 0000 0000 0000 0210  0000 0000 0000 0000 0000
0000 0000 0000 0000 0000
```

The first word is the sum of the operands as computed by the FORTRAN 200 program.

USING A CYBER 200 FILE INTERACTIVELY

So far, this section has described program input that is read from existing files. There are times, however, when you do not want your program to read data that has been previously stored in a file. Instead, you may wish to enter data directly from an interactive terminal to your executing program and have the program send a response.

To perform interactive I/O, a program uses formatted I/O statements or SIL calls to read from and write to an interactive terminal file (device type TE). Interactive terminal files are fully described in the VSOS Reference Manual, Volume 1. The example in figure 3-8 shows the use of interactive terminal files.

You can change the triangle area calculation program shown in figure 2-4 to enable you to enter input and to receive output interactively. Modifying the program for interactive I/O results in the following program changes:

- Because the interactive terminal file is used as the input file and the output file, the READ and WRITE statements must reference the same file.
- Because the program is interactive, it sends messages to the user describing the input the program expects. The program also sends a final termination message to the user. Refer to figure 3-9 for examples of messages sent from the system to the user.

Figure 3-8 shows modifications made to the triangle calculation program to allow for interactive use.

```
First LOOK:

  BIT ADDR      WORD ADDR                                   CONTENTS                                                       ASCII
000000000000  00000000000  00110000 08000078  00000000 00000010  00000000 00000200  00000000 00003000              X               0
000000000100  00000000004  00000000 00040000  00000000 00500000  00000000 06000000  00000000 70000000                     P       P
000000000200  00000000008  00000008 00000000  00000090 00000000  00000A00 00000000  0000B000 00000000
000000000300  0000000000C  000C0000 00000000  00D00000 00000000  0E000000 00000000  F0000000 00000010


First MDUMP:

  BIT ADDRESS                                         C-O-N-T-E-N-T-S                  WORD ADDRESS                 ASCII

000000083A00  00000000 00000010  00000000 00000200  00000000 00003000  00000000 00040000  000000020E8                       0
000000083B00  00000000 00500000  00000000 06000000  00000000 70000000  00000008 00000000  000000020EC        P              P
000000083C00  00000090 00000000  00000A00 00000000  0000B000 00000000  000C0000 00000000  000000020F0
000000083D00  00D00000 00000000  0E000000 00000000  F0000000 00000010                     000000020F4


Second MDUMP:

  BIT ADDRESS                                         C-O-N-T-E-N-T-S                  WORD ADDRESS                 ASCII

000000082E80  00000000 00000001  00000000 00000002  00000000 00000003  00000000 00000004  000000020BA
000000082F80  00000000 00000005  00000000 00000006  00000000 00000007  00000000 00000008  000000020BE
000000083080  00000000 00000009  00000000 0000000A  00000000 0000000B  00000000 0000000C  000000020C2
000000083180  00D00000 0000000D  00000000 0000000E  00000000 0000000F  00000000 00000010  000000020C6


Third MDUMP:

  BIT ADDRESS                                         C-O-N-T-E-N-T-S                  WORD ADDRESS                 ASCII

0000000833C0  00000000 00000010  00000000 00000200  00000000 00003000  00000000 00040000  000000020CF                       0
0000000834C0  00000000 00500000  00000000 06000000  00000000 70000000  00000008 00000000  000000020D3        P              P
0000000835C0  00000090 00000000  00000A00 00000000  0000B000 00000000  000C0000 00000000  000000020D7
0000000836C0  00D00000 00000000  0E000000 00000000  F0000000 00000010  00000000 00000880  000000020DB


Second LOOK:

  BIT ADDR      WORD ADDR                                   CONTENTS                                                       ASCII
000000000000  00000000000  00110000 08000088  00000000 00000010  00000000 00000200  00000000 00003000                              0
000000000100  00000000004  00000000 00040000  00000000 00500000  00000000 06000000  00000000 70000000        P                     P
000000000200  00000000008  00000008 00000000  00000090 00000000  00000A00 00000000  0000B000 00000000
000000000300  0000000000C  000C0000 00000000  00D00000 00000000  0E000000 00000010  F0000000 00000010
000000000400  00000000010  00000000 00000880  00000000 00000088  00130000 90000000  000C0000 08000000
```

Figure 3-7. Data Dumps From MFLINK Job Example

```
ADEY,ST=ABC.
USER,USER=123456,ACCOUNT=ACCT933,PASSWORD=XYZ.
RESOURCE,TL=20.
FTN200.
LOAD.
DEFINE,GO.
(EOR)
      PROGRAM TRIANG(INFILE1,UNIT1=INFILE1)
      WRITE (1,100)
  100 FORMAT('TRIANGLE AREA CALCULATION')
C
   10 WRITE (1,200)
  200 FORMAT('WAIT FOR THE .. PROMPT AND THEN ENTER ')
      WRITE (1,250)
  250 FORMAT('THE TRIANGLE SIDE LENGTHS AS THREE 7-CHAR FIELDS')
      WRITE (1,300)
  300 FORMAT('OR ENTER 0 (ZERO) TO STOP.')
C
      READ(1,400) A, B, C
  400 FORMAT(3F7.0)
      IF (A .EQ. 0.0) GO TO 30
      S = (A + B + C) / 2.0
      RDCL = S * (S-A) * (S-B) * (S-C)
      IF (RDCL .LT. 0.0) GO TO 20
      AREA = SQRT(RDCL)
      WRITE(1,500) A, B, C, AREA
  500 FORMAT('SIDES ARE ', 3F9.4, /,' AREA IS ', F11.4/)
      GO TO 10
C
   20 CONTINUE
      WRITE(2,600) A, B, C
  600 FORMAT('SIDES OF ', 3F9.4, ' FORM AN INVALID TRIANGLE',/)
      GO TO 10
C
   30 WRITE(1,700)
  700 FORMAT('ZERO ENTERED.  TRIANGLE AREA CALCULATION ENDS.')
      STOP
      END
(EOR)
```

Figure 3-8. Job File Modified for Interactive I/O

After the modified program is compiled and loaded, and the executable file is defined as the permanent CYBER 200 file named GO, the interactive session can begin. Figure 3-9 shows a CYBER 200 interactive session that executes file GO.

```
 PLEASE ENTER CY200 LOGIN
login,123456,acct933,xyz                          ◄— CYBER 200 login line
VSOS 2.2 RSYS22E3 VSYS22E3     G  09980.6596  ACTIVE  NONE

attach,go                                         ◄— Attaches the executable file.
 ALL DONE
request,infile1,device=te                         ◄— Requests the terminal file.
 ALL DONE
go                                                ◄— Executes the interactive task.
TRIANGLE AREA CALCULATION
WAIT FOR THE .. PROMPT AND THEN ENTER
THE TRIANGLE SIDE LENGTHS AS THREE 7-CHAR FIELDS
OR ENTER 0 (ZERO) TO STOP.
..
    3.0    4.0    5.0                             ◄— First set of input values.
SIDES ARE    3.0000   4.0000   5.0000
$EOR

AREA IS      6.0000                               ◄— First result value.
$EOR

WAIT FOR THE .. PROMPT AND THEN ENTER
THE TRIANGLE SIDE LENGTHS AS THREE 7-CHAR FIELDS
OR ENTER 0 (ZERO) TO STOP.
..
      2      1      5                             ◄— Second set of input values.
SIDES OF    2.0000   1.0000   5.0000 FORM AN INVALID TRIANGLE
$EOR

WAIT FOR THE .. PROMPT AND THEN ENTER
THE TRIANGLE SIDE LENGTHS AS THREE 7-CHAR FIELDS
OR ENTER 0 (ZERO) TO STOP.
..
0                                                 ◄— Stops program execution.
ZERO ENTERED.  TRIANGLE AREA CALCULATION ENDS.
$EOR

STOP
 ALL DONE
$bye                                              ◄— Ends CYBER 200 interactive
                                                     session.
BYE
ITF      CONNECT TIME 00.03.20.
```

Figure 3-9. Interactive I/O Example

USING MAGNETIC TAPE FILES

If the input data for your CYBER 200 program is stored on magnetic tape, there are two possible ways to read the data.

The first method uses the tape drives in the front-end system configuration. The front-end job reads the data and copies it to a front-end mass storage file. The subsequent CYBER 200 job then transfers the data from the front-end file to a CYBER 200 file.

If your site is configured with a CYBER 200 tape system, there is a second method to read data stored on tape. This second method uses the CYBER 200 online tape drives. If your site has a CYBER 200 tape system, your CYBER 200 job can read and write data directly to these tape files.

RESERVING TAPE DRIVES

When a CYBER 200 job reads or writes tape files, the job reserves the tape drives for use on the RESOURCE statement NT parameter. The following statement, for example, reserves one tape drive:

```
RESOURCE,NT=1.
```

REQUESTING A TAPE FILE

A CYBER 200 job requests each tape file with a REQUEST statement. The REQUEST statement has numerous parameters. Except for the file name and the device assignment, the following parameters are optional:

1. Specify file name and device assignment. These are required. Default device assignment is to disk, DEVICE=MS.

2. Specify if you intend to write to the file. If you intend to write to the file, you must specify write, ACCESS=W. Default for tape files is read only, ACCESS=R.

3. Specify the volume serial number of the tape volume on the VSN parameter (VSN=xxxxxx). If you want the operator to mount a scratch tape, omit the VSN parameter and specify an operator message on the MESSAGE parameter (such as MESSAGE="MOUNT SCRATCH TAPE").

4. Specify a recording density with the DENSITY parameter. VSOS supports two recording densities; phase encoded (PE), 1600 cpi, and group encoded (GE), 6250 cpi. If density is omitted, the installation-defined density is used.

5. Specify tape labeling. If a tape is unlabeled, specify LABEL=UL. If the tape has nonstandard labels, specify LABEL=NS. Your program can process nonstandard labels using SIL calls as described in the VSOS Reference Manual, Volume 1. Default is ANSI standard labels, LABEL=AN.

   If you specify ACCESS=RW, the labels are not rewritten unless you specify LPROC=W. If you specify ACCESS=W, the labels are rewritten unless you specify LPROC=R.

6. Specify the volume accessibility character in the VOL1 label. The character is specified on VA parameter, VA=xx.

7. Specify whether file data is recorded as character codes or as binary data. For character data, use the CONVERT parameter. Specify the character code set (AS for ASCII or EB for EBCDIC) on the CM parameter.

8. Specify the tape format and blocking type with the TF and BT parameters (refer to the VSOS Reference Manual, Volume 1). If the data format is V, use the MPRU parameter to specify the maximum PRU size. If the blocking type is K, use the RPB parameter to specify the number of records per block.

9. Specify the SIL record type with the RT parameter. The following additional parameters provide record specifications:

   RMD Nondefault record delimiter character for record type R

   PC Nonblank padding character for record type F

   RLMAX Maximum record length or the fixed record length for record type F

   RLMIN Minimum record length

10. Specify any special tape processing options as follows:

   HEC=N Use standard error recovery instead of on-the-fly correction

   IU=Y Inhibit unload of tape volumes

   RETRY=N Omit standard error recovery

   RU=Y Read past end-of-tape marker

FORTRAN PROGRAM CONVERSION 4

This guide assumes that your primary reason for learning about the CYBER 200 system is to enable you to compile and to execute FORTRAN programs on the system. The guide also assumes that you intend to use the CDC FORTRAN 200 Version 1 compiler, which is referred to hereafter as FORTRAN 200.

FORTRAN 200 is a superset of the American National Standards Institute FORTRAN language, which is described in ANSI document X3.9-1978. FORTRAN 200 is fully described in the FORTRAN 200 Version 1 Reference Manual (publication 60480200). Access to the FORTRAN 200 Reference Manual is required to use the full capabilities of the compiler.

In many respects, FORTRAN 200 resembles CDC FORTRAN Extended Version 5, which is referred to hereafter as FORTRAN 5. Both compilers are supersets of ANSI FORTRAN X3.9-1978. Both support many standard CDC FORTRAN extensions.

This section highlights the areas in which the FORTRAN 200 and FORTRAN 5 compilers differ. These differences can be categorized as follows:

- Syntax differences
- FORTRAN-provided function differences
- FORTRAN-provided subroutine differences
- Product interface differences
- Machine-dependent differences

FORTRAN 5/FORTRAN 200 SYNTAX DIFFERENCES

If you have a FORTRAN program that compiles with the FORTRAN 5 compiler and you want to convert the program to compile with the FORTRAN 200 compiler, you must begin by searching the source code for syntax that is not supported by FORTRAN 200. Table 4-1 lists syntax differences between the two compilers.

Table 4-1. FORTRAN 5/FORTRAN 200 Syntax Differences

Item	Difference
Character set	Unlike the FORTRAN 5 character set, the FORTRAN 200 character set does not include the $ and " characters. This results in the following changes: • No C$ compiler directives • No $ in namelist input; replace each $ character with a & character • No Boolean constants enclosed in quotes (") • No " edit specifier
Boolean type	FORTRAN 200 does not support the Boolean data type. Each Boolean constant, variable, and array must be converted to another data type depending on its content. Boolean type can be changed to character, Hollerith, hexadecimal, or bit data type.
Octal constants	FORTRAN 200 does not support octal constants. Convert each octal constant to the appropriate decimal or hexadecimal constant. Hexadecimal constants can only be used in DATA statements; a hexadecimal constant cannot be a symbolic constant.
Complex array subscripts	FORTRAN 200 does not support complex array subscripts. Convert the subscripts to integer values.
PARAMETER statement	FORTRAN 200 does not support extended constant expressions in a PARAMETER statement. Convert each extended constant expression to a simple constant expression.
OVERLAY statement	FORTRAN 200 does not support the OVERLAY statement because it is not needed on a virtual storage machine. For the same reason, FORTRAN 200 does not provide STATIC capsule loading routines.
LEVEL statement	FORTRAN 200 does not support the LEVEL statement because a CYBER 200 system does not have extended memory.
DATA statement	FORTRAN 200 does not support the form r(c,c,...) on DATA statements. Convert each r(c,c,...) specification to individual r*c specifications.
Computed GO TO statements	FORTRAN 200 does not support arithmetic or Boolean expressions on computed GO TO statements. Convert each arithmetic or Boolean expression to an integer expression.
FORMAT statement	FORTRAN 200 does not support the O or " edit descriptors.
I/O statements	The unit specification cannot be a display code name in L format. FORTRAN 200 does not return CRM error codes in the IOSTAT variable; it returns its own execution-time error codes instead.
RETURN statement	A RETURN statement can only specify an integer or a simple integer variable.

FORTRAN-SUPPLIED FUNCTIONS

FORTRAN 200 provides the same ANSI standard intrinsic functions that FORTRAN 5 provides and many of the FORTRAN 5 nonstandard functions. FORTRAN 200, however, does not provide all FORTRAN 5 intrinsic functions. Table 4-2 lists the missing intrinsic functions.

BOOLEAN FUNCTIONS

Because FORTRAN 200 does not support the Boolean data type, it does not provide the Boolean functions BOOL, EQV, or NEQV. Other Boolean functions are provided for compatibility although the result is typeless, rather than Boolean. (A typeless result is not converted when used as an argument or when assigned to a variable of another type.)

The argument type for the following functions can be logical, integer, or real. The result is typeless.

AND	Bit-by-bit logical product
COMPL	Bit-by-bit Boolean complement
MASK	Bit mask generator (0 to 64 bits set to 1, starting at the left of the word)
OR	Bit-by-bit logical sum
SHIFT	Bit shift (Positive integer argument, left circular shift. Negative integer argument, right end-off shift with sign extension)
XOR	Bit-by-bit exclusive OR

Appendix G of the FORTRAN 200 Reference Manual describes these functions.

I/O FUNCTIONS

FORTRAN 200 provides the UNIT and LENGTH functions for buffer I/O compatibility. These functions are described in appendix G of the FORTRAN 200 Reference Manual.

FORTRAN-SUPPLIED SUBROUTINES

FORTRAN 5 and FORTRAN 200 each provide a set of subroutines that extend the capabilities of the program. However, not all subroutines provided by FORTRAN 5 are provided by FORTRAN 200. The following paragraphs list the FORTRAN 5 subroutines that FORTRAN 200 does not have.

Table 4-2. FORTRAN 5 Intrinsic Functions Not Provided by FORTRAN 200

Name	Function
	FORTRAN 5 Boolean Functions
BOOL	Type conversion to Boolean.
EQV	Boolean equivalence.
NEQV	Boolean nonequivalence.
	Trigonometic Functions
ATANH	Hyperbolic arctangent (replace with computation using hyperbolic tangent function, TANH).
COSD	Cosine from argument expressed in degrees (replace with computation using radians and COS function).
SIND	Sine from argument expressed in degrees (replace with computation using radians and SIN function).
TAND	Tangent from argument expressed in degrees (replace with computation using radians and TAN function).
	I/O Functions
EOF	Use the END specifier on the I/O statement.
IOCHEC	No specific parity error function is provided; however, the IOSTAT specifier on an I/O statement can return an error code.
	Other Functions
JDATE	Current Julian date (use DATE or call Q5TIME for the binary Julian date).
CLOCK	Current time (use TIME).
ERF	Error function; computes the following: $erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$
ERFC	Complement of ERF function.
LOCF	Address of argument (FORTRAN 5 does not recommend use of LOCF).

SUBROUTINES WITH FUNCTIONAL EQUIVALENTS

The following FORTRAN 5 subroutines are not available to the FORTRAN 200 programmer, but other methods are available to perform these same functions. The VSOS system interface language (SIL) subroutines usually provide a functional equivalent. Each SIL subroutine name begins with the characters Q5. Sections 8 and 9 of the VSOS Reference Manual, Volume 1, describe SIL.

Subroutine	Equivalent Processing
DISPLAY or REMARK	Call Q5SNDMDF to send a message to the job dayfile. Call Q5SNDMOP to send a message to the system console.
LENGTHX	VSOS SIL calls return the number of bytes read or written.
EXIT	Use a STOP statement.
CHEKPTX	Use the CHKPNT subroutine to checkpoint a task as described in section 7 of the VSOS Reference Manual, Volume 1.
RECOVR	Use abnormal termination control (ATC) calls for condition interrupts or the data flag branch manager (DFBM) for computation errors. The VSOS Reference Manual, Volume 1, describes ATC. DFBM is explained in the FORTRAN 200 Reference Manual.
CONNEC and DISCON	For interactive files, call Q5RQUEST to request a local file. Call Q5RETURN to return an interactive file.
LABEL	Use VSOS SIL calls to read and write tape labels.
OPENMS WRITMS READMS CLOSMS STINDX	Use VSOS SIL direct access file organization to read and write records randomly.
DUMP PDUMP STRACE LEGVAR SYSTEM SYSTEMC LIMERR	For debugging, use the MDUMP subroutine and the DEBUG utility.

SUBROUTINES WITHOUT FUNCTIONAL EQUIVALENTS

The following FORTRAN 5 subroutines are not available to the CYBER 200 programmer. No functional equivalent is provided.

Subroutine	Equivalent Processing
GETPARM	VSOS does not provide a subroutine to get the parameters from the execution statement.
SSWTCH	VSOS does not maintain sense switches for a task.
MOVLEV or MOVLCH	The CYBER 200 does not use extended memory.
COLSEQ WTSET CSOWN	FORTRAN 200 does not provide a collating sequence control capability.

PRODUCT INTERFACES

A CYBER 200 program cannot interface with CYBER 170 products. Functional equivalents, however, are available for the following CYBER 170 products:

Product	Functional Equivalent
NOS permanent file commands (PF subroutine)	SIL calls can perform all VSOS permanent file functions.
CYBER Record Manager (CRM)	SIL calls can manipulate the file information table (FIT) and perform record and block I/O.
CYBER Memory Manager (CMM)	SIL calls can advise the system of the program's memory requirements (Q5ADVISE and Q5MEMORY).
COMPASS	A program can call subprograms written in CYBER 200 assembly language (META) or its implementation language (IMPL).
8-Bit Subroutines	VSOS provides arithmetic conversion routines for IBM and CYBER 170 data.

Functional equivalents are not available for the following CYBER 170 product interfaces: Sort/Merge, CYBER Database Control System (CDCS), Information Management Facility (IMF), Queued Terminal Record Manager (QTRM), and Transaction Facility (TAF).

MACHINE-DEPENDENT DIFFERENCES

Machine-dependent code is code that executes correctly only on a particular computer. This code often assumes that certain data representation is used. When converting a FORTRAN program for use on the CYBER 200, look closely for machine dependencies in the following areas:

- Masking expressions
- Shift operations
- Logical operators used for purposes other than creating logical variables
- Equivalenced real and integer variables
- Initialization of real and integer variables when the difference in data types is ignored
- Plus and minus zero
- Hollerith variables
- Character variables/data (especially if equivalenced)

The machine-dependent differences between a CYBER 200 and a CYBER 170 stem from the following two differences:

- Different word size (64 bits on the CYBER 200 instead of 60 bits on the CYBER 170)
- Different number representation (two's complement on the CYBER 200 instead of one's complement on the CYBER 170).

The different word size affects the character codes and the base in which numbers are displayed.

- The 60-bit CYBER 170 word is split into ten 6-bit bytes. Each byte represents one display code. The 64-bit CYBER 200 word is split into eight 8-bit bytes. Each byte representing one ASCII character.
- CYBER 170 numbers are shown using octal notation with 3 bits per digit. CYBER 200 numbers are shown using hexadecimal notation with 4 bits per digit.

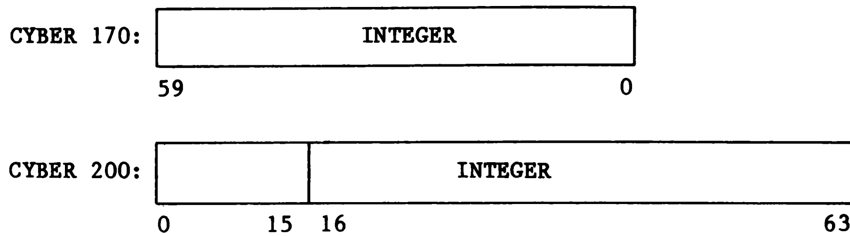
Two's complement number representation results in the same representation for positive and negative zero. Integer zero and floating-point zero, however, do not have the same representation.

Note also that the bits within a CYBER 170 word are numbered from right to left (bits 59 through 0). The bits within a CYBER 200 word are numbered from left to right (bits 0 through 63). The CYBER 200 is a bit-addressable machine, so each successive bit has the next bit address in sequence.

The following paragraphs focus on the differences in data representation for each data type. For more information on number systems and CYBER 200 arithmetic operations, refer to appendixes A and B in the CYBER 200 Model 205 Hardware Reference Manual.

INTEGER REPRESENTATION

The following are graphic representations of the CYBER 170 integer format and the CYBER 200 integer format:



Bit 59 is the sign bit for a CYBER 170 integer (0 for a positive number, 1 for a negative number). Bit 16 is the sign bit for a CYBER 200 integer (0 for a positive number, 1 for a negative number).

In most cases for a CYBER 170 integer, only the lower 48 bits are used for multiplication and division or for conversion from an integer to a real number. The full 60 bits are used for addition and subtraction.

The following examples show the same integer values as represented in the CYBER 170 and in the CYBER 200. CYBER 170 numbers are shown using octal notation. CYBER 200 numbers are shown using hexadecimal notation:

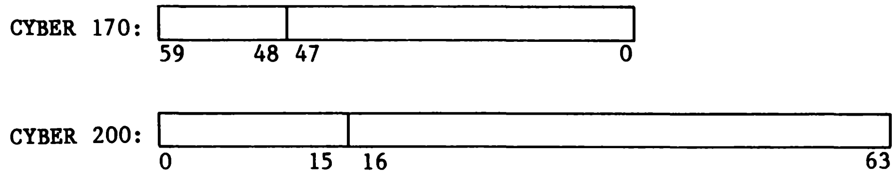
Integer	CYBER 170 Representation	CYBER 200 Representation
$+2^{47}-1$	00003777777777777777	00007FFFFFFFFFFF
+1	00000000000000000001	0000000000000001
+0	00000000000000000000	0000000000000000
-0	77777777777777777777	0000000000000000
-1	77777777777777777776	0000FFFFFFFFFFFF
$-2^{47}-1$	77774000000000000000	0000800000000001

Consider the following example that determines the value of a negative two's complement integer:

