A USER'S GUIDE TO PROGRAM MANAGEMENT TOOLS

Order Number: 121958-001

Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA 95051

Additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

The information in this document is subject to change without notice.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

The following are trademarks of Intel Corporation and its affiliates and may be used to identify Intel products:

BXP	Insite	iSBC	MULTICHANNEL
CREDIT	int$_e$l	iSBX	MULTIMODULE
i	Intelevision	iSXM	Plug-A-Bubble
I²ICE	Intellec	Library Manager	PROMPT
ICE	Intellink	MCS	RMX/80
iCS	iOSP	Megachassis	RUPI
i$_m$	iPDS	MICROMAINFRAME	System 2000
iMMX	iRMX	MULTIBUS	UPI

A710/882/4K DD

REV.	REVISION HISTORY	DATE
-001	Original issue.	8/82

PREFACE

This manual provides the instructions necessary to use Intel's Program Management Tools (PMTs). These tools minimize the administrative overhead of managing a software development project. PMTs work with the existing operating systems and software tools to enable you to control, automate, and examine the evolution of software projects. PMTs are essential on large multi-programmer development efforts and extremely valuable on smaller software projects. PMTs automate the tedious administrative functions associated with software development.

This manual is intended for both systems designers and application programmers. It describes tools that aid in handling the complexities of program development and maintenance for projects ranging from a single programmer with a half-dozen modules to large, multi-programmer projects with hundreds of modules.

Chapter 1, "Getting Started with PMTs," presents an overview, describes the environment, and summarizes a software methodology that fully uses the tools.

Chapter 2, "Program Construction (MAKE)," contains the instructions necessary to invoke and execute the MAKE program that creates the generation of a new software release.

Chapter 3, "Software Version Control System (SVCS)," contains the instructions necessary to invoke and execute the SVCS program that provides control over software changes and versions.

Appendix A, "Summary of MAKE/SVCS Commands and Prompts," provides an easily accessed list of commands and prompts.

Appendix B, "Additional Information for the Series III User," provides more information about using PMTs on a Series III development system.

Related Publications

For further information on the Series III, refer to the following publications:

- *Intellec Series III Microcomputer Development System Product Overview*, 121575
- *Intellec Series III Microcomputer Development System Console Operating Instructions*, 121609
- *Intellec Series III Microcomputer Development System Programmer's Reference Manual*, 121618
- *ISIS-II User's Guide*, 9800306
- *ISIS-II Software Toolbox User's Guide*, 121727
- *Winchester Peripheral Chassis ISIS-II(W) Supplement*, 121899

For more information on the NDS-II system, refer to the following manuals:

- *NDS-II ISIS-III(N) User's Guide*, 121765
- *NDS-II Network Development System Overview*, 121761
- *NDS-II System Generation Instructions*, 121763
- *NDS-II Network Resource Manager Operating Instructions*, 121883

Notational Conventions

This manual adheres to the following conventions in describing the syntax of the commands accepted by MAKE and SVCS:

`UPPERCASE`	Characters shown in uppercase must be entered in the order shown. You may enter the characters in uppercase or lowercase.
italic	Italic indicates a meta symbol that may be replaced with an item that fulfills the rules for that symbol. The actual symbol may be any of the following:
directory-name	Is that portion of a *pathname* that acts as a file locator by identifying the device and/or directory containing the *filename.*
filename	Is a valid name for the part of a *pathname* that names a file.
pathname	Is a valid designation for a file; in its entirety, it consists of a *directory* and a *filename.*
system-id	Is a generic label placed on sample listings where an operating system-dependent name would actually be printed.
V*x.y*	Is a generic label placed on sample listings where the version number of the product that produced the listing would actually be printed.
[ ]	Brackets indicate optional arguments or parameters.
{ }	One and only one of the enclosed entries must be selected unless the field is also surrounded by brackets, in which case it is optional.
{ }...	At least one of the enclosed items must be selected unless the field is also surrounded by brackets, in which case it is optional. The items may be used in any order unless otherwise noted.
|	The vertical bar separates options within brackets [ ] or braces { }.
...	Ellipses indicate that the preceding argument or parameter may be repeated.
[,...]	The preceding item may be repeated, but each repetition must be separated by a comma.
punctuation	Punctuation other than ellipses, braces, and brackets must be entered as shown. For example, the punctuation shown in the following command must be entered: `SUBMIT PLM86(PROGA,SRC,'9 SEPT 81')`
`input lines`	In interactive examples, user input lines are printed in white on black to differentiate them from system output.
`<cr>`	Indicates a carriage return.

CONTENTS

CHAPTER 1 PAGE
GETTING STARTED WITH PMTs
What Are Intel's Program Management Tools? 1-1
PMT Components 1-1
The MAKE Program 1-1
The SVCS Program 1-1
What Environment Is Needed? 1-2
Learning to Use MAKE and SVCS 1-2
Incorporating MAKE and SVCS into an Existing Project 1-5
Starting with MAKE 1-6
Invoking MAKE 1-6
Constructing a MAKE File 1-6
Starting with SVCS 1-8
Invoking SVCS 1-8
GET Command 1-8
GET with Permission to Read 1-8
GET with Permission to Modify a Module 1-8
PUT Command 1-8
RETURN Command 1-9
SVCS Completion 1-9
SVCS Data Base Administration 1-9
ADMIN Command with CREATE Option 1-9
ADMIN Command with ADD Option 1-10
ADMIN Command with Delete Option 1-10
ADMIN Command with Add Option/ Initialized Source 1-10