```
0000FFFFFFFFFFC2     Bit 16 is 1 so the number is negative.
               1     Subtract one.
0000FFFFFFFFFFC1

0000FFFFFFFFFFFF     Take the one's complement of the rightmost 48 bits.
0000FFFFFFFFFFC1
000000000000003E     The value is -3E.
```

FLOATING-POINT REPRESENTATION

The following are graphic representations of the CYBER 170 floating-point (real) format and the CYBER 200 floating-point format:



The exponent in the CYBER 200 floating-point format is not biased; the value of a floating-point number with exponent E and mantissa M is exactly M*(2**E).

Both CYBER 170 and CYBER 200 floating-points are normalized. In a normalized CYBER 170 floating-point, bit 47 is different from bit 59 (the sign bit). In a normalized CYBER 200 floating-point, bit 16 (the sign bit) is different from bit 17.

Converting to Normalized CYBER 200 Real Format

The following example converts the value 0.25 to normalized CYBER 200 floating-point format.

The decimal value 0.25 is 0.0100 binary or 0.4 hexadecimal. By shifting the decimal point 2 bits to the right, the value can be represented as 1.0 * 2**-2. Because the exponent is negative, it is converted to two's complement form as follows:

```
FFFF   Take the one's complement.
0002
FFFD
   1   Add one.
FFFE
```

The 16-bit exponent value is FFFE. The unnormalized floating-point number is as follows:

```
FFFE 0000 0000 0001
```

To normalize the floating-point number, the mantissa is shifted left until bits 16 and 17 differ. At the same time, the exponent is decremented by 1 for each bit shifted left. The normalized mantissa appears as follows (bit 16 is zero, bit 17 is 1):

```
4000 0000 0000
```

To move bit 1 to bit 17 requires 46 decimal shifts left (2E hexadecimal). Therefore, 2E is subtracted from the unnormalized exponent FFFE as follows:

```
FFFE
  2E
----
FFD0    Normalized exponent
```

Combining the normalized exponent and mantissa, the normalized form of the floating-point number 0.4 hexadecimal is as follows:

```
FFD0 4000 0000 0000
```

Converting From CYBER 200 Normalized Real Format

Consider the following CYBER 200 floating-point number:

```
FFD0 C000 0000 0000
```

Because the leftmost bit of the mantissa is a 1 (C is 1100 binary), the value is negative. Both the exponent and the mantissa, therefore, are in two's complement form. Convert each as follows:

```
FFD0    C000 0000 0000  }
   1                 1  }  Subtract one.
----    --------------  }
FFCF    BFFF FFFF FFFF  }

FFFF    FFFF FFFF FFFF  }  Take the one's
FFCF    BFFF FFFF FFFF  }  complement.
----    --------------  }
0030    4000 0000 0000  }
```

To convert the value to unnormalized form, shift the rightmost bit to the right and decrement the exponent until the exponent is 0000 or the bit has reached the rightmost bit of the word. (If you shifted further, the value would lose significance.) The unnormalized value appears as follows:

```
0002 0000 0000 0001
```

Remember that both the exponent and the mantissa are negative values translated from two's complement. The unnormalized number, therefore, represents -1.0 * 2**-2, -0.0100 binary or -0.4 hexadecimal.

Floating-Point Zero and Floating-Point Indefinite

Each floating-point number whose exponent begins with the digit 8 represents floating-point zero. Each floating-point number whose exponent begins with the digit 7 represents an indefinite value.

Unlike the CYBER 170 representations, floating-point zero and integer zero are not the same in a CYBER 200 machine:

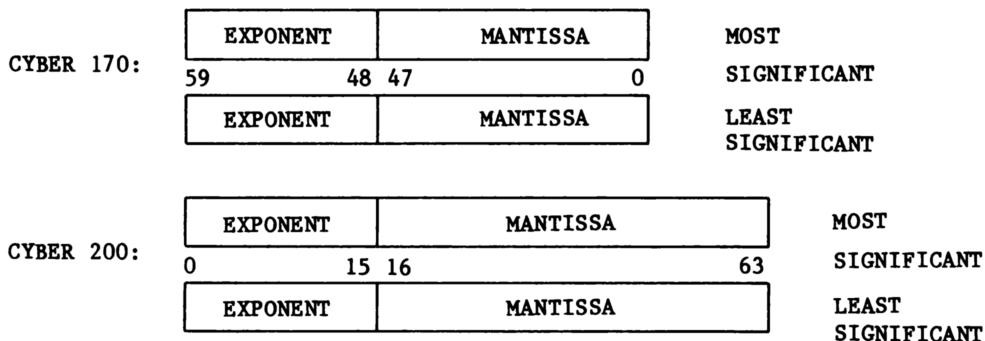
```
Floating-point zero:  8xxx xxxx xxxx xxxx
Integer zero:         0000 0000 0000 0000
```

This is significant when you initialize an element to zero. If initialized to integer zero, the value can be used for integer arithmetic operations, but not for floating-point arithmetic. If initialized to floating-point zero, the value can be used for floating-point arithmetic operations, but not for integer arithmetic.

It is recommended that the program perform its own initialization rather than use the system preset value. If the system clears space to all zeros, for example, it is a valid value for integer operations but it is not valid for floating-point operations.

DOUBLE-PRECISION REPRESENTATION

The following are graphic representations of the CYBER 170 double-precision format and the CYBER 200 double-precision format:

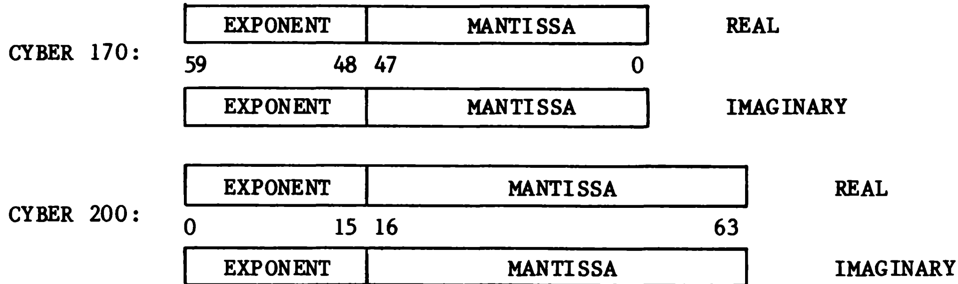


In general, a double-precision number is represented using two words in floating-point format. The first word represents the more significant half of the value, and the second word represents the less significant half of the value.

The exponent of the second word in the CYBER 200 double-precision format is the exponent of the first word minus 47. The mantissa of the second word is not normalized. The second word is always zero or positive; that is, bit 16 is zero even when the number is negative.

COMPLEX REPRESENTATION

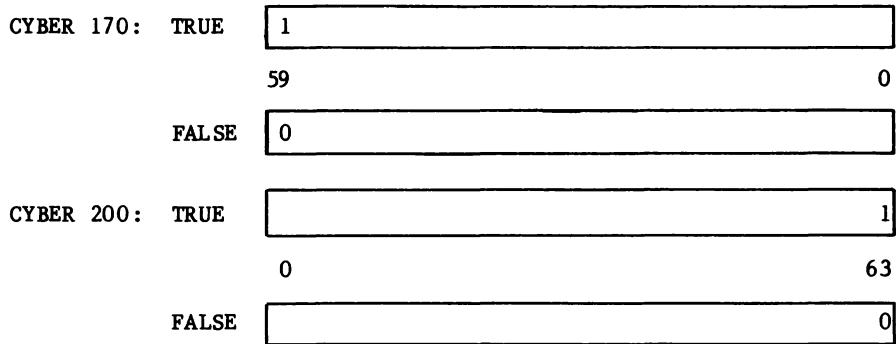
The following are graphic representations of the CYBER 170 complex format and the CYBER 200 complex format:



A complex value on either the CYBER 170 or the CYBER 200 is represented as two words in floating-point format. The first word represents the real part, and the second word represents the imaginary part of the complex value.

LOGICAL REPRESENTATION

The following are graphic representations of the CYBER 170 logical format and the CYBER 200 logical format:



Only bit 59 (the sign bit) is significant within a CYBER 170 word representing a logical value. If bit 59 is 1, the value is true; if bit 59 is 0, the value is false.

Only bit 63 is significant within a CYBER 200 word representing a logical value. If bit 63 is 1, the value is true; if bit 63 is 0, the value is false.

FORTRAN I/O 5

The File Management section of this manual showed how to connect a file to your FORTRAN 200 program. The section showed file I/O using READ, WRITE, and PRINT statements. However, FORTRAN programs executed on a CYBER 200 system have two methods of performing file I/O. These methods are as follows:

- File I/O using READ, WRITE, PRINT, and PUNCH statements (the standard FORTRAN runtime routines)
- File I/O using system interface language (SIL) calls.

As illustrated in figure 5-1, a FORTRAN program executes FORTRAN runtime I/O routines that call SIL I/O routines to issue system I/O requests.

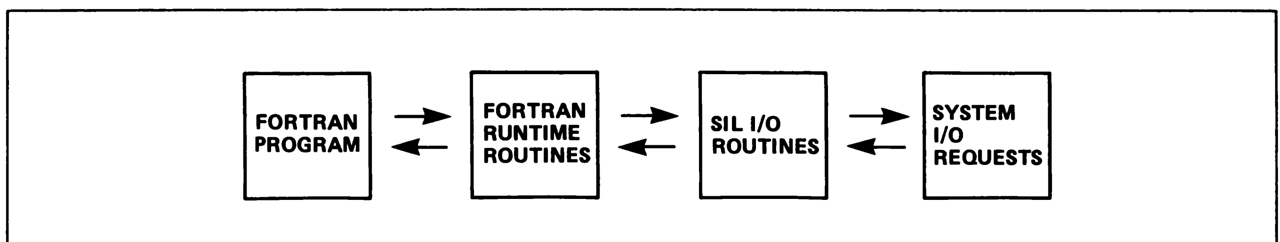


Figure 5-1. FORTRAN I/O Interface

The FORTRAN 200 Reference Manual describes file I/O using FORTRAN runtime routines. The same I/O methods you have used with other CDC FORTRAN compilers generally are available with FORTRAN 200. These I/O methods include the following:

- Formatted I/O
- Unformatted I/O
- List-directed I/O
- Namelist I/O
- Buffer I/O

A FORTRAN program can also call SIL I/O routines directly without the intervening FORTRAN runtime I/O routines. This allows the program to specify the file data format in greater detail. Problems may arise, however, if a program mixes the two I/O methods.

MIXING FORTRAN RUNTIME AND SIL I/O

Do not attempt to use both standard FORTRAN runtime I/O and SIL I/O routines to open the same file at the same time.

When a FORTRAN runtime routine opens a file, the routine associates the file with a unit identifier. The file remains associated with that unit identifier until the file is closed. SIL calls cannot reference a file by its unit identifier and, therefore, cannot reference the same instance of opening.

When a SIL call (Q5OPEN or Q5GETFIL) opens a file, the SIL call associates the file with a file logical unit number (FLUN) and a file information table (FIT). Subsequent SIL calls can reference the file by the FLUN and use the information in the file's FIT. A FORTRAN runtime routine, however, must first use a Q5REQUEST followed by an OPEN call to reference the file by its FLUN. FORTRAN runtime routines cannot reference a file by its FLUN or directly reference the information in the FIT. An OPEN statement that attempts to open the file would call Q5OPEN and receive an error status message because the file is already open.

Therefore, if your program first performs I/O on a file using SIL routines, issue a Q5CLOSE call to close the file before performing I/O on the file using FORTRAN runtime routines. Similarly, after performing file I/O using FORTRAN runtime routines, execute a CLOSE statement before opening the file for SIL I/O.

Figure 5-2 shows a program that opens and closes a file twice. The first time, SIL calls are used to open and close the file, and the second time, FORTRAN statements are used. A Q5GETFIL call creates a file named NEW and opens it, a Q5PUTN call writes a sequence of integers to the file, and a Q5CLOSE call closes the file. An OPEN statement reopens the file, a READ statement reads the integers from the file, and the CLOSE statement closes the file again.

Notice the RFLUN and FLUN parameters on the SIL calls. The RFLUN parameter returns the file logical unit number (FLUN) for the instance of open. The FLUN is an integer that identifies the instance of open. It is recommended that SIL calls identify an open file by its FLUN instead of by its file name (LFN) because this creates greater I/O efficiency. When a call specifies an open file by name, SIL must search for the FLUN associated with that name.

The Q5GETFIL call specifies W as the record type (RT) because that is the record type used in the file that the Q5GETFIL call will create by default. If the Q5GETFIL call references an existing file, this file is opened and used regardless of the record type used on the existing file.

Note that the Q5PUTN call specifies the length of the integers it writes (WSL=) as 8 bytes. SIL adds the control word delimiter to the data when the Q5PUTN call stores it in the buffer. In the second example, the Q5GETN transfers only 8 bytes (the integer) to the OUTARAY element.

The MDUMP calls are not required. They were added to display the contents of the WSA array.

Example 1:

```
      PROGRAM MIX
      INTEGER FLUN, WSA, INARAY(512), OUTARAY(512)
      COMMON /BUFFER/ INARAY, OUTARAY

      CALL Q5GETFIL('LFN=','MIXED','RFLUN=',FLUN,
     + 'BUF1=',INARAY,'BUFL1=',1,'RT=','W')
      DO 10 I=1,512
      WSA = I
   10 CALL Q5PUTN('FLUN=',FLUN,'WSA=',WSA,'WSL=',8)
      CALL Q5CLOSE('FLUN=',FLUN)

      OPEN(UNIT=1,FILE='MIXED')
      DO 20 I=1,512
   20 READ(UNIT=1) OUTARAY(I)
      CLOSE(UNIT=1)

      CALL MDUMP(OUTARAY,512,'Z',6HOUTPUT)
      STOP
      END
```

Example 2:

```
      PROGRAM MIX
      INTEGER FLUN, WSA, INARAY(512), OUTARAY(512)
      COMMON /BUFFER/ INARAY, OUTARAY

      OPEN(UNIT=1,FILE='MIXED')
      DO 10 I=1,512
      WSA = I
   10 WRITE(UNIT=1) WSA
      CLOSE(UNIT=1)

      CALL Q5GETFIL('LFN=','MIXED','RFLUN=',FLUN,
     + 'BUF1=',INARAY,'BUFL1=',1,'RT=','W')
      DO 10 I=1,512
   10 CALL Q5GETN('FLUN=',FLUN,'WSA=',OUTARAY(I),
     + 'WSL=',16)

      CALL Q5CLOSE('FLUN=',FLUN)


      CALL MDUMP(OUTARAY,512,'Z',6HOUTPUT)

      STOP

      END
```

Figure 5-2. Mixed I/O Examples

RECORD I/O

Record I/O is the reading and writing of data records. The data records exist on the file within a logical record structure. A request to read a record extracts a record of data from the logical structure. A request to write a record writes the data and any record delimiters required by the logical structure.

Either FORTRAN runtime routines or SIL calls can perform record I/O.

FORTRAN RUNTIME RECORD I/O

The READ, WRITE, PRINT, and PUNCH statements perform record I/O. Each statement reads or writes one or more data records.

The FORTRAN runtime routines read and write data within a SIL record structure. However, this record structure is transparent to the program when using FORTRAN runtime routines. Delimiters are added or removed without program specification.

However, as shown in figures 3-6 and 3-7, you can see the SIL record delimiters if you request a dump of the file data. If the file is a sequential-access file that, by default, uses the record type W format (variable-length records with control word delimiters), you can see the delimiters embedded in the data. If the file is a direct-access file that, by default, uses the record type F format (fixed-length records with no delimiters), you can see the padding characters added to short records.

SIL RECORD I/O

The SIL calls that perform record I/O are called get and put calls because they either read (get) or write (put) a data record. These calls can read or write a single record, the next record in a group of records, or a partial record.

Get Calls	Put Calls	Record Read/Written
Q5GETB	Q5PUTB	A single record
Q5GETN	Q5PUTN	The next record
Q5GETP	Q5PUTP	A partial record

Using the Q5GETP and Q5PUTP calls for partial record I/O, the program can read and write records longer than the working storage area in the program. The call specifies which part of the record is being read or written.

A Q5ENDPAR call can write a group delimiter to a group of records as they are written.

SIL calls do not perform formatted I/O. Data is read or written without data editing.

SIL calls allow tailoring of the file structure by specifying the record type and/or blocking type. The available record and blocking types are described in the VSOS Reference Manual, Volume 1.

ACCESS METHODS

Sequential access is the default access method. In sequential access, records are accessed in the order they were written. A request to read or write a record positions the file to read or write the next record.

You can request direct access on a FORTRAN OPEN statement or on a Q5OPEN or Q5GETFIL call. When any of these statements are used to open a file, records are accessed by number.

The ACCESS parameter on the OPEN statement or the SFO= parameter on the Q5OPEN or Q5GETFIL call requests direct access.

Direct access normally uses record type F (fixed-length records with no delimiters). If a different record type is used, direct access may require specification of record length. The OPEN statement specifies the record length using the RECL= parameter. The Q5OPEN or Q5GETFIL call specifies the record length with its MXR= parameter.

Each read or write request specifies the record number. The record number is specified on the REC= parameter of the READ statement or the REC= parameter on the Q5GETN call.

The direct access program example in figure 5-3 lists a program that opens a file for direct access and writes a sequence of integers on the file. The program then reads the records in reverse order, copying the integers to an array. The MDUMP call prints the contents of the array showing the integers in reverse order.

Note that the RESOURCE statement in figure 5-3 specifies a larger time limit than previous jobs in the manual. A longer time limit is required because the file is repositioned for each read.

```
ADEY,ST=ABC.
USER,USER=123456,ACCOUNT=ACCT933.
RESOURCE,TL=25.
FTN200.
LOAD.
GO.
(EOR)
      PROGRAM DIRECT
      INTEGER X, WSA(512)
C
      OPEN(UNIT=1,FILE='NEW',ACCESS='DIRECT',RECL=8)
      DO 10 I=1,512
      X=I
   10 WRITE(1, REC=I) X
C
      DO 20 I=1,512
      N = 513 - I
   20 READ(1, REC=N) WSA(I)
C
      CALL MDUMP(WSA,512,'Z',6HOUTPUT)
      STOP
      END
```

Figure 5-3. Direct Access Program Example

DATA TRANSFER

As illustrated in figure 5-4, record I/O transfers data between a working storage area and a data buffer in the program. A read request reads a record from the buffer. A write request writes a record to the buffer. The system manages the transfer of data between the buffer and the file; that is, when writing data, the system waits until the buffer is full before writing the data to the file. When reading data, the system fills the buffer when required by the program.

Figure 5-4 also illustrates block I/O transfer. In this case, the program allocates only a data buffer. There is no working storage area because the program requests the transfer of data only between the buffer and the file; that is, a read request fills the entire buffer. A write request writes the entire buffer.

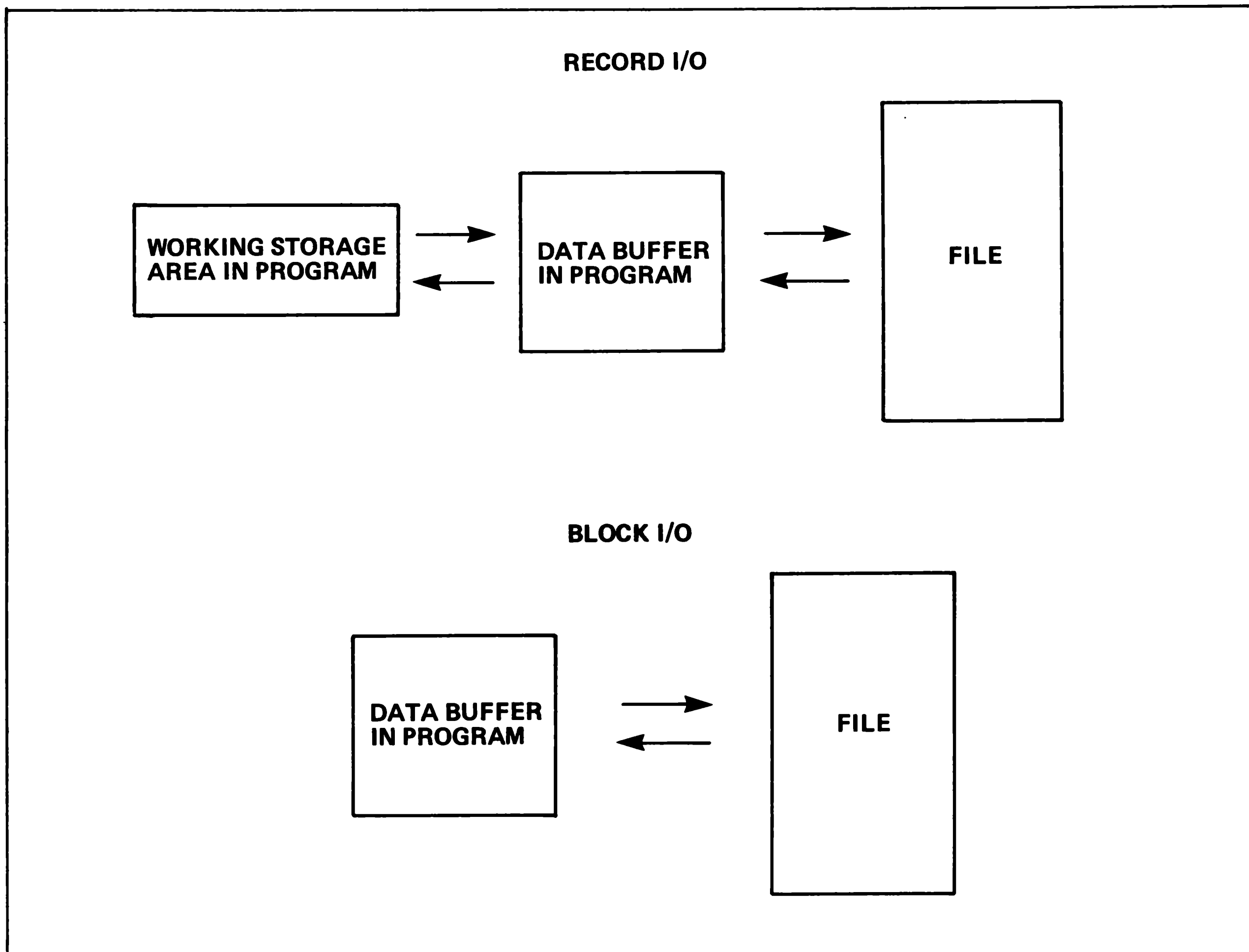


Figure 5-4. Explicit I/O Data Transfers

Note that the data transferred between the data buffer and the file includes embedded record delimiters. A request to read a record (that is, transfer a record from the data buffer to the working storage area) removes the record delimiter. A request to write a record (that is, transfer a record from the working storage area to the data buffer) adds the record delimiter.

BLOCK I/O

Block I/O allows your program to perform concurrent I/O. Block I/O means the program can issue a call to read or write data and then perform other computations while the data requested is being read or written. When the program is ready to use the data that was just read or the program is ready to write more data, it checks to see if the previously issued I/O request is completed.

Like record I/O, either FORTRAN runtime routines or SIL calls can perform block I/O. The FORTRAN runtime routines that perform block I/O are Q7BUFIN, Q7BUFOUT, and Q7SEEK.

The SIL calls used to perform block I/O are Q5READ, Q5WRITE, and Q5CHECK. Q5CHECK can return the status of an I/O request (whether the request has been completed or not). Q5CHECK can also suspend program execution until an I/O request is completed. If it is possible that more than one I/O request is outstanding, the Q5CHECK call should identify the request by specifying the request number (RSN) returned by the Q5READ or Q5WRITE call.

Figure 5-5 illustrates a job that compiles, loads, and executes a program that opens file OLD, reads 16 blocks of data, and then writes the data on file NEW. While reading data from file OLD, the program executes a subprogram that does not use the data being read. When the subprogram completes, Q5CHECK suspends program execution until the read request completes.

```
ADEY,ST=ABC.
USER,USER=123456,ACCOUNT=ACCT933.
RESOURCE,TL=10.
ATTACH,OLD.
FTN200.
LOAD,GRSP=*BUFFER.
GO.
(EOR)
      PROGRAM BLOCCPY
      INTEGER FLUN, FLUN2, RSN, BIGARAY(8192)
      COMMON /BUFFER/ BIGARAY

      CALL Q5OPEN('LFN=','OLD','RFLUN=',FLUN)
      CALL Q5READ('FLUN=',FLUN,'BUFFER=',BIGARAY,
     + 'BUFLEN=',16,'RSN=',RSN)
      CALL SUBPROG
      CALL Q5CHECK('FLUN=',FLUN,'RSN=',RSN,'WAIT')
      CALL Q5GETFIL('LFN=','NEW','RFLUN=',FLUN2')
      CALL Q5WRITE('FLUN=',FLUN2,'BUFFER=',BIGARAY,
     + 'BYTCNT=',65536)
      STOP
      END
```

Figure 5-5. Block I/O Example

Notice that BIGARAY is in a common block. A common block is required to be aligned on a page boundary and to be a multiple of 512 words. To align the common block on a page boundary, specify the common block on a GRSP= or GRLP= parameter on the LOAD statement. The GRSP=*BUFFER parameter on the LOAD statement aligns common block BUFFER on a small page boundary.

The Q5READ call specifies the length of the data read as sixteen 512-word blocks using the BUFLEN= parameter. The Q5WRITE call specifies the length of the data written as 65536 bytes using the BYTCNT= parameter.

CYBER 200 MEMORY MANAGEMENT 6

The CYBER 200 Computer is a virtual memory machine. This means that each program has its own virtual address space in which it can reference any virtual address between 4000 and 800 000 000 000 hexadecimal. (The top of this range is increased to C00 000 000 000 hexadecimal when the shared library is active.) While this fact suggests that the program has unlimited virtual memory available, there are limitations to the amount of virtual memory the program can use.

A virtual memory addressing scheme requires that every virtual address referenced by a program must be mapped to a location on a disk file. The virtual memory available to the program, therefore, is limited by the disk and map space available to the program.

The operating system limits the physical memory available to your program. The operating system must determine if there is enough physical memory available to schedule a program for execution. The operating system must also determine when an executing task should get any additional physical memory it needs.

Memory management is important to your job. With efficient memory management, your job requires fewer system resources and completes faster. To understand how you can improve the efficiency of your program, you must understand memory paging.

MEMORY PAGING

Program space (or virtual space) is the range of virtual addresses used by the execution of the program. When the program is executed, the virtual space is mapped to physical space (central memory and disk space). During execution, the program is assigned central memory only for the code and the data the program is currently using. The rest of the code and data remain on disk. When the program requires additional code or data that is not currently in central memory, VSOS automatically copies it from disk to central memory. The process of copying code and data in and out of central memory is called paging.

VSOS uses two page sizes: small pages and large pages. Small page size is determined by site. A small page can be one, four, or sixteen 512-word blocks. Large page size is always 128 512-word blocks (65536).

A virtual address specification is often required to be on a block or page boundary.

- A virtual address is on a block boundary if the address is a multiple of 8000 hexadecimal
- A virtual address is on a large-page boundary if the address is a multiple of 400000 hexadecimal
- A virtual address is on a small-page boundary if the address is one of the following:
    - a multiple of 8000 hexadecimal for a 1-block page
    - a multiple of 20000 hexadecimal for a 4-block page
    - a multiple of 80000 hexadecimal for a 16-block page

A page fault occurs when an executing program references a virtual address whose contents are not currently in physical memory. A page fault suspends program execution until the system copies the contents of the referenced virtual address into physical memory. This, however, is not always just a one-page copy. It may be a two-page copy if the system must first save the page currently in memory on a disk to make room for the new page and then copy the new page contents from disk to the newly available page.

To minimize the execution time of your program, you should minimize page faults. You should also consider whether the page faults are for large pages or small pages. A page fault for a large page takes much longer than a page fault for a small page. For large volumes of data, however, paging in one large page is more efficient than paging in the corresponding number of small pages because less I/O overhead is incurred.

The SUMMARY control statement can tell you how many small page and large page faults your job has required. Figure 6-1 shows an example of the SUMMARY output contained in a job dayfile. This job required no large page faults; therefore, the Number of Large Page Faults line is omitted from the SUMMARY output.

```
16.11.43  SUMMARY.
16.11.43  SYSTEM TIME UNITS (STU)                  5.273
16.11.43  USER CPU TIME (SECS)                     1.053
16.11.43  SYSTEM CPU TIME (SECS)                   1.628
16.11.43  USER MEMORY USAGE (PAGE*SECS)          393.512
16.11.43  USER AVERAGE WORKING SET SIZE (PAGES)  373
16.11.43  NUMBER OF VIRTUAL SYSTEM REQUESTS      338
16.11.43  NUMBER OF SMALL PAGE FAULTS             65
16.11.43  NUMBER OF DISK I/O REQUESTS             20
16.11.43  NUMBER OF DISK SECTORS TRANSFERRED     147
```

Figure 6-1. SUMMARY Output Example

PROGRAM COMPONENTS

A program consists of the executable statements that indicate what the program does and the data structures that indicate how the program references data. These statements are often split between two or more subprograms. The compiler transforms the statements in each subprogram into a code module.

Data structures are either local variables accessible by only one module or common blocks accessible by more than one module. FORTRAN common blocks can be named or unnamed. (Named common is also called labeled common. Unnamed common is also called blank or unlabeled common.)

If you specify the M option on the LO= parameter of the FTN200 control statement, the FORTRAN 200 compilation writes a register file map and a storage map for the program. The register file map lists the contents of the 256-register register file. The storage map lists the components of the FORTRAN 200 program.

Figure 6-2 shows a sample storage map. (The program whose compile generated the sample storage map is shown in figure 6-6.) Storage maps list the following items:

1. Program name
2. Starting address of the data area copy for all registers
3. Name, location, class, and data type of all scalars, constants, and externals assigned to registers
4. Name, location, and class of descriptors assigned to registers
5. Length and starting address of the object code
6. Length and starting address of character constants, literals, and format segments
7. Length and starting address of argument vectors
8. Length and starting addresses of constants, externals, descriptors, variables not in common, namelist groups, and character scalars not assigned to registers
9. Location of temporary storage
10. Common blocks
11. Entry points
12. Externals

GENERATING THE EXECUTABLE FILE

The LOAD utility generates the controllee file by combining the components of the program as specified on the LOAD statement. These components include code modules and common blocks.

Code modules are the modules copied from the object code files specified on the LOAD statement and the modules that load copies from object libraries to satisfy external references. In the previous job examples, the LOAD statement has copied code modules from the default object code file (BINARY) written by the preceding FORTRAN 200 compilation.

LOAD attempts to satisfy external references by copying code modules from the F200LIB object library. If no code module in the F200LIB library has an entry point matching an unsatisfied external reference, LOAD looks for the entry point on the SYSLIB library file.

As shown in figure 6-3, the load map lists the code modules copied to the controllee file. The load map lists the module name, its address within the controllee file, its length, and the file from which the module was copied.

```
FORTRAN 200 CYCLE L670      BUILT 06/08/86 17:08   STORAGE MAP      SEQ        COMPILED 09/14/86 13:50              PAGE 3
                                   (1)PROGRAM NAME IS SEQ           TOTAL LENGTH IS       35 HEX HALF WORDS

 (2)DATA AREA COPY OF ALL REGISTERS USED BY THIS FORTRAN PROGRAM
       START ADDRESS =         A00                                   (START ADDRESS IS RELATIVE TO DATA AREA BASE ADDRESS)

 (3)SCALARS AND CONSTANTS ASSIGNED TO REGISTERS                      (LOCATIONS ARE RELATIVE TO DATA AREA BASE ADDRESS)
     LOCATION   REG.NO            NAME                           CLASS              TYPE

        A80      22      PI_DYNSP                            SIMPLE VARIABLE        INTGR
        AC0      23      C_#200                              CONSTANT               INTGR
        B40      25      D_L_0000                            SIMPLE VARIABLE        INTGR
        B80      26      I                                   SIMPLE VARIABLE        INTGR

 (4)DESCRIPTIONS ASSIGNED TO REGISTERS                               (LOCATIONS ARE RELATIVE TO DATA AREA BASE ADDRESS)
     LOCATION   REG.NO            NAME                           CLASS

       C00        28     BIGARAY 8F_DESCR                    ARRAY NAME
       C40        29     *ARG_VECT                           ARGUMENT VECTOR
       C80        2A     *ARG_VECT                           ARGUMENT VECTOR

      NOTE: TOTAL NUMBER OF REGISTERS TO BE FETCHED INTO REG.FILE STARTING WITH REG.20 HEX IS 0B HEX

 (5)GENERATED OBJECT CODE
       START ADDRESS =          0       LENGTH =      35 HEX HALF WORDS     (START ADDRESS IS RELATIVE TO CODE AREA BASE ADDRESS)

 (6)CHARACTER CONSTANTS,LITERALS AND FORMAT SEGMENTS
       START ADDRESS =          0       LENGTH =      12 HEX HALF WORDS     (START ADDRESS IS RELATIVE TO DATA AREA BASE ADDRESS)

 (7)ARGUMENT VECTORS
       START ADDRESS =          240     LENGTH =      1A HEX HALF WORDS     (START ADDRESS IS RELATIVE TO DATA AREA BASE ADDRESS)

 (8)CONSTANTS, DESCRIPTORS, NON-COMMON VARIABLES, AND NAMELISTS NOT ASSIGNED TO REGISTERS
       START ADDRESS =           580    LENGTH =       0 HEX HALF WORDS     (START ADDRESS IS RELATIVE TO DATA AREA BASE ADDRESS)

 (9)  LOCATION     SYMBOLIC NAME OR HEX VALUE         CLASS          TYPE  (LOCATIONS ARE RELATIVE TO DATA AREA BASE ADDRESS)

         500            RLEN                     SIMPLE VARIABLE     INTGR
         600            F_PROLOG                 REF.EXTERNAL SUBPR  UNKNW
         680            Q5MAPIN                  REF.EXTERNAL SUBPR  REAL
         700            Q5GETFIL                 REF.EXTERNAL SUBPR  REAL
         780            F_EPILOG                 REF.EXTERNAL SUBPR  INTGR
         800                   80                CONSTANT            INTGR

      TEMPORARY STORAGE
                                        LENGTH =       0 HEX HALF WORDS     (STORAGE IS SCATTERED THROUGHOUT DATA AREA)


    COMMON BLOCKS
```

Figure 6-2. Storage Map Example (Sheet 1 of 2)

```
FORTRAN 200 CYCLE L670      BUILT 06/08/86 17:08   STORAGE MAP      SEQ      COMPILED 09/14/86 13:50                PAGE 4
(10) COMMON BLOCKS

   BLANK COMMON NAME BLOCK

     BLANK COMMON NAME BLOCK IS NOT SPECIFIED

   NAMED COMMON BLOCK BUFFER

     START ADDRESS =              0      LENGTH =    20000 HEX HALF WORDS

     LOCATION     SYMBOLIC NAME          CLASS           TYPE (LOCATIONS ARE RELATIVE TO COMMON BLOCK START ADDR.)

           0            BIGARAY      ARRAY VARIABLE      INTGR

(11) LIST OF ALL ENTRY POINTS

      LOCATION     SYMBOLIC NAME                                       (LOCATIONS ARE RELATIVE TO DATA AREA BASE ADDRESS)

            0        SEQ

(12) LIST OF ALL EXTERNALS

                   SYMBOLIC NAME                                       (LOCATIONS ARE RELATIVE TO DATA AREA BASE ADDRESS)

                     F EPILOG
                     Q5GETFIL
                     Q5MAPIN
                     F_PROLOG
        NO ERRORS
```

Figure 6-2. Storage Map Example (Sheet 2 of 2)

Beside each code module listed in the load map is a description of the database for the module. A database is space for data constants and variables accessed by a module. Unless the LOAD statement specifies GDWC=NO, LOAD copies the database for a module immediately after the module in the controllee file. This is done so that the code and the database used by the code are grouped together in virtual memory. This improves the chances that the code can access its database and bring in the data without causing a page fault.

After the code modules and databases, LOAD reserves space for the named common blocks. If a DATA statement initializes variables in the common block, LOAD initializes the common block space in the executable file to the specified values.

LOAD records an entry for blank common but never reserves space for blank common in the controllee file. Blank common space is always allocated in the drop file.

If the GROS or GROL option is specified, LOAD can also allocate named common blocks to the drop file without reserving space in the controllee file.

The load map in figure 6-3 is a listing of how memory was allocated by the loader for your job. The listing shows the following items:

1. Loader update level
2. Job-specified parameters
3. Name, location, and other information about all object modules and data bases in the controllee file
4. Compilation/assembly date and time of memory allocation for object modules and data bases
5. Name, address, and word length of all common blocks
6. Address and word length of all error processing information
7. Address and block length of bound virtual map entries
8. Address and block length of drop file map entries
9. Address of the dynamic stack (Unless specified otherwise by the DSA parameter, the dynamic stack is always allocated following the last virtual address allocated when the task executed.)
10. Size of the controllee file in blocks

LOAD MAP

(1) LOAD R2.3 CYCLE L670

(2) PARAMETERS SPECIFIED: GROL=*BUFFER,LIB=F200LIB

(3) MODULES

NAME	CODE ADDRESS	WORD LENGTH(HEX)	FILE	DATA BASE ADDRESS	WORD LENGTH (HEX)	PROCESSORS	(4) DATE	TIME
SEQ	80080	1E	BINARY	80780	34	FTN200	84/9/14	13:50
F_EPILOG	81500	7C	F200LIB	83380	1E	META	84/6/2	17:24
Q5GTFIL	83880	1A2	SYSLIB	8A380	7E	IMPL	84/9/6	17:45
Q5MAPIN	95080	AE	SYSLIB	97880	50	IMPL	84/9/6	17:45
F_PROLOG	98880	152	F200LIB	A0C80	58	META	84/9/2	17:24
Q5CLOSE	A2300	226	SYSLIB	AAC00	66	IMPL	84/9/6	17:45
Q5REDUCE	81080	88	SYSLIB	83200	48	IMPL	84/9/6	17:45
R5RETURN	B5D80	AE	SYSLIB	B8880	4C	IMPL	84/9/6	17:45
Q5SNDMCR	BBB00	24	SYSLIB	BC380	A0	META	84/9/6	17:32
Q5TERM	BEC00	36	SYSLIB	BF900	9E	META	84/9/6	17:32
Q5 DCDCK	C2100	11E	SYSLIB	C6800	46	IMPL	84/9/6	17:45
Q5LFIHIR	C8080	60	SYSLIB	C9800	A2	META	84/9/6	17:32
Q5DCDPFI	CC100	234	SYSLIB	D4D80	E4	META	84/9/6	17:32
.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.
Q5 PROER	1DDB00	4C	SYSLIB	1DED80	0	META	84/9/6	17:32
Q5 PTNUP	1DEE00	28	SYSLIB	1DF780	0	META	84/9/6	17:32
Q5_MVEDL	1DF800	1C	SYSLIB	1DFE80	0	META	84/9/6	17:32
Q5_SELSS	1DFF00	32	SYSLIB	1E0B00	0	META	84/9/6	17:32
Q5LSTCH	1E0B80	B0	SYSLIB	1E3700	40	IMPL	84/9/6	17:54
Q5GENFIT	1E6C80	1FA	SYSLIB	1EEA80	6E	IMPL	84/9/6	17:45
F_DFDMP	1F6100	2E	F200LIB	1F6C00	5A	META	84/6/2	17:25
Q5GETMPG	1F8300	2E	SYSLIB	1F8E00	2E	IMPL	84/9/6	17:54
Q5RECALL	1FA900	32	SYSLIB	1FB500	2E	IMPL	84/9/6	17:54
Q5PUTP	1FD000	596	SYSLIB	213500	DA	IMPL	84/9/6	17:45
F CPOPEN	21A900	F2	F200LIB	21E500	66	META	84/6/2	17:22
Q5SKIP	21FF00	716	SYSLIB	23C400	A4	IMPL	84/9/6	17:45
Q5READ	241400	316	SYSLIB	24D900	78	IMPL	84/9/6	17:45
Q5GETFIT	252600	56	SYSLIB	253B00	42	IMPL	84/9/6	17:45

MODULE HEADER ADDRESS IS CODE ADDRESS MINUS #80.

COMMON BLOCKS

(5) NAME	ADDRESS	WORD LENGTH HEX	WORD LENGTH DECIMAL
$Q5GTFIL	8C300	234	564
$Q5MAPIN	98F80	A2	162
$Q5CLOSE	AC580	12A	298
$Q5REDUC	B4400	64	100
$Q5RETUR	B9B80	7C	124
.	.	.	.
.	.	.	.
.	.	.	.
$Q5CHECK	1A9F00	AC	172
$Q5CLIOE	1B1B80	9A	154
$Q5ENDPA	1C3980	96	150
$Q5WRITE	1D5C00	BE	190

Figure 6-3. Load Map Example (Sheet 1 of 2)

$Q5RETFI	1DAF80	3C	60
$Q5LSTCH	1E4700	94	148
$Q5GENFI	1F0600	16A	362
$Q5GETMP	1F9980	3C	60
$Q5RECAL	1FC080	3C	60
$Q5PUTP	216B80	F4	244
$Q5SKIP	23ED00	9A	154
$Q5READ	24F700	BA	186
.	.	.	.
.	.	.	.
.	.	.	.
Q5_CLFHI	292180	64	100
Q5_MNE01	293A80	32	50
Q5_EMS04	294700	E	14
Q5_DCDPF	294A80	21C	540
Q5_MNE02	29D180	DC	220
Q5CGETTN	2A0880	4C	76
Q5CGETCR	2A1880	7E	126
Q5BETA	2A3B00	24	36
99415048	300000	C00	3072
99575341	330000	6000	24576
99464954	500000	1B80	7040
55000000	56E000	8010	32784
99434642	76E400	FA	250
BUFFER	800000	10000	65536

ERROR PROCESSING INFORMATION

(6) ADDRESS	WORD LENGTH (HEX)
2A4400	AA7

BOUND VIRTUAL MAP ENTRIES

(7) ADDRESS	BLOCK LENGTH (HEX)
80000	50

DROP FILE MAP ENTRIES

(8) ADDRESS	BLOCK LENGTH (HEX)
500000	50
300000	40

(9) DYNAMIC STACK ADDRESS = 1000000

(10) CONTROLLEE FILE SIZE (BLOCKS)

HEX	DECIMAL
52	82

Figure 6-3. Load Map Example (Sheet 2 of 2)

Figure 6-4 shows how a controllee file is mapped into its virtual address space.

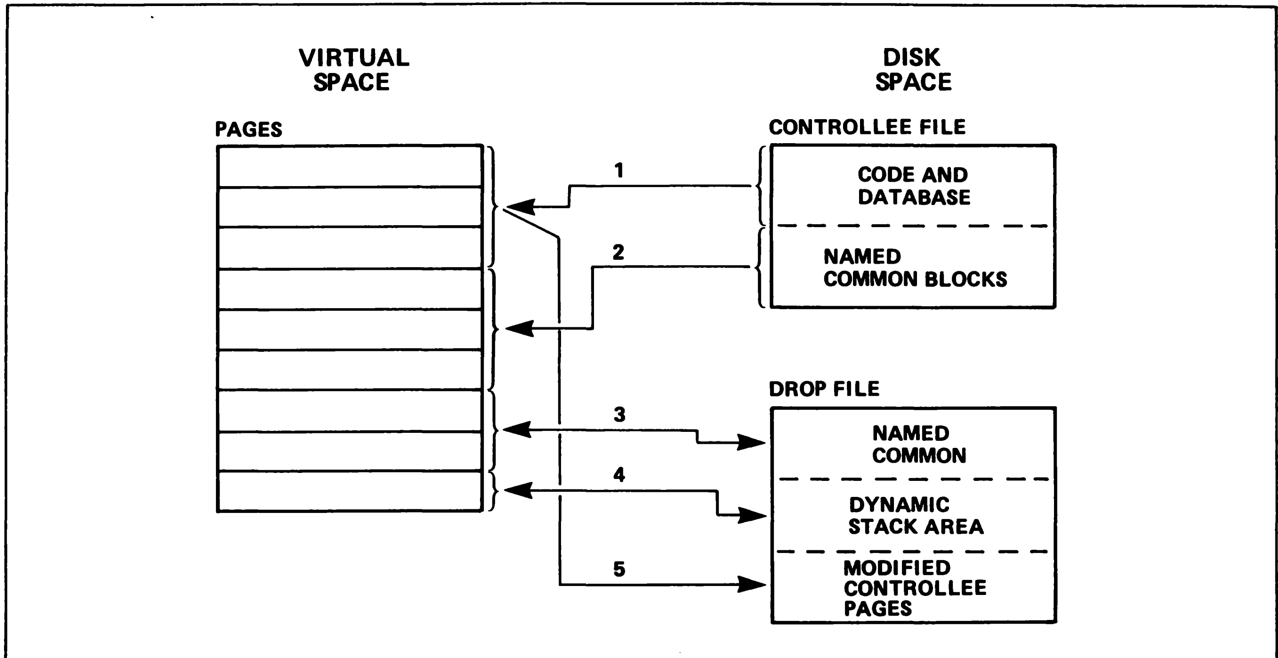


Figure 6-4. Task Virtual Space Mapping

Mapping occurs as follows:

1. By default, the controllee file is mapped to small pages.

2. Named common is mapped to small pages.

3. Space is allocated for named common in virtual space and in the task's drop file.

4. Room for the dynamic stack area is allocated in virtual space and in the drop file. The dynamic stack area is scratch space used for stacking register contents on subroutine calls and for compiler-generated intermediate vectors. (Vectors can also be explicitly assigned to the dynamic stack area using the descriptor ASSIGN statement.)

5. During task execution, if the system must page out modified pages from the controllee file, the modified pages are copied to the drop file, not to the controllee file. (The controllee file is not modified during execution.)

USER-CONTROLLABLE PAGE MAPPING

Page mapping does not have to be controlled entirely by VSOS. You can determine the way data in your program is mapped to virtual space.

Suppose your program declares three common blocks: A, B, and C. One part of the program uses the arrays in common block B and another part of the program uses the arrays in common blocks A and C. It may be worthwhile to tell the system to group common blocks A and C together when mapping the controllee into virtual space. This grouping indicates that blocks are combined for one mapping that begins on a page boundary. (Grouping does not indicate any ordering of the common blocks in memory.)

To tell the system to group common blocks A and C, specify these blocks on the grouping parameter of the LOAD statement. If common blocks A and C are too small to use a large page efficiently, specify them on the GRSP parameter as follows:

```
GRSP=*A,*C
```

If common blocks A and C are large enough to use a large page efficiently, you should specify them on the GRLP parameter as follows:

```
GRLP=*A,*C
```

If the unnamed common block is very large, it may be more efficient if the block is mapped to large pages. To specify large pages, include the following parameter on the LOAD statement:

```
GRLP=*
```

To reduce the size of the controllee file, specify the common blocks on a GROS or GROL parameter. Use GROS if the blocks should use small pages. Use GROL if the blocks should use large pages. If a named common block is not initialized by a DATA statement, it need not have space reserved in the controllee file.

For common blocks specified on GROS or GROL parameters, LOAD records an entry describing the common block in the controllee file but does not reserve space for the block. No space is allocated for the block until the block is actually referenced by the executing program. The memory allocated for the common block requires corresponding space on a disk file. The allocated disk space is in the drop file, not in the controllee file. This process, which results in a shorter controllee file and in a longer drop file, is illustrated in figure 6-5.

Normally, dynamic stack space is mapped to small pages. If your program uses many vector operations, however, dynamic stack space may use a large page efficiently. The specification to map dynamic stack space to a large page is on the PROGRAM statement of your program, not on the LOAD statement in your job.

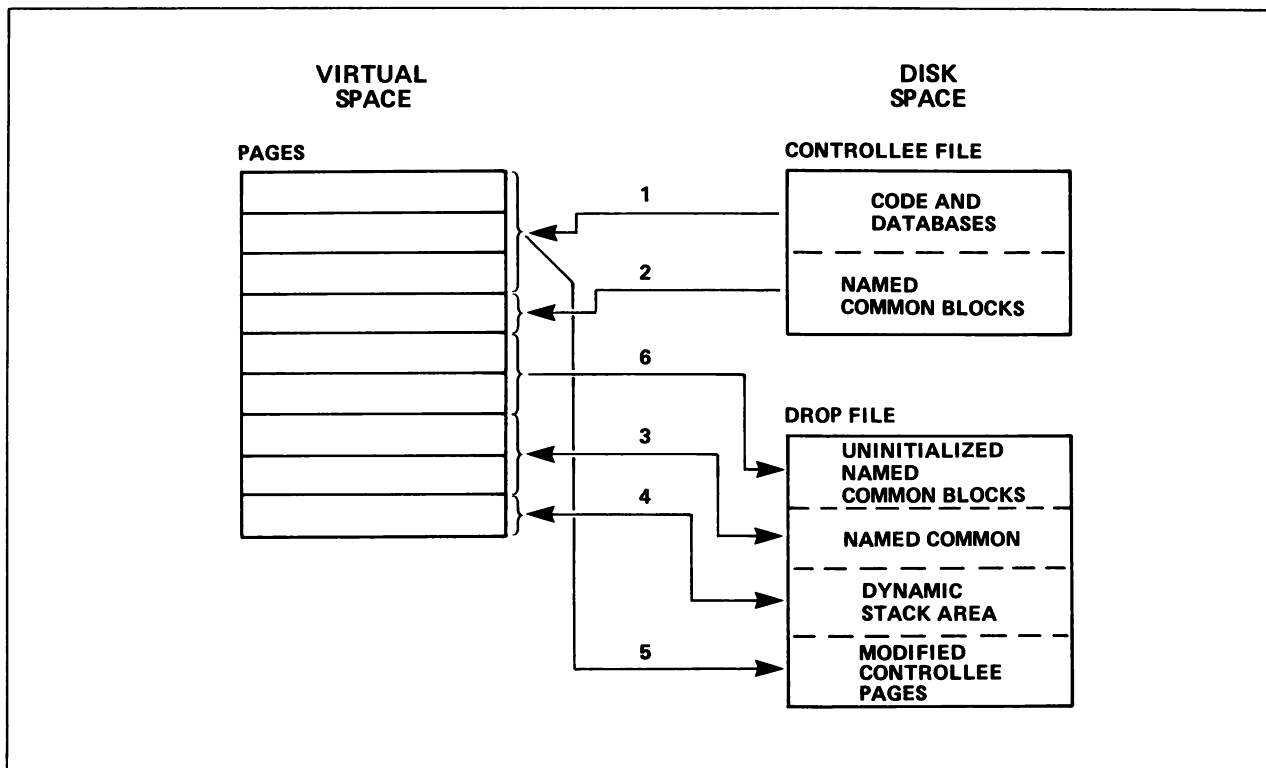


Figure 6-5. Mapping Uninitialized Common to the Drop File

In figure 6-5, the mapping occurs as follows:

1. By default, the controllee file is mapped to small pages.

2. Named common is mapped to small pages.

3. Space is allocated for named common in virtual space and in the task's drop file.

4. Room for the dynamic stack area is allocated in virtual space and in the drop file.

5. During task execution, if the system must page out modified pages from the controllee file, the modified pages are copied to the drop file, not to the controllee file. (The controllee file is not modified by its execution.)

6. Uninitialized named common is mapped to the drop file. (Because of this, use of the GROS and GROL parameters can increase the size of the drop file.)

ADVISING THE SYSTEM OF MEMORY REQUIREMENTS

The previous job examples have used the RESOURCE statement to specify a time limit for the job and to reserve tape drives. The RESOURCE statement can also advise the system of job memory requirements.

Each site can set up job categories that the system uses in job scheduling. The site specifies time and memory limits for each job category. Specifying a job category with the JCAT parameter on the RESOURCE statement changes the maximum limits of the job from the default job category limits to the limits of the specified category. Specifying a category whose limits are lower than the default limits could result in quicker scheduling of your job, but the job must execute within those limits.

The RESOURCE statement provides automatic job category selection based on WS= (small page memory), LP= (large page memory), and TL= (time limit) specifications.

If you know that the memory requirements for your job are higher than what is allowed by the job category, you can also use the WS=parameter to specify that your job requires all available memory in the machine by specifying the following:

```
WS=*
```

If the job uses large pages, you can advise the system of the maximum number of large pages required with the LP=parameter on the RESOURCE statement.

You can advise the system of changes in the job memory requirements with the SET control statement. A Q5SETLP call can specify changes in large-page requirements within a program. Using a Q5ADVISE call, your program can tell the system which large arrays it can page out because the arrays are not currently needed and which arrays it should leave paged in because these arrays are needed shortly. The SET statement and the Q5SETLP and Q5ADVISE calls are described in the VSOS Reference Manual, Volume 1.

IMPLICIT I/O

A program can use the paging mechanism of the system to perform file I/O. This process is called implicit I/O and requires no explicit READ or WRITE statements. Instead, the program calls Q5MAPIN to associate an array with a disk file. (This array must not be allocated on the controllee file or mapped to the drop file prior to the Q5MAPIN call.) Similarly, a program can call Q5MAPOUT to end the association and move the data back to the disk file.

While an array is associated with a file, the program reads file data by referencing the array and writes file data by storing data in the array. This is possible only if the array is in a common block and that common block is not in the controllee file. Blank common is never in the controllee file. If the common block is a named common block, you must specify the common block on the GROS or GROL parameter on the LOAD statement so that LOAD does not reserve space for the common block in the controllee file.

Figure 6-6 shows a job that compiles, loads, and executes a FORTRAN 200 program that performs implicit I/O on a file named TEST. The job defines the file TEST and then writes a sequence of integers on the file.

```
ADEY,ST=ABC.
USER,USER=123456,PASSWORD=XYZ.
RESOURCE,TL=20.
DEFINE,TEST/128.
FTN200.
COMMENT. THE BUFFER ARRAY IS MAPPED TO FILE TEST.
LOAD,GROL=*BUFFER.
GO.
(EOR)
      PROGRAM SEQ
C
      INTEGER BIGARAY, LEN, RLEN
      DIMENSION BIGARAY(65536)
      COMMON / BUFFER / BIGARAY
C
C  Opens a file named TEST for implicit I/O.
C  The requested file length is specified by LEN;
C  the initial file length is returned in RLEN.
      CALL Q5GETFIL('LFN=',4HTEST, 'LEN=',128,
     +'RLEN=',RLEN,'IMP')
C
C  Associates file TEST with array BIGARAY.
      CALL Q5MAPIN('LFN=',4HTEST,'VBA=',BIGARAY,
     +'LEN=',RLEN)
C
C  Writes a sequence of integers in TEST.
      DO 10 I=1,RLEN*512
   10 BIGARAY(I) = I
      STOP
      END
```

Figure 6-6. Implicit I/O Job Example

A job to read file TEST using implicit I/O would attach the file and then execute a program to read the file. The program could use the same Q5GETFIL and Q5MAPIN calls used in the program that wrote the file. Once the file is mapped to an array, the program can reference the contents of the array as the contents of file TEST.

An array specified on a Q5MAPIN call must be in a common block specified on a GROS or GROL parameter in the LOAD statement. An attempt to map in an array that is not specified on a GROS or GROL parameter returns the following message:

```
F Q5MAPIN 1519 VIRTUAL ADDRESS OVERLAP ON FILE TEST
```

NOTE

While GROS and GROL allow Q5MAPIN, they do not require it.

SYSTEM SHARED LIBRARY

Up to this point, it has appeared that each user is assigned a portion of virtual memory that is used exclusively by their program and which no other user can access. This is not necessarily true. If the system shared library is turned on, users share portions of virtual memory. The system shared library allows all batch and interactive users to share the same pages of virtual memory and to share the code of frequently used modules. The system sets aside a sufficient number of physical pages to contain the working set of the system shared library file (SHRLIB). These reserved, shared pages are then unavailable for any other use. The shared pages contain the contents of the system shared library. The format of the system shared library file is illustrated in figure 6-7.

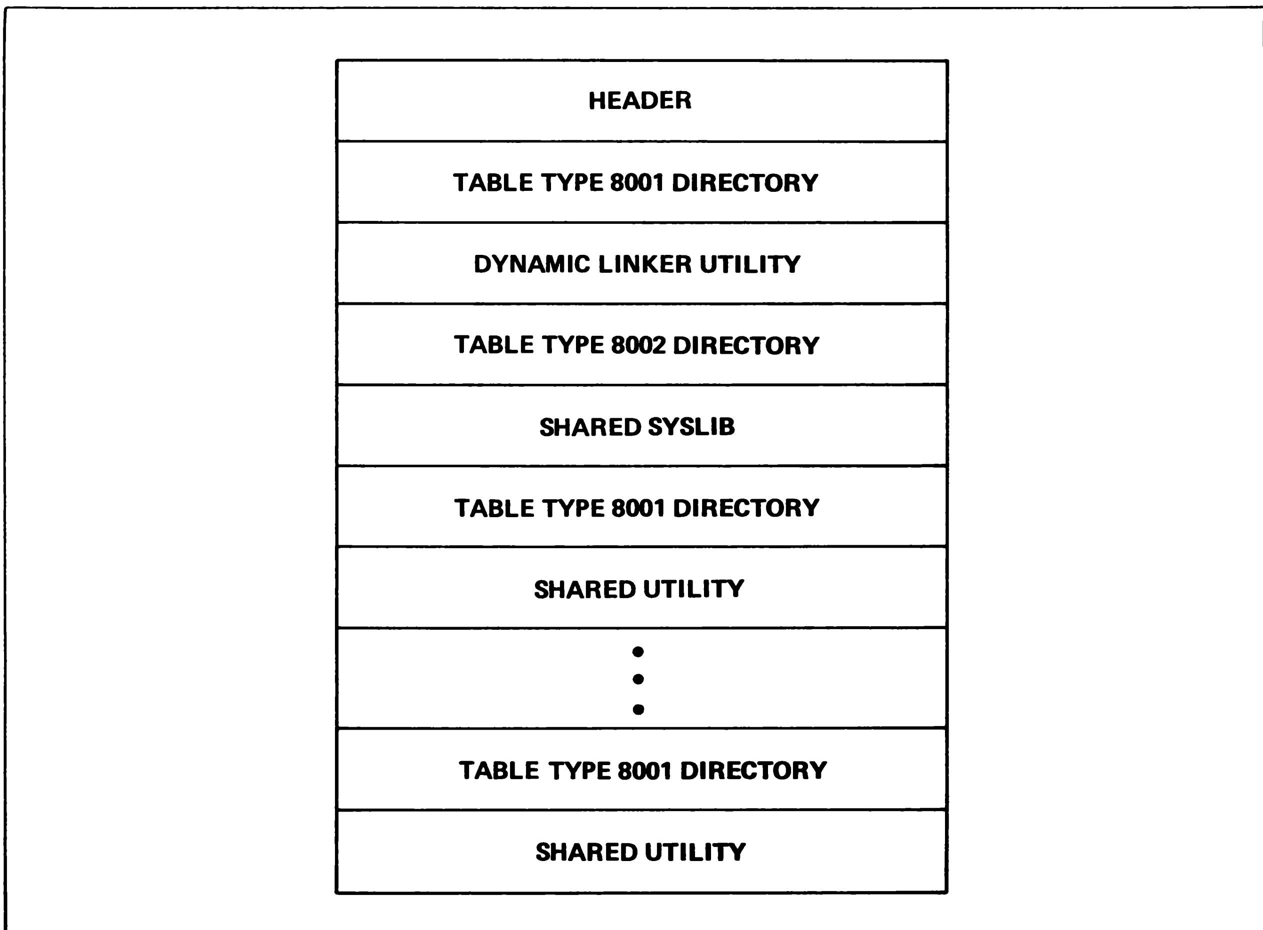


Figure 6-7. System Shared Library Format

In figure 6-7, the following items are illustrated:

- The HEADER describes the system shared library file and indicates the locations of the directories.

- A TABLE TYPE 8001 DIRECTORY describes a shared utility, including the Dynamic Linker Utility. There can be more than one directory of this type, and they can reside anywhere after the header.

- A TABLE TYPE 8002 DIRECTORY describes a shared SYSLIB that is contained in the system shared library file. There can be more than one of these directories, and they can reside anywhere after the header.

- SHARED UTILITIES are system utilities whose code resides in the shared library. The SHARED UTILITIES currently contain BATCHPRO and FTN200.

- A DYNAMIC LINKER UTILITY loads dynamic modules.

- A SHARED SYSLIB contains all the modules you normally expect to find in SYSLIB, a directory of module names, and a listing of module entry points.

CONTROLLEE FILES

Earlier in this manual a controllee file was defined as an executable file containing the information needed to execute the object code within a file.

If you are using the LOAD utility (omitting the compile and GO option) and you plan to use the system shared library and the LINKER utility, specify one of the following items:

- LOAD,LINK=D.

- LOAD,LINK=C.

Specifying LINK=D causes the LOAD utility to expect that all unsatisfied externals will be dynamically loaded and executed. If you specify LINK=D when the system shared library is turned off, LOAD builds the controllee file by mapping the existing SHRLIB into the loader's work space and using the existing SHRLIB to construct the new controllee file. Specifying LINK=C allows dynamic modules to call modules in the controllee.

You should note that a controllee built for dynamic loading and execution will not execute with the system library turned off. If a controllee is partially statically loaded with the system shared library active, and you attempt to run it without having the library turned on, your job aborts with the following error message:

```
CONTROLLEE REQUIRES SHARED LIB
```

Figure 6-8 shows how memory is allocated for controllee files during dynamic loading and execution.

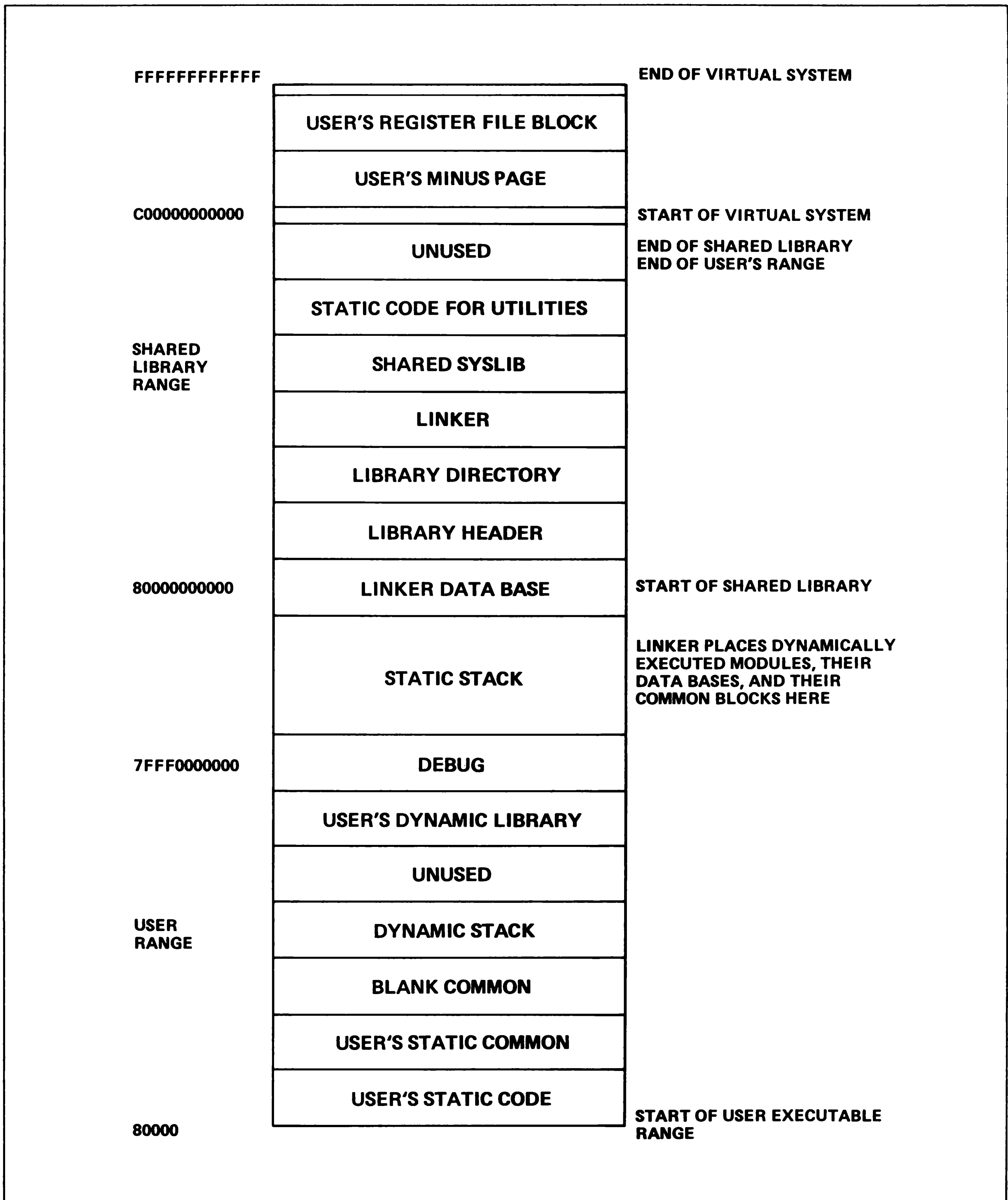


Figure 6-8. Memory Allocation for Dynamic Files

ABNORMAL TERMINATION 7

This section discusses the CYBER 200 job termination and task termination processes and how you can control them. The section also describes various tools provided with the system to help you debug your program.

CYBER 200 JOB TERMINATION

A CYBER 200 job ends with one of the following events:

- Job abort
- Abnormal termination
- Normal termination

If the task abort flag is set, the batch processor dumps task information. (Refer to the DUMP utility in the VSOS Reference Manual, Volume 1.) The batch processor then initiates abnormal job termination processing followed by normal job termination procedures.

If a task returns a termination value greater than the job threshold value, abnormal job termination processing is begun followed by normal job termination procedures.

If the end of the job file is read, only normal job termination procedures are followed.

If the job contains an EXIT control statement, job processing continues at the statement following the EXIT statement if a task has returned an abnormal termination code. If an EXIT statement is encountered during normal job advancement, job processing ends normally at the EXIT statement.

TV CONTROL STATEMENT

A task requests abnormal job termination by issuing a return code greater than the threshold value. By default, the return code 8 (FATAL ERROR) is greater than the threshold value and initiates abnormal job termination. You can lower this threshold value, however, using a TV control statement.

If you want return code 4 (NONFATAL ERROR) to initiate abnormal job termination at a certain point in your job, you can specify this by inserting a TV statement at the desired point.

If you specify TV,0+, any subsequent task that issues a return code greater than 0 (successful termination) initiates abnormal termination.

CYBER 200 TASK TERMINATION

When a task terminates, it either requests a job abort or returns a code indicating the completion status of the task. Code 0 indicates successful task completion. You can control task termination in your job through the following methods:

- User reprieve processing
- Abnormal termination control (ATC) processing
- SIL status code processing

User reprieve processing is performed when a task terminates normally or abnormally. ATC is used only when a task terminates abnormally. SIL processing allows you to choose how fatal status codes are handled.

USER REPRIEVE PROCESSING

User reprieve processing allows you to specify an external entry point to call when a program terminates. This allows your program to perform whatever cleanup processing is required at program termination whether the program terminated normally or abnormally.

User reprieve processing goes into effect when the program issues a Q5REPREV call that specifies the external entry point given control when the task terminates. This entry point must be declared external.

The user-reprieve subroutine must issue a Q5TERM call to return control to the system.

ABNORMAL TERMINATION CONTROL

Abnormal termination control (ATC) allows you to specify a subroutine that is executed only when a system error condition occurs during task execution.

ATC processing goes into effect when the program issues a Q5ENATI call that specifies the ATC subroutine. It remains in effect until the program terminates or the program issues a Q5DISATI call.

The ATC subroutine can determine whether to abort the task or to allow it to continue processing. The ATC subroutine may include a Q5RFI call to return control to the interrupted task, to abort the task, or to continue processing at a specified entry point. Refer to the VSOS Reference Manual, Volume 1, for a complete description of the ATC subroutine.

SIL STATUS CODE PROCESSING

You can specify three parameters that return information to you about the last error encountered (if any) during SIL call processing:

- 'STATUS=',stat
- 'ERRMSG=',mesg
- 'ERRLEN=',len

SIL then returns the status code of the last error encountered in the specified integer variable. (These status codes are listed in the VSOS Reference Manual, Volume 1.)

If you specify the 'ERRMSG=' parameter, SIL returns an error message of up to 80 bytes. The actual length of the message depends on what you specify on the 'ERRLEN=' parameter.

Each error message has a severity level; that is, either warning or fatal. Warning errors return control to the caller. Fatal errors also return control to the caller if the 'STATUS=' parameter is specified on the call. If the 'STATUS=' parameter is omitted from the call and a fatal error occurs, the task is terminated and a fatal error code (8) is returned to the controller.

Error message routing depends on whether the 'ERRMSG=' parameter is specified and on whether the task should be aborted as the result of the error. Possible actions are summarized as follows:

Action	'ERRMSG='Specified	'ERRMSG=' Not Specified
Task to be aborted	Message sent to the controller and to message variable	Message sent to the controller
Task not to be aborted	Message sent to message variable	Message sent to the controller

Messages that are sent to the task's controller usually appear in the dayfile for a batch job and at the terminal for an interactive job.

When you include a SIL call in your program, you must decide whether to have the program check and handle the status code returned or allow a fatal status code to request fatal termination processing for the task. The following Q5DEFINE call, for example, attempts to define a file named TEST:

```
CALL Q5DEFINE('LFN=',4HTEST,'STATUS=',ISTAT,'ERRMSG=')
IF (ISTAT .NE. 0) THEN
    IF (ISTAT .EQ. 1505) THEN
        CALL Q5ATTACH('LFN=',4HTEST)
    ELSE
        CALL Q5TERM('FATAL')
    END IF
END IF
```

If Q5DEFINE returns the status code 1505, which indicates the file already exists, the program attempts to attach the file. If Q5DEFINE returns a nonzero status code other than 1505, the program terminates and returns a fatal error (return code 8).

You can use a Q5TERM call to terminate a program at any time, not just in response to a SIL call status code. The Q5TERM call can specify 'FATAL' to issue return code 8, 'ERROR' to issue return code 4, or 'ABORT' to request a job abort.

NOTE

Ending your program with a Q5TERM call may prevent FORTRAN runtime cleanup from completing.

DATA FLAG BRANCH MANAGER

The data flag branch manager (DFBM) is the software that processes computational error conditions detected by the CYBER 200 hardware. By default, DFBM returns an error message and aborts your program if you have done the following:

- Computed an indefinite result
- Taken the square root of a negative number
- Performed floating-point division by zero

Your program can change the conditions detected by DFBM by calling the Q7DFSET subroutine. You can use this subroutine to disable detection of any of the default error conditions listed in this discussion or to enable detection of additional error conditions. You determine the error conditions disabled and/or enabled.

NOTE

The default error conditions result in fatal errors and job aborts. If enabled, additional error conditions are only warning errors unless the system error processor (SEP) is used.

For example, the following call enables detection of the exponent overflow condition and disables detection of all other conditions:

```
CALL Q7DFSET(0,'EXO')
```

The Q7DFSET subroutine can also specify a subroutine to be executed when one of the enabled conditions occurs. When a condition processing subroutine is specified, the system executes the subroutine instead of sending an error message for the condition. For example, the following call specifies the SUB1 error processing subroutine.

```
CALL Q7DFSET(SUB1)
```

The subroutine cannot have any arguments. The subroutine must communicate with other routines through common blocks.

You can disable detection of any or all of the error conditions listed in this section. To disable detection of all error conditions whose detection can be disabled, include the following call in your program:

```
CALL Q7DFSET(0,'NUL')
```

For a complete description of the data flag branch manager, refer to the FORTRAN 200 Reference Manual.

System Error Processor

The DFBM error messages in table B-2 of the FORTRAN 200 Reference Manual include fatal and nonfatal execution time error messages. By calling the system error processor (SEP), you can change certain attributes of execution time errors. You can make the following changes:

- Change the error severity from nonfatal to fatal
- Change the contents of an error message
- Suppress printing of an error message
- Change the number of nonfatal errors that can occur during program execution

The following examples illustrate some possible uses of SEP:

- `CALL SEP(022,'F')`

  This call changes the severity of error number 022 from W (warning) to F (fatal).

- `CALL SEP(022,0,0,0,0,53,'FP DIVIDE BY ZERO, EXPONENT OVERFLOW, OR MACHINE ZERO')`

  This call changes the contents of error message 022.

- `CALL SEP(022,0,0,0,'S')`

  This call suppresses printing of the 022 error message.

- `CALL SEP(0,0,0,-1)`

  This call changes the nonfatal error limit to infinite so that nonfatal errors cannot cause program termination.

For a complete description of the SEP call format and an explanation of the parameters used in the sample calls, refer to the FORTRAN 200 Reference Manual.

DEBUGGING TOOLS

The debugging process requires that you take a close look at the part of the program you believe is causing the problem. The CYBER 200 tools that help you take a close look include the MDUMP subroutine and the DUMP, LOOK, and DEBUG utilities.

MDUMP SUBROUTINE

The MDUMP subroutine prints the contents of the virtual memory area (usually an array) specified on the call. Use of the MDUMP subroutine is described in the FORTRAN 200 Reference Manual.

DUMP CONTROL STATEMENT

When your job aborts, you receive a dump of information from the drop file of the task that requested the abort. The DUMP utility lists task information saved in the drop file. The utility can do this only if the drop file is saved during task termination. A task can save the drop file by calling either the CHKPNT or Q5TERM subroutine.

For example, you can insert several CHKPNT calls in your program to copy the current state of the drop file to file CKPFILE, ending your control statement sequence with the following statements:

```
EXIT.
DEFINE,CKPFILE.
DUMP,CKPFILE.
```

If the task then terminates abnormally, you can use file CKPFILE to restart the task at the point in processing where the last checkpoint call was processed. The DUMP statement produces a dump of information showing the state of your task at the last checkpoint executed.

As a second example, your program could terminate when it issues the following SIL call:

```
CALL Q5TERM('FATAL','RESTART')
```

The drop file is available after program termination. Therefore, the job could execute a DUMP statement to print information from the drop file. For example, if the controllee file is named GO and the drop file is named 2GO, the DUMP statement for a batch job may be as follows:

```
DUMP,2GO.
```

The information provided in the dump is listed in the DUMP control statement description in the VSOS Reference Manual, Volume 1, which also describes the CHKPNT and Q5TERM calls.

LOOK UTILITY

The LOOK utility can print all or part of the contents of a mass storage file. The utility can also search for occurrences of a hexadecimal or character value, and it can replace the contents of locations within a file.

As shown in figures 3-6 and 3-7, the LOOK utility is called by the LOOK control statement. The LOOK utility reads directives to determine processing. The directive sequence ends with an END directive.

The LOOK utility is described in the VSOS Reference Manual, Volume 1.

DEBUG UTILITY

You can use the DEBUG utility to suspend execution of a controllee file at specified breakpoints. The DEBUG utility can also display and change code and the contents of arrays and variables.

The DEBUG control statement must specify a controllee file. It should also specify an output file for DEBUG to avoid file name conflicts with the runtime OUTPUT. For example, the following statement debugs file GO:

```
DEBUG,GO,O=DOUT.
```

The DEBUG utility reads directives that indicate processing. In general, these directives take the following actions:

- Specify breakpoints (BKPT and MBKPT directives)
- Initiate execution (EXECUTE)
- Display the contents of variables or arrays when execution is suspended at a breakpoint
- Alter the contents of variables or arrays, if appropriate
- Continue execution to the next breakpoint and repeat the display and alteration process (CONTINUE)
- End DEBUG processing (END)

Figure 7-1 shows a FORTRAN program and a DEBUG session that displays values as the program is executed. The DEBUG session first sets three breakpoints at statement labels 10, 20, and 40. The session then requests a status listing. The status listing shows the virtual addresses of the breakpoints and their relative addresses within DBEXMP.

The next command begins program execution. If you wish to change the file connection identifier list specified on the PROGRAM statement, enter the new connection specifier list in response to the following prompt:

```
PLEASE ENTER PRECONNECTION IDENTIFIER LIST:
```

If you are not going to change the connection identifier list, press the carriage return.

In figure 7-1, when DEBUG reaches the first breakpoint, the user displays variable J. Note that J is still 0. This is because the statement has not been executed. When a program reaches a breakpoint, the statement or instruction at that point has not yet been executed. When J is displayed at the next breakpoint, J has been set to 5.

At the final breakpoint, the user displays the five words in array IARAY. Only the fifth word has been assigned the value 2. Looking back at the program, the user sees that to assign 2 to all five words of the array, variable I rather than J should be the IARAY subscript in the DO loop.

Program Compiled and Loaded on File GO

```
   PROGRAM DBEXMP
   COMMON /A/ IARAY(5)
10 J=5
20 K=2
   DO 30 I=1,J
30 IARAY(J) = K
40 CONTINUE
   STOP
   END
```

DEBUG Session

```
attach,go
 ALL DONE
debug,go
?bkpt,10                                   <-- Sets breakpoint.
?bkpt,20                                   <-- Sets breakpoint.
?bkpt,40                                   <-- Sets breakpoint.
?stat                                      <-- Produces listing of
                                               breakpoints; last
DEBUG STATUS.                                  DEBUG and BKPT direc-
 BKPTS SET=                                    tives issued; last
000000080320               (DBEXMP=2A0)        routine or program
000000080340               (DBEXMP=2C0)        referenced; last
000000080460               (DBEXMP=3E0)        command, type and
 NO MBKPT SET.                                 module referenced;
LAST ROUTINE REFERENCED= DBEXMP                and; next execution
LAST COMMAND ISSUED=     BKPT                  address in user program.

CURRENT DEFAULT LOCATION TYPE =                    S
LAST DISPLAYED ADDRESS= 000000000000                  (NOT IN CODE)
USER NEXT EXECUTE ADDR= 000000080080                  (DBEXMP=0)
?execute                                   <-- Begins program
                                               execution.
PLEASE ENTER PRECONNECTION IDENTIFIER LIST:
BKPT ADDRESS  - 000000080320                 (DBEXMP=2A0)
?display,j                                 <-- Displays contents
7FFF80008A00    00000000 00000000              of variable J.
?continue                                  <-- Continues to next
                                               breakpoint.
BKPT ADDRESS  - 000000080340             (DBEXMP=2C0)
?display,j
7FFF80008A00    00000000 00000005
?continue
BKPT ADDRESS  - 000000080460             (DBEXMP=3E0)
?display,iaray,5
000000260000    00000000 00000000
000000260040    00000000 00000000
000000260080    00000000 00000000
0000002600C0    00000000 00000000
000000260100    00000000 00000002
?end                                       <-- Ends DEBUG session.
STOP
   ALL DONE
```

Figure 7-1. DEBUG Session Example

PROGRAM OPTIMIZATION 8

In general, program optimization is the process of improving efficiency to reduce the resources required to execute your program. Optimization can be measured in the following two ways:

- Lower memory requirements
- Shorter execution time

CYBER memory management was described in section 6. Section 8 concentrates on shortening execution time for your program through the following methods:

- Scalar optimization
- Vector optimization
- Efficient subroutine calls

The CPU on the CYBER 205 has two parts: the scalar processor and the vector processor. The scalar processor executes scalar instructions. A scalar instruction operates on a value, or a pair of values, held in registers. For each scalar instruction processed, the value (or values) must be moved from memory into a register. To perform a scalar operation, the following three steps must take place in sequence:

1. Load two operands from memory into registers.
2. Perform the operation.
3. Store the result of the operation in memory.

The second part of the CYBER 205 CPU executes vector instructions. Vector instructions operate directly on memory. By sending the operands through pipes, the vector processor can perform all of the three previous steps at one time. Once a vector instruction has started, the vector processor simultaneously can load two operands from memory, compute the result, and store an earlier result back into memory.

You can increase the efficiency of your programs through proper use of the scalar and vector processors. In general, shorter execution times result when a program uses the CYBER 205 vector processor. This section concentrates on how you can use these processors to optimize your code. Details of vector programming are described in the FORTRAN 200 Reference Manual and in the CYBER 200 FORTRAN Language Reference Manual.

SCALAR PROCESSOR

The CYBER 205 scalar processor can perform any of the functions that you normally expect from a computer:

- Issue instructions
- Perform integer and floating-point arithmetic
- Perform logical operations
- Branch from one address to another

The scalar processor consists of a central, organizational section surrounded by several functional units. The central section controls the locating, decoding, and issuing of instructions. The functional units are responsible for the execution of instructions. Figure 8-1 shows a diagram of the scalar processor.

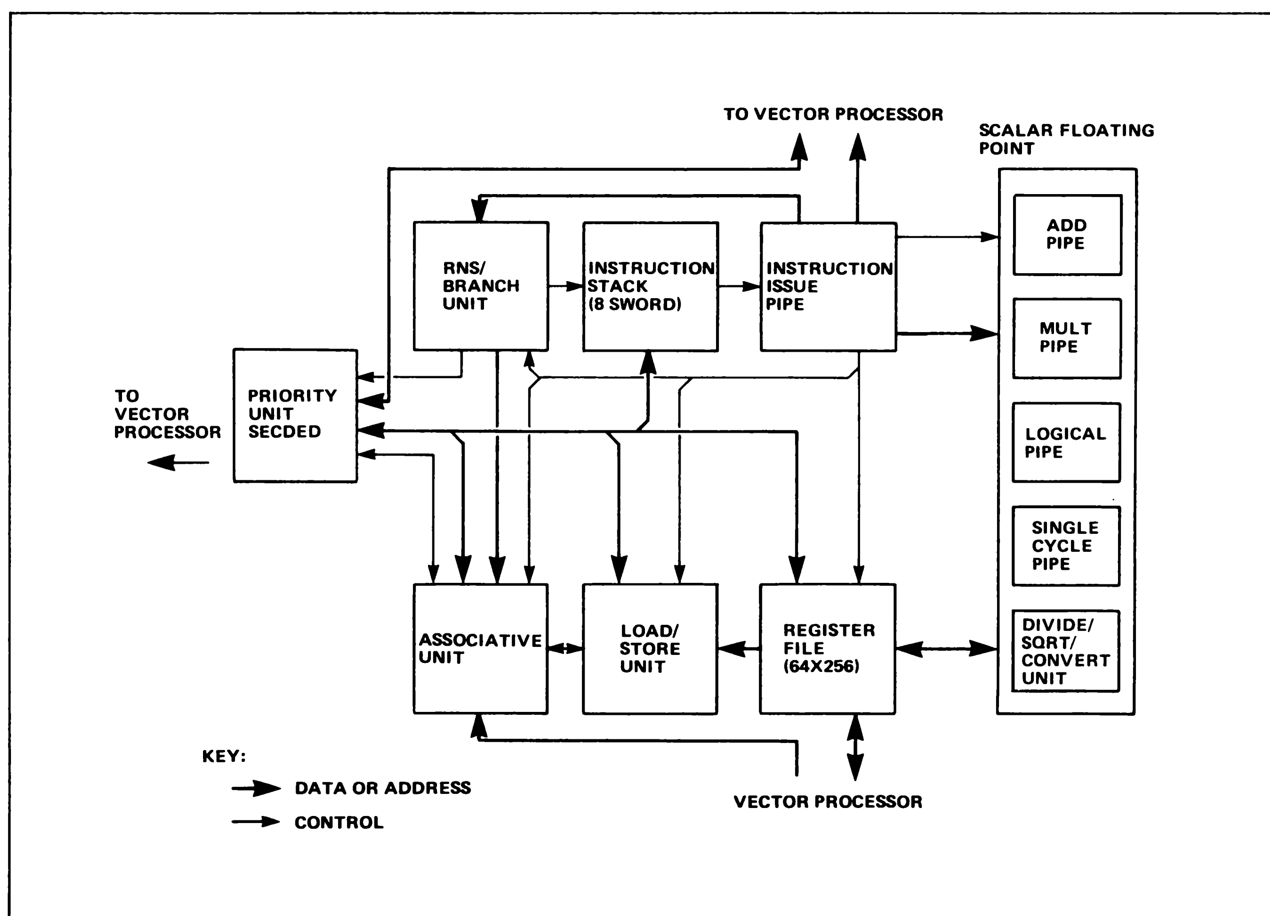


Figure 8-1. Scalar Processor Diagram

Within the central section, there are two units whose workings can affect the performance of your code. These units are as follows:

- The register file
- The instruction stack

REGISTER FILE

The register file is a set of 256 64-bit working registers that serve as work space for the scalar processor when it executes instructions. These registers are used as follows:

- Instruction and operand addressing
- Indexing
- Field length counts
- Source or destination points for register-type instructions (except LOAD and STORE)

By convention, 15 registers are reserved for special purposes and 17 serve as temporary registers. (A temporary register is one whose contents are not expected to be preserved across subroutine calls.) When a particular subroutine is entered, an image of all registers used by the caller is saved in memory. This frees 224 registers that the subroutine can use until it is time to return to the caller program. Just before the return, the previously saved image of the registers is loaded back. The caller program, therefore, notices no change in the contents of the registers.

If a subroutine contains more scalar variables than there is room for in the register file, some variables are assigned memory locations rather than register file slots.

NOTE

When a frequently used variable, such as a loop index, is located in memory, execution time will likely be longer than if the variable has a slot in the register file.

If you suspect that one of your subroutines may include more variables than can be placed in register files, use the M option on the FORTRAN control card to produce a register file map. When your program compiles, a map is produced listing which variables in a subroutine are assigned a register and which are assigned to memory.

The register file map is useful in determining which subroutines make efficient use of the register files and which routines would perform better if they were split into smaller routines.

INSTRUCTION STACK

Prior to execution, the set of instructions that comprises the machine code representation of the program is stored in the code section of the controllee file. The central, organizational section of the scalar processor is responsible for decoding and issuing these instructions. However, the scalar processor can only decode instructions residing in the instruction stack.

The size of the instruction stack is eight swords. (Sword is a contraction of super word, which is a term describing eight consecutive words in memory.) Instructions to be decoded are not loaded from memory one by one, but in units of one sword. The processor contains a look-ahead feature that tries to keep the contents of the instruction stack two full swords ahead.

Because of the look-ahead feature, processing of sequential code can proceed without delays caused by the loading of new swords. However, if a branch instruction is encountered that points to code not currently in the instruction stack, the appropriate sword must be loaded. A branch to an out-of-stack instruction takes approximately three times longer than is required to branch to code that is in-stack.

A DO loop represents a piece of sequential code that is executed from top to bottom and then is reentered at the top through a backward branch. If you want this branch to be an in-stack branch, the DO loop must not exceed six swords. The largest loop that fits in-stack is 81 half words or 40-80 instructions.

SCALAR OPTIMIZATION

Once your code has been modified to run on the CYBER 205, there are two options available to enhance program performance. These options are as follows:

- Automatic optimization
- DO loop modification

AUTOMATIC OPTIMIZATION

At compile time, you have the option of selecting any of several types of automatic optimization. This is done on the OPTIMIZE= parameter of the FTN200 statement card. The OPTIMIZE= parameter specifies to the compiler whether to optimize scalar code. The parameter also allows you to choose the type of optimization you wish to use. Two of the most frequently used choices are as follows:

- **OPTIMIZE=D**

  This tells the compiler to optimize DO loops.

- **OPTIMIZE=V**

  This tells the compiler to vectorize certain types of DO loops and transform other types into STACKLIB calls.

Refer to the FORTRAN 200 Reference Manual for a complete explanation of all the options available.

DO LOOP MODIFICATION

DO loops can be modified in several ways to speed execution time. The four major techniques you can use are as follows:

- Using recursive DO loops
- Merging short DO loops
- Unrolling DO loops
- Splitting DO loops

However, modifying a loop may adversely affect the compiler's ability to optimize a loop. It may be necessary to try different modifications and options to maximize performance.

Using Recursive DO Loops

If a pass through a DO loop uses results calculated in an earlier pass, the loop is recursive. You can significantly speed up recursive loops by keeping duplicates of some values in a temporary variable.

A typical example using a recursive loop is the following code that computes the factorial of a number:

```
F(J) = (J-1) !
```

On your first attempt, you might write the following loop if (NM1 = N-1):

```
Example 1a:      F(1) = 1.
                 F(2) = 1.
                 DO 10 J=2, NM1
              10 F(J+1) = J*F(J)
```

This code requires that the F(J) that is stored during the first pass (pass J) must be reloaded during the second pass (pass J+1).

If the result of pass J is saved in the register file, the extra load is avoided. Example 1b displays DO loop coding that eliminates the time required for the extra LOAD:

```
Example 1b:      F(1) = 1.
                 F(2) = 1.
                 FACT = 1.
                 DO 10 J=2, NM1
                 FACT = FACT*J
              10 F(J+1) = FACT
```

Merging Short DO Loops

A load-bound loop is a loop dominated by the 15 cycles needed to complete a LOAD instruction. A branch-bound loop is dominated by the 9 cycles required for a test and branch operation at the end of each pass. Short loops are almost always load bound or branch bound. One way to avoid load-bound and branch-bound loops is to merge small loops into bigger ones.

Big loops are usually busy loops in which a lot of work is performed during each pass. In general, a busy loop is not branch bound or load bound since LOAD instructions often are issued early. This permits useful work to be done while waiting for the loaded values to arrive in the register file. Even without more efficient loads, instruction scheduling is speedier when a loop is busy. The more instructions the scheduler can move, the better the results.

NOTE

If small loops can be automatically vectorized, and you plan to select the OPTIMIZE=V option at compile time, then merging into bigger loops may not save time.

The two following examples illustrate how you can merge two short DO loops into one longer DO loop:

Example 2a:

```
      DO 10 J=1, N
   10 X(J) = A(J)**2 + B(J)**2
      DO 20 J=1,N
   20 Y(J) = R(J) + S(J)
```

To improve efficiency, you can combine the DO loops as follows:

Example 2b:

```
      DO 10 J=1,N
      X(J) = A(J)**2 + B(J)**2
   10 Y(J) = R(J) + S(J)
```

Unrolling DO Loops

In addition to merging DO loops, you can also unroll a DO loop to two or more levels. This procedure reduces the time required to process a DO loop by providing the instruction scheduler with more instructions to handle.

There is one drawback to unrolling loops rather than merging them. Unrolling loops that can be automatically vectorized destroys the vector property of the loops. Merging loops that can be automatically vectorized does not destroy the vector property of the loops.

The following examples illustrate the unrolling of a short DO loop into two levels:

Example 3a:

```
      DO 10 J=1,N
   10 A(J) = X*B(J) + C(J)
```

Unroll the DO loop as follows:

Example 3b:

```
      IF (N.EQ.1) GO TO 11
      NM1 = N-1
      DO 10 J = 1,NM1,2
      JJ = J
      A(J) = X*B(J) + C(J)
   10 A(J+1) = X*B(J+1) + C(J+1)
      IF (JJ.EQ.NM1) GO TO 12
   11 A(N) = X*B(N) + C(N)
   12 CONTINUE
```

Typically, you can expect performance improvement of 30 to 40 percent when unrolling a DO loop to two levels. Unrolling a loop to three or more levels does not provide much greater efficiency than unrolling to two levels, and is usually not worth the effort required.

Splitting DO Loops

If a DO loop contains one or more loop-independent IF tests, you can speed up execution time by splitting the loop into smaller loops.

The following examples show how to split a loop into smaller loops:

Example 4a:

```
      DO 10 J=1,N
      A(J) = X(J)**2 + Y(J)**2
      IF (IFLAG.EQ.0) A(J) = A(J) + DELTA
   10 CONTINUE
```

You can split the DO loop in example 4a as follows:

Example 4b:

```
      DO 10 J=1,N
   10 A(J) = X(J)**2 + Y(J)**2
      IF (IFLAG.NE.0) GO TO 12
      DO 11 J=1,N
   11 A(J) = A(J) + DELTA
   12 CONTINUE
```

The added efficiency of loop 4b points out an important difference between the two examples. The number of branches within example 4b has less impact on performance than does the IF statement in loop 4a. The compiler can optimize the two loops in 4b more efficiently than it can handle the IF statement in 4a. An IF statement in a loop seriously affects the efficiency of instruction scheduling.

VECTOR PROCESSOR

Figure 8-2 shows a simplified view of the vector processor.

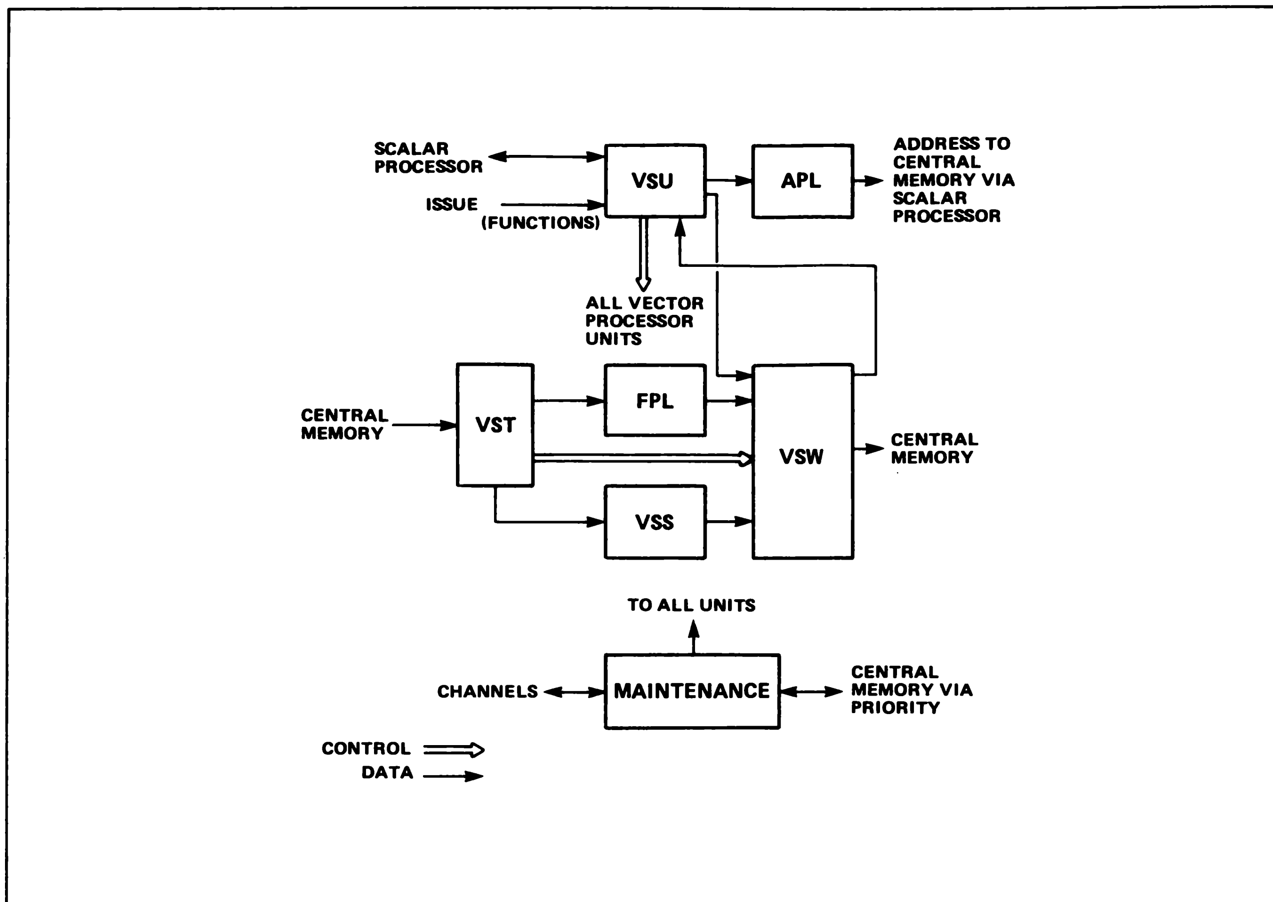


Figure 8-2. Vector Processor Diagram

The CYBER 205 has two versions. One has two identical pipes, and one has four identical pipes. In a two-pipe model, one pipe processes odd pairs of operands, and the other pipe processes even pairs. In a four-pipe model, a pipe processes every fourth pair of operands.

TWO-PIPE AND FOUR-PIPE PROCESSORS

In general, the vector processor of the CYBER 205 consists of segmented pipes that carry information. Each pipe performs one small part of an arithmetic operation.

The important concepts about pipes in the vector processor are as follows:

- Every operation is broken down into a number of steps.
- Operands must travel sequentially through the segmented pipes.

STORING ARRAYS

If you have been writing code for use on a scalar processor, you may not have paid a great deal of attention to how FORTRAN arrays are stored in memory. This is because the manner in which arrays are stored does not drastically affect execution time. On a vector processor, the way arrays are stored has a dramatic affect on execution time. On the CYBER 205, a vector may be defined as a set of contiguous memory locations. Contiguity, in this instance, is defined in terms of virtual addresses, not physical memory locations.

The following two DO loops illustrate the importance of understanding the manner in which an array is stored:

```
      DO 10 J = 1,N                  DO 20 K = 1,N
      DO 10 K = 1,N                  DO 20 J = 1,N
   10 A(J,K) = 0                  20 A(J,K) = 0
```

If a two-dimensional array is stored in columns; that is, if A(J,K) is followed immediately by A(J+1,K) in memory, then loop 10 executes more quickly than loop 20.

However, if a two-dimensional array is stored in rows; that is, if A(J,K) is followed immediately by A(J,K+1), then loop 20 executes more efficiently than loop 10.

A loop that accesses sequential storage locations in memory is always the better choice. This is particularly true for virtual memory machines like the CYBER 205. By changing from non-sequential data access to sequential, execution speed is greatly increased. You gain large increases in speed because the vector processor unit of the CYBER 205 is designed to operate on sequential memory locations.

VECTOR OPTIMIZATION

You can control how vector instructions are generated by the compiler in the following two ways:

- Automatic vectorization by specifying the OPTIMIZE=V option on the FORTRAN control card
- Explicit vectorization by using vector programming statements, vector assign statements, or vector function references

For details on how to use these two methods, refer to the Automatic Vectorization and the Explicit Vectorization headings of this section.

AUTOMATIC VECTORIZATION

You may choose the automatic vectorization option by specifying OPTIMIZE=V on the FTN200 control card. Once you select this option, the compiler attempts to transform each DO loop into a vector operation. If DO loop vectorization is not possible, the compiler tries to change the loop into a highly efficient scalar operation called a STACKLIB call.

Sometimes the compiler cannot vectorize a loop or change the loop to a STACKLIB call. In these cases, the complier listing indicates the loops it could not vectorize. (These loops are called uncollapsible loops.) The complier listing also indicates the reason the loop could not be vectorized. Examine any uncollapsible loops shown on the complier listing to determine if you can rewrite them as vector operations or as DO loops that the compiler can vectorize.

Linked Triads

The CYBER 205 can process certain forms of linked triads in vector mode as if each represents a single vector instruction. The CYBER 205 can process linked triads in vector mode if the following conditions are true:

- One or two of the input operands are vectors

- One of the two operators is a floating-point multiply, and the other is a floating-point add or subtract

Using V for vector, S for scalar, and R for result, these forms are as follows:

- VR = V1 + S1 * V2
- VR = V1 - S1 * V2
- VR = S1 + V1 * V2
- VR = S1 + S2 * V2
- VR = V1 + S1 * S2
- VR = S1 - V1 * V2
- VR = S1 - S2 * V2
- VR = V1 - S1 * S2

(The first two forms on the list are the most commonly used computations for linear algebra routines.)

Linked triad operations offer one major advantage. The processing time for linked triad operations is about the same as that of all other vector floating-point instructions (except the divide), yet twice as many arithmetic operations are performed.

However, you may discover that the compiler cannot directly vectorize some of your previously coded FORTRAN algorithms containing linked triads. This is not a big problem. Usually, you only need to reorder the sequence of arithmetic operations to obtain the needed vector structure.

Consider the problem of multiplying a rectangular matrix A with a column matrix X to obtain column matrix B; for example:

B = AX

The elements of B are formed as inner products expressed by the following formula where A has the dimensions (M, N):

$$b_j = \sum_{k=1}^{n} (a_{jk}, x_k) \quad (J = 1,2,\ldots,m)$$

You may initially decide to reproduce the mathematic formula as closely as possible, for example:

```
       DO 10 J = 1,M
       B(J) = 0.
       DO 10 K = 1,N
    10 B(J) = B(J) + A(J,K)*X(K)
```

This code executes properly but does not have good vector structure. In the innermost loop, only X is accessed sequentially; A is accessed by row and B is effectively a scalar.

To obtain good vector structure you must exchange the outer and inner loops as follows:

```
       DO 20 J = 1,M
    20 B(J) = 0.

       DO 30 K = 1,N
       DO 30 J = 1,M
    30 B(J) = B(J) + A(J,K)*X(K)
```

You now have two loops with explicit vector structure.

Factorizing DO Loops

A one-liner is a DO loop that contains exactly one FORTRAN statement (not counting the DO and the CONTINUE statements). If a DO loop can be broken down into a sequence of one-liners, the loop can be factorized.

Consider the following example:

```
       DO 10 J = 1,N
       XX(J) = X(J)**2
       YY(J) = Y(J)**2
       SN(J) = SQRT(XX(J)+YY(J))
    10 CONTINUE
```

You could factorize this example as follows:

```
       DO 11 J = 1,N
    11 XX(J) = X(J)**2

       DO 12 J = 1,N
    12 YY(J) = Y(J)**2

       DO 13 J = 1,N
       SN(J) = SQRT(XX(J)+YY(J))
    13 CONTINUE
```

You also could go one step further and completely factorize loop 13. A loop is considered completely factorized if each one-liner contains one of the following:

- Only one arithmetic (+, -, *, /, **) operator
- Only one logical (.AND., .OR., .XOR., .NOT.) operator

To completely factorize loop 13, introduce a temporary array called XY as follows:

```
      DO 14 J = 1,N
   14 XY(J) = XX(J)+YY(J)

      DO 15 J = 1,N
      SN(J) = SQRT(XX(J))
   15 CONTINUE
```

In this example, you can use SN as a temporary array. That solution, however, is not always possible. If a scalar appears on the left side of an equal sign, you may have to introduce an additional temporary array.

This factorization example shows the value of using the criteria of whether a loop can be factored as a test of whether the loop can be vectorized. One simple rule you can use to determine if your DO loops can be vectorized is as follows:

> A DO loop that can be completely factored can be vectorized if, and only if, all of the resulting one-liners can be vectorized.

Contiguity in Memory

Earlier, a vector was defined as a set of contiguous storage locations in memory. On the CYBER 205, you have two machine instructions that move data from nonsequential to sequential memory locations (GATHER) or move data back to nonsequential memory locations (SCATTER).

Suppose you have written the following code:

```
      DO 10 J = 1,N,2
   10 A(J) = B(J) + C(J)
```

If you have chosen the OPTIMIZE=V option, the compiler processes the loop in five steps as follows:

1. Gathers (B(J),J=1,N,2) into the first N/2 locations of the dynamic stack area (VB)
2. Gathers (C(J),J=1,N,2) into the next N/2 locations of the dynamic stack area (VC)
3. Performs the vector addition of VB+VC
4. Stores the result of the add in the next N/2 locations of the dynamic stack area (VA)
5. Scatters VA into (A(J),J=1,N,2)

Maximum Vector Length

If you want the compiler to optimize your code, be aware of the maximum iteration count of your DO loops. The largest vector length allowed on the CYBER 205 is 65535. If this length is exceeded, the compiler will not vectorize your code, for example:

- DO 10 J=1,60000 will vectorize (assuming other conditions are met).
- DO 20 J=1,70000 will not vectorize unless it is the innermost loop.

The innermost loop can be vectorized regardless of the iteration count. In the preceding examples, the vector lengths are specified as 60000 and 70000, but what can you do to inform the compiler of the maximum iteration count of the following:

```
DO 30 J=1,N
```

In this case, you can inform the compiler of the maximum iteration count through a DIMENSION statement as follows:

```
      DIMENSION A(50000,4),B(50000)
      DO 30 J=1,N
30    A(J,2) = B(J)**2
```

Now your loop will not miss being vectorized because of an unknown iteration count. The compiler uses the 50000 in the DIMENSION statement as the basis of the decision about whether your loop exceeds the maximum allowable length.

A problem can arise if, within a nest of loops, a control variable only indexes assumed or adjustable dimensions of dummy arrays at the same time that the loop count is unknown. In this case, the compiler has no basis for deciding whether to vectorize the outer loops. If you are sure that no loop counts exceed 65535, you can specify OPTIMIZE=V and UNSAFE=1 on the FORTRAN control card. The V (vectorize) requests vectorization, and the 1 relieves the compiler of the need to make the decision about whether to vectorize the loop. When you select these options, vector length no longer is a factor in determining if your DO loops vectorize.

EXPLICIT VECTORIZATION

There may be times when you are not satisfied with the results of an automatic vector operation or times when you wish to choose the specific DO loops to be vectorized. Regardless of why you wish to exercise more control over vector operations, the CYBER 205 FORTRAN compiler provides a method of vector control. This method is explicit vectorization.

Explicit vectorization uses special vector syntax available as an extension of the FORTRAN 200 language. With the vector syntax, you can explicitly vectorize your code or portions of your code. The two forms of vector syntax are as follows:

- Explicit syntax
- Implicit syntax

Explicit Vector Syntax

To fully define a vector, specify the following information:

- Starting address
- Data type
- Length

Earlier, a vector was defined as contiguous storage locations in memory. Since arrays are also stored contiguously in memory, the starting address of a vector is represented by an array element.

Each element in an array has a defined data type, which is declared in one of two ways. The data type can be declared implicitly or through a data type declaration statement. If declared implicitly, the array element used as a pointer automatically defines the data type of the vector elements. Allowable data types are: REAL, INTEGER, COMPLEX, DOUBLE PRECISION, and BIT. Specify length as a subscript preceded by a semicolon (;).

The following examples illustrate several types of defined vectors:

- A(1;60) A real vector consisting of (A(J),J=1,60)
- A(5;90) A real vector consisting of (A(J),J=5,94)
- KQ(1,2;100) An integer vector consisting of ((KQ(J,K),J=1,50),K=2,3)
- C(1,1;5*100) A complex vector consisting of ((C(J,K),J=1,100),K=1,5)
- R(5,5;996) A real vector consisting of (R(5,J),J=5,1000)

Once you have explicitly defined a vector, you are free to vectorize manually any loop displaying vector structure regardless of whether the system would have automatically vectored the loop.

Figure 8-3, which shows several scalar loops and their vector equivalents, illustrates simple use of vector syntax. The examples in the figure are only samples of possible explicit vector syntax. For a complete explanation, refer to the FORTRAN 200 Reference Manual.

With the explicit vector syntax shown in figure 8-3, each vector statement compiles as a single vector instruction just as if the automatic vectorizer had vectorized the corresponding scalar loop.

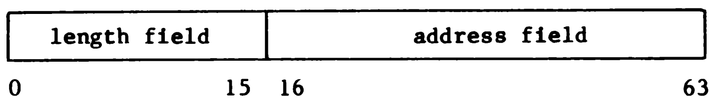
Because the semicolon notation displays all relevent characteristics of a given vector, it is often referred to as explicit vector notation. The other important type of notation, implicit or descriptor notation, is discussed next.

```
Scalar 10          DO 10 J = 1,100
                10 R(J) = S(J) + T(J+KX)

Vector 10          R(1;100) = S(1;100) + T(1+KX;100)


Scalar 20          DIMENSION A(200,10),Y(200,10),Z(200,10)
                   DO 20 K = 1,N
                   DO 20 J = 1,200
                20 A(J,K) = Y(J,K) + Z(J,K)

Vector 20          DO 20 K = 1,N
                20 A(1,K;200) = Y(1,K;200) + Z(1,K;200)


Scalar 30          COMPLEX CX(100), CY(100)
                   DO 30 J = L,M
                30 CX(J) = CY(J) * CON

Vector 30       CX(L;M-L+1) = CY(L;M-L+1) * CON
```

Figure 8-3. Explicit Vector Syntax

Implicit Vector Syntax

On the machine level, a vector is represented as a 64-bit word that contains the integer length in the 16 leftmost bits (the length field) and the starting location in the 48 rightmost bits (the address field):



This entire 64-bit word is called a descriptor. The compiler will generate code to create such descriptors for vectors, as in the following example.

The following sample code shows the assembly language (META) representation of the machine code that the compiler generates for a floating-point vector add that has been written using explicit vector references:

```
FORTRAN      A(1;N) = B(1;N) + C(1;N)

META         PACK    N,ADDA,AD
             PACK    N,ADDB,BD
             PACK    N,ADDC,CD
             ADDNV   BD,CD,AD
```

The PACK instructions create a descriptor for each of the vectors A(1;N), B(1;N), and C(1;N). The META code shows that each vector instruction creates a certain amount of overhead because of the scalar instructions involved. This overhead, however, usually is not significant to the efficient processing of your code. While one vector instruction is running, the scalar processor can still work to prepare the next scalar instruction. Most vector instructions permit overlapping scalar processing provided no LOAD/STORE memory references appear.

One other very important fact about the META code is that the value of N is not checked during execution time. PACK simply strips off the the 16 rightmost bits of N and treats them as the vector length. The largest number that can be represented by 16 bits is as follows:

```
2**16-1 = 65535
```

This explains why 65535 is the longest allowable vector length. Explicit vectorization requires you to ensure that you do not exceed the maximum vector length permitted.

DESCRIPTOR statements and ASSIGN statements allow you to create and use descriptors directly in FTN200. The DESCRIPTOR statement is a nonexecutable statement that declares a symbolic name to be a descriptor. The executable ASSIGN statement associates such a descriptor with a vector, allowing you to reference the vector using the descriptor symbolic name alone. For example:

```
DESCRIPTOR AD,BD,CD

ASSIGN  AD,A (1;N)
ASSIGN  BD,B (1;N)
ASSIGN  CD,C (1;N)
AD = BD + CD
```

The FORTRAN 200 code corresponds directly to the META code shown earlier. Each descriptor represents a vector. Therefore, each statement (except the ASSIGN) that contains one or more descriptors, is automatically a vector statement.

Descriptors must be the same data type as the vectors to which they are linked in the ASSIGN statement. A descriptor can be defined once through an ASSIGN statement, and it can be redefined many times by using additional ASSIGN statements.

There is one negative fact about using descriptors. They may make it harder for others to read and maintain your code if the defining characteristics of the vectors appear only in the ASSIGN statement and not in the vector statements. When using descriptors, you can improve the legibility of your code by using the following simple guidelines:

- Let all descriptor names end with a D. Avoid a final D on all other variables. If possible, use the name of the array to which the descriptor name points as the descriptor name, then add a final D.

- Place all ASSIGN statements just prior to the statements in which they are used. If necessary, redefine a descriptor to the same value several times. Although this creates more coding, it helps clarity.

OTHER VECTOR FUNCTIONS

In addition to the capabilities provided by explicit and implicit vector syntax, there are several other enhancements you can use to improve program performance. These are as follows:

- Vector functions

- Control vectors

- WHERE statements

- Q8 functions

The use of any or all of these capabilities can help you write more efficient code, speed execution time for your programs, and reduce your system overhead.

Vector Functions

Most of the intrinsic scalar functions available in any FORTRAN environment are also available as vector functions on the CYBER 205. The vector names for these functions are simply the scalar names prefixed by V. (Refer to the FORTRAN 200 Reference Manual for a complete list and description of the vector functions available.)

Except for DBLE, you cannot use any intrinsic function whose input argument is a DOUBLE PRECISION data type. Nor can you use a vector expression as the input argument for a vector function. However, you can use one vector (V1), two vectors (V1 and V2), or one vector and one scalar (V1 and S1) as input arguments. The result of a vector function is always a vector (VR).

Since a vector represents storage in memory, the syntax of your vector functions must always include a pointer indicating where the result (VR) of the vector function will reside in memory or an integer indicating the size of a temporary area that will receive the result. One easy way of including this pointer is making the result part of the vector function expression:

```
VFUNC (V1;VR)
VFUNC (V1;V2;VR)
```

Using this type of syntax allows you to treat a vector function expression as a true vector with its data type determined by the data type of VR. For example, if A and B are REAL arrays and X is COMPLEX, then:

```
VSIN(A(1;N);B(1;N))            is a REAL vector
VCMPLX(A(1;N),B(1;N);X(1;N))   is a COMPLEX vector
```

You can use vector functions as vectors in vector expressions. However, their most common use is in vector assignment statements where the target vector (the left-hand side) coincides with the vector function's result location, as in the following example:

```
B(1;N) = VEXP(A(1;N);B(1;N))
```

With the vector functions, you can explicitly vector almost all intrinsic function references. You are not limited to those that are accepted for automatic vectorization.

The vector length at which vector functions are more efficient than their scalar counterparts is quite small. It can be as low as a length of five. Therefore, you can use vector functions frequently in your code. This results in substantial savings of time and resources.

Control Vectors

A logical constant or variable occupies one word of storage. However, only the rightmost bit of the word is used: 1 for .TRUE. and 0 for .FALSE. This leaves 63 unused bits. More efficient use of space occurs if 64 logical values are packed into one word. This is exactly what you can do by using the data type BIT. Using data type BIT packs more than one logical value into a single word of memory.

Arrays of the BIT type are frequently used to control vector operations. They are often referred to, therefore, as control vectors rather than BIT vectors.

When you use a descriptor to represent a control vector, you must declare the descriptor as data type BIT; for example:

```
DESCRIPTOR BVD
BIT BV(100),BVD
REAL A(100),B(100)
.
.
.
ASSIGN BVD,BV(1;100)
BVD = A(1;100).GT.0
B(1;100) = Q8VCTRL(A(1;100),BVD;B(1;100))
```

In this example, the first vector statement compares each element of A with zero. If A(K) is greater than zero, the BIT element BV(K) is set to one. If A(K) is less than or equal to zero, BV(K) is cleared to zero. The final result of the compare provides a string of 100 bits whose value reflects the outcome of the 100 individual comparisons.

In the last line of this example, a Q8-function is used. (Q8-functions are described later in this section.) In this example, Q8VCTRL copies the Kth element of the source vector into the Kth location of the target vector if the Kth bit of the control vector is set to one. If the Kth bit of the control vector is zero, Q8VCTRL does nothing.

Using a control vector in the above example allows you to do the same work as shown in the following scalar loop, which could not have been vectored without the control vector:

```
      DO 10 J = 1,100
      IF (A(J).GT.0) B(J) = A(J)
   10 CONTINUE
```

You can think of BIT vectors as strings of logical variables. Therefore, like any other logical variable, BIT vectors can be used in expressions with logical operators as follows:

```
   REAL X(100),Y(100),A(100),B(100)
   BIT B1(100),B2(100),B3(100)
   .
   .
   .
   B1(1;N) = X(1;N).GT.Y(1;N)
   B2(1;N) = A(1;N).EQ.B(1;N)
   B3(1;N) = B1(1;N).OR.B2(1;N)
```

Since equal signs always connect items of the same data type, A(1;N).EQ.B(1;N) must be a BIT vector. Thus, the following expression is legal:

```
   B3(1;N) = B1(1;N).OR.A(1;N).EQ.B(1;N)
```

If you wish to clear (zero) a BIT vector, you can use the following expression:

```
   B3(1;N) = B3(1;N).XOR.B3(1;N)
```

WHERE Statements

As a standard feature, most vector instructions accept a control vector. Figuratively speaking, this control vector gives you the ability to turn a vector operation on or off. (In reality, the operation is always performed. You are simply telling the system whether to store the result or throw it away.) Use of control vectors in this manner is determined by your use of the WHERE statement. Refer to the FORTRAN 200 Reference Manual for the exact format of the WHERE statement.

The following three rules must be observed when writing WHERE statements:

- All of the vector operands in the bit expression and the vector expression that appears in the assignment statement must be the same length.

- All vectors and vector expressions in the assignment statement must be of the type INTEGER or REAL.

- A vector expression appearing in the vector assignment statement can contain only +, -, *, and / operations or a reference to the vector functions VFLOAT, VIFIX, VINT, VAINT, VSQRT, VABS, or VIABS.

Using the WHERE statement allows you to perform certain operations that in other cases would cause your program to abort. Consider the examples shown in figure 8-4.

```
Scalar 10          DO 10 J = 1,N
                   IF (Y(J).NE.0) A(J) = X(J)/Y(J)
                10 CONTINUE

Vector 10          WHERE (Y(1;N).NE.0) A(1;N)=X(1;N)/Y(1;N)

Scalar 20          DO 20 J = 1,N
                   IF (Y(J).GE.0) X(J) = SQRT(Y(J))
                20 CONTINUE

Vector 20          BIT BIT(N),BITD
                   DESCRIPTOR BITD
                   ASSIGN BITD,BIT(1;N)
                   BITD = Y(1;N).GE.0
                   WHERE (BITD) X(1;N) = VSQRT (Y(1;N);X(1;N))
```

Figure 8-4. Use of the WHERE Statement

When control vectors guide operations in this manner, dividing by zero or taking the square root of a negative number does not cause your program to abort, because the system throws the results away.

WHERE statements can also be used to define a block WHERE structure. When a block WHERE structure is executed, the bit expression is evaluated to produce a control vector. This control vector then is used for all vector assignment statements that appear between the block WHERE statement and the END WHERE statement.

The block WHERE and END WHERE statements are often used with the OTHERWISE statement in the following format:

```
WHERE

    where block

OTHERWISE

    otherwise block

END WHERE
```

Q8 Functions

Some FTN200-supplied functions are provided to give you access to specific machine instructions that perform vector operations. You can recognize these instructions by their Q8 prefix. Any function name preceded by Q8 indicates that the function is a direct machine instruction that generates in-line code instead of a subroutine call. There are two forms of Q8-functions:

- Q8S - when computing a scalar result
- Q8V - when computing a vector result

You can use both forms of Q8 functions in expressions. Their arguments usually determine their data types. Arguments for Q8 functions can be one vector, two vectors, or a control vector.

There are two important differences between the Q8V and the Q8S functions. Q8V functions follow the same syntax as the vector functions; that is, you must specify the value vector (the result field) as the last argument. The value vector is preceded by a semicolon. This is not true of Q8S functions because it would be meaningless to specify storage for a scalar instruction. The second difference is that Q8V functions must not appear in the argument list of a function reference or a subroutine call.

There are many Q8 functions available. Some perform types of data motion while others perform miscellaneous tasks. Using these functions improves your explicit vector programming and represents significant time saving over scalar code.

Refer to the FORTRAN 200 Reference Manual for a complete list and description of all the Q8 functions available.

SPEEDING UP SUBROUTINE CALLS

Most of your programs are composed of several independent program units: one main program and several subroutines and functions. You gain several advantages by using subroutines.

- Fewer lines of code are needed
- Logic is easier to follow
- Debugging is facilitated
- Code is easier to maintain

The connections between these units are handled by the following three FORTRAN interfaces:

- CALL statements
- Function references
- RETURN statements

These interfaces determine the transfer of control (jumps or branches) and the exchange of information (parameter passing).

There is one drawback to using subroutines. When you split off a section of code and make it a subroutine, your code may execute more slowly than before the split. This slower execution happens because calling a subroutine creates increased overhead. For subroutines that perform a lot of tasks, the time spent on useful work dominates. Repeated calls to a subroutine that does only a small task, however, may cause the overhead associated with those calls to show up as a significant part of your total job cost. The two major factors that create the overhead cost of subroutine calling are:

- Register file swapping
- Parameter passing

REGISTER FILE SWAPPING

Assume that you wrote a subroutine (SUB1) with the following structure, and assume that you did not specify any optimization options on the FORTRAN control card:

Beginning	Save (swap out) the caller's registers Load (swap in) SUB1's local, scalar variables Load dummy parameters
Middle	Execute FORTRAN statements
Ending	Store the dummy parameters Save (swap out) SUB1's local, scalar variables Restore (swap in) the caller's registers Return to caller program

Execution of SUB1 interrupts execution of your main program (the caller). When the branch to SUB1 occurs, the system must save the contents of the registers used by the caller in memory. A machine instruction called SWAP performs this save. The procedure of saving registers is often referred to as "swapping out."

Once the contents of the registers used by the caller program are saved, the local, scalar variables of SUB1 are "swapped in."

Fortunately, two separate SWAPS are not required. There is only one bidirectional SWAP performed. The SWAP instruction swaps in and swaps out at the same time. Therefore, only one machine instruction is necessary to save the caller's registers and to fill these registers with the local, scalar variables of SUB1. Even this single SWAP, however, causes system overhead.

For most subroutines that do a lot of work, this overhead time is usually trivial compared to the amount of time doing useful work. For these routines, you need not worry about avoiding swaps. If you have a routine that performs little work, you may wish to avoid the type of swapping described. You may wish to use the zero-swap option explained in the next paragraphs.

Swapping is the standard output of the FORTRAN 200 compiler. Under some circumstances, however, the compiler may generate a zero-swap routine that can reduce execution time for the subroutines.

Zero-swap routines use only the 17 temporary registers. Since the contents of these registers are not expected to be preserved across subroutine calls, no SWAP instruction is necessary. To qualify for zero-swapping, your subroutine must meet the following seven criteria:

- OPTIMIZE = D must be selected at compile time.
- There must be no calls or function references other than to FORTRAN routines that can be generated in-line. (Refer to the FORTRAN 200 Reference Manual.)
- There cannot be any INPUT/OUTPUT statements.
- There cannot be any explicit vector statements.
- There cannot be any vector instructions generated by automatic vectorization.
- There cannot be any special calls (for example, CALL Q8).
- The code must reasonably be expected to execute using only the 17 temporary registers.

PARAMETER PASSING

If your program uses subroutines, there are times when you need to pass information between a caller and its subroutines. The usual way of passing an array in a subroutine call is to pass the base address of the array. (The base address of an array is the address of the first element in the array.) Two methods are used to pass scalar variables. They can be passed by value or by address.

Although passing by value is simpler and requires less overhead than passing by address, value passing limits the number of scalar parameters that can be passed. Therefore, with the exception of some calls to special system library routines, scalar parameters are generally passed by address.

Another way to pass parameters is through COMMON blocks. These blocks allow quicker parameter passing because they require only a single load. For a COMMON block, the compiler generates code that effectively treats the entire block as one long array. Like any other array, one base address is sufficient to access any location in the block. This location may contain a scalar variable or an array element.

GLOSSARY A

The following terms are used in this manual. Refer to the VSOS 2 Reference Manual, Volumes 1 and 2, and the FORTRAN 200 Reference Manual for terms not included in this glossary.

Abnormal termination

The procedure the system follows when a batch task returns a completion code greater than the error threshold value for the batch job.

Access

Permitted use of a file or files. For example, a user with access to a pool can use the files that belong to the pool. A user with one or more access permissions to a file can use the file in the permitted modes.

Account identifier

One through eight characters indicating who is to be charged for system resources used by a job.

Batch input file

A mass storage file containing the control statements, programs, data, and directives that define a batch job.

Batch job

A sequence of tasks the batch processor executes for a user number.

BATCHPRO

Refer to Batch Processor.

Batch processor

A system utility that processes batch jobs. Each batch task is executed as a controllee of the batch processor.

Block

The smallest quantity of data that one device access can read or write. On CYBER 200 mass storage, a block is 512 64-bit words.

Byte

A sequence of 8 bits that is a subdivision of a word and is sufficient to represent a single character.

Checkpoint

A system feature that captures a task and any of its controllees at some point into execution such that the task can be restarted from that point. A FORTRAN program named CHKPNT calls checkpoint.

Collapsible loop

A DO loop that is convertible to a vector operation. (Contrast with Uncollapsible Loop.)

Controllee

A task started by another task (its controller).

Controllee chain

An ordered set of tasks. Except for the first and last tasks in the chain, each task is started by the task at the next lower level (its controller) and starts the task at the next higher level (its controllee). The chain can comprise up to nine tasks.

Controllee file

An executable file having a minus page as the first page and a page 0 as the second page. The file may or may not contain code. A controllee file

contains all the information needed to execute the object code contained in the file. Also called a virtual code file.

Controller

A task that starts another task (its controllee).

CPU

Central processing unit; the computational facility of the CYBER 200 system.

Data file

A nonexecutable file; also called a physical file or physical data file.

Dayfile

A file in which VSOS maintains a history of processing events. See also Job Dayfile and System Dayfile.

Directive

Supplementary control information in a file required in addition to a utility call. Directives are required, for example, with UPDATE and SLGEN.

Drop file

A file the system creates for each task to which the system maps modified pages from the task's virtual space. The system shifts the controllee file name right one character and prefixes it with a one-digit decimal prefix from 1 to 9 to form the drop file names.

Dynamic library

A library of modules that can be dynamically loaded and linked.

Dynamic linking

An execution-time process that locates an external subroutine and causes a transfer of control to the subroutine. (Contrast with Static Linking.)

Dynamic loading

An execution-time process that locates an external subroutine, initializes its common blocks and data base, and causes transfer of control to the subroutine. The subroutine code is not moved. (Contrast with Static Loading.)

Dynamic module

A module that contains unprocessed loader text. The text does not modify the code. (Contrast with Static Module Set.)

End of information (EOI)

The end of the file data. The R, W, and F record formats mark the EOF with a file delimiter.

End of file (EOF)

The point in a file after which no data exists. For labeled tape files and non-V format unlabeled tape files, the EOI is the point before the EOF1 label. For V format unlabeled tape files, two consecutive tape marks indicate the EOI.

Exponent

A number that indicates the position of a radix point in floating-point number. This indicates the power to which the number is raised.

File

A collection of data that the file name can access. Unless otherwise indicated, all references to files in this manual assume mass storage files.

File index table (FILEI)

A system table that holds all information relating to file characteristics. Output from the AUDIT or FILES utilities shows much of the table information.

FORTRAN 200

A superset of the American National Standards Institute FORTRAN language. FORTRAN 200 is the version of FORTRAN used on the CYBER 200 Series computers.

FTN200

Refer to FORTRAN 200.

Group

A set of data within a file consisting of one or more records. Groups can exist within R and W format files.

Implicit input/output

A means of accessing a mass storage file in which the system brings a page of the file into central memory in response to a reference to that page.

Input/output connector (IOC)

An entry in a minus page that links a mass storage file with a task.

Job

A sequence of tasks initiated by the batch processor executing for a user number. The control statement sequence in the batch input file and any procedure files inserted into the sequence by BEGIN statements determine the task sequence.

Job dayfile

A file the batch processor creates to record the history of a job. The job dayfile is printed at the end of the job output.

Large page

128 512-word blocks; 65536 contiguous 64-bit words. (Contrast with Small Page.)

Library

A file of modules generated by the OLE utility that the LOAD utility can use to satisfy external references during generation of a controllee file.

Local file

A private file that is destroyed by the system after termination of the batch job or terminal session that created the file.

Mantissa

The significant digits of a floating-point number.

Map

1. Process of assigning a physical address range to a virtual address range.

2. Table containing the correspondence between virtual address ranges and physical address ranges.

Mass storage

1. Generally, mass storage is disk storage.

2. Specifically, a file management category that indicates no special file processing after task termination.

Minus page

The first page of a virtual file that the system uses to hold items such as the invisible package, input/output connector information, and maps of defined virtual space. Drop files contain a second minus page.

Multifile set

A set of tape files that a single tape file request can access. The files are contiguous; each is delimited by the HDR1 and EOF1 labels. The multifile set can extend for one or more tape volumes.

Nonprivileged

A status that allows access to files owned by the same user number under which the task is running, to public files, and to authorized pool files.

Object code file

A file generated by a compiler or assembler containing relocatable code modules.

One-liner

A DO loop that contains one and only one FORTRAN statement, not counting the DO and CONTINUE statements.

Output file

1. A file destined for print or punch equipment.

2. Also, a generic term for a file being written, as opposed to an input file being read.

Ownership

A file characteristic that determines what nonprivileged tasks can access a mass storage file. Ownership categories are private, pool, and public. Private includes local and permanent files.

Page

The unit by which central memory is allocated. (See also Large Page and Small Page.)

Page fault

Reference by virtual address to a page not currently in central memory, causing a task interrupt and paging in.

Permanent file

A file that continues to exist after termination of the batch job or interactive session that creates the file.

Physical address

Actual central memory address.

Pool

A group of files accessible by more than one user, but owned by the pool, not by an individual user.

Pool file

An ownership category that indicates a file can be accessed by any privileged task and by the pool boss or any pool member.

Private file

An ownership category; a user number owns a private file. (Contrast with Public File.)

Privileged

A status that allows access to all permanent files in the system and to almost all operating system functions.

Procedure file

A file that contains a sequence of control statements headed by a PROC statement. When the batch processor processes a BEGIN statement that specifies the procedure file, the batch processor inserts the control statement sequence from the procedure file into the control statement sequence for the job.

Public file

An ownership category that indicates all users can access a file. (Contrast with Private File.)

Record

The smallest logical set of data defined within a SIL file format.

Scalar

A data item representing a single value that a scalar machine instruction processes. (Contrast with Vector.)

Scalar processor

The normal scalar processor that performs arithmetic in a scalar mode. (Contrast with Vector Processor.)

Security level

Attribute of a file, task, job, or user number used to prevent unauthorized data access. The eight security levels are numbered 1 through 8, from least to greatest security.

Segment

An area of contiguous disk space allocated to a file.

Shared SYSLIB

An OLE library altered by the SLGEN command so it can exist as a dynamic library within the shared system library.

Shared utility

A static module set that consists of loaded code that resides in the system shared library and a controllee file containing everything but the code.

SHRLIB

Refer to System Shared Library.

SIL

Refer to System Interface Language.

Small page

The smaller of the two page sizes. The small page size is chosen during VSOS autoload. The possible sizes are one, four, or sixteen 512-word blocks. (Contrast with Large Page.)

Source file

1. A generic term for a file containing text read by a utility or other task.

2. In an UPDATE utility context, a file produced by UPDATE that would allow recreation of a new program library on a subsequent creation run.

STACKLIB

A set of highly optimized scalar subroutines.

Static linking

Linking of all external references prior to execution of any of them. (Contrast with Dynamic Linking.)

Static loading

Loading of all modules at one time to resolve all external references. (Contrast with Dynamic Loading.)

Static module set

A set that consists of a statically loaded utility whose code resides in the system shared library and a controllee file containing everything but the code. The utility can be partially statically linked. (Contrast with Dynamic Module.)

Sword

Contraction of the term "super word" meaning eight consecutive words in memory.

System dayfile

A file on which VSOS records information on all tasks. A privileged user can send a message to the system dayfile.

System interface language (SIL)

A set of subroutines that user programs can call to perform system functions.

System pool

A pool of files used instead of the public files with the same names. If a system pool exists in the current system, the pool is attached to each user number when the user number submits a job or logs on.

System shared library

A file that is read or partially read during system initialization into the pages set aside for the shared library region of memory. This file contains directories, shared utilities, and a shared SYSLIB.

Task

The execution of a controllee file.

Threshold value

The maximum error code that a task can return without causing the batch processor to initiate abnormal job termination. The user sets the threshold value with the TV control statement.

Uncollapsible loop

A DO loop that cannot be converted to a vector operation or a STACKLIB call. (Contrast with Collapsible Loop.)

User number

Six digits that identify a file owner or user of system resources.

Vector

A set of data items specified as a single operand for a vector machine instruction. Execution of the vector instruction processes each data item in the set. (Contrast with Scalar.)

Vector processor

A processor that performs arithmetic as vector operations. (Contrast with Scalar Processor.)

Virtual address

Address referring to virtual memory and translated by the page table into a physical address.

Virtual code file

Refer to Controllee File.

Virtual memory

A means of addressing memory in which the system maps the addresses referenced by a task to actual physical addresses in memory.

Volume

A reel of magnetic tape.

Word

A division of central memory or mass storage corresponding to 64 bits.

Working set

The pages of a task's virtual space that are most frequently referenced during task execution.

INDEX

Abnormal termination control 7-2
Access validation 2-2
Account name 2-4
Arrays
    Columnwise order 8-9,11
    Dummy 8-13
    Rowwise order 8-9
    Storage in memory 8-9,14
    Temporary 8-12
ASSIGN statement 8-16,17
ATC
    See Abnormal termination control
    ATTACH control statement 3-10
    Attaching files to your job 3-10,12
    AUDIT control statement 3-9

Batch job
    Contents 2-5
    Definition A-1
    Example 2-9,10
    Routing 2-6
    Submitting 2-6,9,10
BATCHPRO
    See Batch processor
Batch processor 2-5; A-1
Boolean data type 4-2,3
Boolean functions 4-3

CHKPNT call 7-6; A-1
CMM
    See CYBER memory manager
Common blocks
    Aligning on a page boundary 5-8
    Named common 6-2,9,11
    Unnamed common 6-2
Compile and GO option 3-6,7
Control statement group 2-4
Control vectors 8-18,19,20
Controllee file 3-1; 6-9,11,15; A-1
Conversion from FORTRAN 5 to
  FORTRAN 200 4-1
CRM
    See CYBER record manager
CRM delimiters 3-16,17
CYBER memory manager 4-6
CYBER record manager 3-16; 4-6

CYBER 200
    Back-end system 2-1
    Login 2-11,12
    Logout 2-11

Data conversion routines 3-17,19
Data declaration keywords 3-13,16
Data dumps 3-21
Data file transfers 3-13
Data files 3-1
Data flag branch manager 7-4
Data representation
    Complex 4-12
    Double-precision 4-11
    Floating-point 4-9,11
    Integer 4-8
    Logical 4-12
DEBUG utility 7-7,8
Debugging 7-5,7
DEFINE control statement 3-8,9
Descriptor notation
    See Implicit vector notation
DESCRIPTOR statement 8-16
Descriptors 8-15,16
DFBM
    See Data flag branch manager
DIMENSION statement 8-13
Direct file access 5-5
DO loops
    Branch-bound 8-5
    Collapsible A-2
    Combining 8-5
    Factorizing 8-11,12
    Load-bound 8-5
    One-liners 8-11,12; A-4
    Recursive 8-5
    Splitting 8-7
    Uncollapsible loops 8-10; A-6
    Unrolling 8-6
    Vectorization of 8-11,12
Drop files
    Creation of 3-7
    Definition A-2
    File type in a CYBER 200 job 3-1
    Mapping of 6-9,11
    Specifying size 3-8
    Use with the DUMP statement 7-6
DUMP control statement 7-6

ENQUIRE command 2-6,7
EXIT control statement 2-8; 3-9; 7-1
Explicit vector notation 8-14,15

Factorizing DO loops
See DO loops, Factorizing
File access methods 5-5
File connections
Changing connections 3-4
Establishing connections 3-3
File ownership
Changing file ownership 3-12
Pool files 3-12; A-4
Private files 3-10; A-4
Public 3-10; A-4
File transfer between front-end and CYBER 200 3-12
File types 3-1
FILES control statement 3-9
Front-end system
Login 2-2,11
Reasons for 2-1
Tape drives on 3-24
Transferring files to and from 3-12
Use of text editor on 2-3,8
FTN200 control statement
As a part of control statement group 2-4
As a task in a CYBER 200 job 2-5; 3-6
Use of LO parameter 6-2
Use of OPTIMIZE parameter 8-10,13

GATHER instruction 8-12
GET directive 3-13
GIVE control statement 3-12

Implicit vector notation 8-15
Input files 2-5; 3-6
Input/Output
Block 5-6,7,8
Explicit 5-6
File 3-2; 5-1
Formatted I/O statements 3-20
FORTRAN runtime 5-1,2,4
Implicit 6-12
Interactive 3-20,22,23
Record 5-4
SIL 5-1,4
Instruction stack 8-4
Interactive access 2-11,12

Job categories 6-12
Job dayfile 2-8; A-3
Job file 2-2,3; 3-5
Job statement 2-4
Job termination
Abnormal 7-1,2
Abort 7-1,3,6
Normal 7-1

Large pages 6-1,2,10; A-3
LCN
See Loosely coupled network
LID
See Logical identifier
LINKER utility 3-6; 6-15
LISTAC control statement 3-11
Listing access permission sets 3-11
Load map 6-3,4,5
LOAD statement 3-6,8; 5-8; 6-12
LOAD utility 2-5; 6-3,6,15
Logical identifier 2-2,4,11,12
LOOK utility 3-19,20; 7-5,6
Loosely coupled network 3-12

Magnetic tape files
File requests 3-24
Processing options 3-25
Record type specification 3-25
Reserving tape drives 3-24
Mass storage files 3-8,9; 7-6
MDUMP subroutine 3-19,20,21; 5-5; 7-5
Memory requirements
Changing 6-12
Specifying 6-11
MFLINK control statement 3-13,14,15, 16,21

OPTIMIZE compilation option 8-4,6,9, 10,12
Output files 2-7; 3-1,6; A-4

Page fault 6-2; A-4
Page mapping
System controlled 6-9
User controlled 6-10
Paging 6-1
Password 2-2,4
PATTACH control statement 3-12
PDETACH control statement 3-12

Permanent files 3-9,10
PERMIT control statement 3-11
Physical memory 6-1
Pool files
Attaching pools 3-12
Definition A-5
Removing pools 3-12
Private files
Attaching 3-10
Creating 3-10
Defining access permission 3-11
Definition A-4
Detaching 3-10
Giving to others 3-12
Listing access permission sets 3-11
PROGRAM statement 3-3,4
Public files 3-10; A-4
PURGE control statement 3-10

Q8 functions 8-20
Q9 routines
See Data conversion routines

READ statement 3-17; 5-4,5
Record I/O 5-4,5
Record types 3-16,19
Register file map 6-2; 8-3
Remote host facility 3-12
REQUEST control statement 3-8,24
RESOURCE control statement 2-5; 3-24; 5-5; 6-11
RETURN control statement 3-8,10
RHF
See Remote host facility
ROUTE command 2-6

Scalar processor 8-2; A-4
SCATTER instruction 8-12
Semicolon notation
See Explicit vector notation
SEP
See System error processor
Sequential file access 5-5
SET control statement 6-12
SIL
See System interface language
Small pages 6-1,2,9,10; A-5
STACKLIB calls 8-10; A-5
Storage map 6-2,3,4,5
STORE instruction 8-3
SUBMIT control statement 2-6
SUBMIT error 2-6
Submitting batch jobs 2-6,9,10
Subroutine calls
Advantages of 8-21
Interfaces to 8-21
SUMMARY control statement 6-2
SWAP instruction 8-22
Sword 8-4; A-5
Syntax differences between FORTRAN 5 and FORTRAN 200 4-2
System error processor 7-4,5
System interface language
Definition A-5
SIL calls 3-19; 5-2,4,7
SIL I/O routines 5-1,4
SIL record types 3-14,15
SIL status codes 7-2,3
System shared library 6-1,14,15; A-6

Task termination
Abnormal termination control 7-2
SIL status code processing 7-2,3
User reprieve processing 7-2
Tasks 3-2,5; A-6
TDUMP subroutine 3-17
Temporary arrays 8-12
Temporary files 3-8
Converting to permanent files 3-8
Creating 3-8
Discarding 3-8
Transferring blocks 5-6
Transferring data
Binary data 3-14,18,19
Character data 3-13,14
Transferring files 3-12
Transferring records 5-4,6
TV control statement 2-8; 7-1

UNSAFE compilation option 8-13
USER control statement 2-4
User number 2-4

Vector functions 8-17,18
Vector length 8-13,16,18
Vector processor 8-8; A-6
Vectorization of DO loops
See DO loops, Vectorization of
Virtual memory 6-1, A-6

WHERE statement 8-19,20

COMMENT SHEET

MANUAL TITLE: CDC VSOS User's Guide for FORTRAN 200 Programmers

PUBLICATION NO.: 60455390 REVISION: B

NAME: ______________________________

COMPANY: ______________________________

STREET ADDRESS: ______________________________

CITY: ____________ STATE: ____________ ZIP CODE: ____________

This form is not intended to be used as an order blank. Control Data Corporation welcomes your evaluation of this manual. Please indicate any errors, suggested additions or deletions, or general comments below (please include page number references).

☐ Please Reply ☐ No Reply Necessary

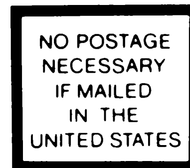
CUT ALONG LINE

REV. 5/86 PRINTED IN U.S.A.

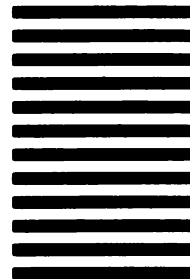
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

FOLD ON DOTTED LINES AND TAPE

FOLD FOLD



POSTAGE WILL BE PAID BY ADDRESSEE



Technology and Publications Division
ARH219
4201 North Lexington Avenue
Saint Paul, MN 55126-6198

FOLD FOLD

CUT ALONG LINE

CORPORATE HEADQUARTERS P.O. BOX 0 MINNEAPOLIS, MINNESOTA 55440