CHAPTER 2
PROGRAM CONSTRUCTION (MAKE)
What Is MAKE? 2-1
MAKE Structure 2-2
Dependency File 2-2
Dependency Nodes 2-3
Task Lines 2-3
Macro Definitions 2-3
Substitution Macros 2-4
Enumeration Macros 2-4
Parameter Macros 2-4
Special Macros and Macro Constructors 2-5
Iteration Command 2-6
SVCS Access Definitions 2-7
Submit File 2-8
MAKE Invocation 2-8
MAKE Syntax 2-8
MAKE Command Options 2-9
TO *submit_file_name* 2-9
GENALL 2-9
TARGET *target name* 2-10
PRINT/NOPRINT (*list file name*) 2-10
PARAMETERS 2-10
PAGELENGTH *length* 2-11
PAGEWIDTH *width* 2-11
PAGING/NOPAGING 2-11
ATTRIB/NOATTRIB 2-11
MAKE Files 2-12
Input File (MAKE File) 2-12
Output File (Submit File) 2-12
Listing File 2-12
Header Summary 2-12
Dependency File Listing 2-12
Dependency Graph Listing 2-12
File Summary 2-12
MAKE Error Messages 2-13
Fatal Command Errors 2-14

CHAPTER 3
SOFTWARE VERSION CONTROL SYSTEM (SVCS)
What Is SVCS? 3-1
SVCS Structure 3-1
Program Units 3-2
Module Units 3-2
System Units 3-2
Unit Classes 3-2
Variations 3-3
Optimal Use of SVCS 3-4
Major Functions of SVCS 3-4
SVCS Commands 3-4
GET 3-4
WRITE [(*variant list*)] 3-5
IDENTIFIER 3-5
HISTORY 3-5
COMMON 3-5
PUT 3-6
FROM *file_name* 3-6
WRITE [(*variant list*)] 3-6
IDENTIFIER 3-6
HISTORY 3-7
RETURN 3-7
ADMIN 3-7
CREATE 3-7
ADD and DELETE 3-7
WRITEACCESS 3-8
DEFAULTACCESS 3-9
PRINT 3-9
SVCS Command Options 3-9
PROMPT/NOPROMPT 3-9
TIMEOUT/NOTIMEOUT 3-9
SVCS Invocation 3-10

CONTENTS (Cont'd.)

	PAGE
SVCS Syntax	3-10
SVCS Files	3-12
Data Base File	3-12
Auxiliary Files	3-12
Retrieved Files	3-12
History Option	3-13
Common Option	3-13
Stored Files	3-13
SVCS Error Messages	3-14
Command Errors	3-14
Fatal Object/Source Interface Errors	3-15
SVCS Prompt Messages	3-16

APPENDIX A
SUMMARY OF MAKE/SVCS COMMANDS AND PROMPTS

APPENDIX B
ADDITIONAL INFORMATION FOR THE SERIES III USER

	PAGE
Series III Literature	B-1
Hardware and Software Required	B-1
System Resources	B-1
Invocation Line	B-1

TABLES

TABLE	TITLE	PAGE
A-1	Summary of MAKE Commands	A-1
A-2	Summary of SVCS Commands	A-1
A-3	Summary of SVCS Prompt Messages	A-2

ILLUSTRATIONS

FIGURE	TITLE	PAGE
1-1	MAKE Dependency Graph	1-3
1-2	Sample MAKE File	1-4
1-3	A Sample of SVCS and MAKE	1-4
2-1	Software Development File Dependencies	2-1
3-1	Software Version Control	3-1

CHAPTER 1
GETTING STARTED WITH PMTs

What are Intel's Program Management Tools?

PMTs are a set of software tools designed to simplify and reduce the manual effort required to manage the software development process. PMTs essentially:

- Act as *programmable secretaries* and eliminate the manual administrative tasks of tracking program changes and managing module variants
- Decrease overhead associated with software management
- Control, automate, and examine the release of a product

PMT Components

The PMTs are independent programs that may be used either separately or together. Two such programs presently exist: MAKE and the Software Version Control System (SVCS).

The MAKE Program

MAKE is a tool that creates software generation procedures for constructing new releases of software. Without MAKE, new releases of a software system are created in one of two ways: One way is to perform the generation from the ground up, compiling all source modules and linking all object modules using a submit file. This method wastes considerable time on unnecessary compilations of modules that were unchanged since the last generation. The second (and more common) approach is to keep track of the modules that have been modified, and create a new and different generation procedure for each release. This method saves compilation time, but involves considerable administrative overhead, thereby compounding the chances of human error.

With the MAKE facility, you can specify how the system is constructed and automatically generate the appropriate minimal submit file. This reduces both unnecessary compiles and links, and the manual effort required to track changed modules.

MAKE provides you with the confidence that the program is constructed with the latest source and object code without unnecessary processing.

The SVCS Program

SVCS is a tool that simplifies many of the module housekeeping tasks. SVCS provides a means of tracking changes to program source code, maintaining variants of the source and object modules for a program, and recording access to these modules in a multi-programmer environment.

Because SVCS tracks source changes, you can print out the change history of a source module showing the author and time of each creation and change.

Another important use of SVCS is to track different variants of a module. These variants may represent different prototype releases or customized variants of software for different end products. SVCS tracks source modules, include files, object modules, link and load modules, and documentation (specifications, design documents, user manuals).

Incorporating MAKE and SVCS into existing software methodology is straightforward. These programs are designed to work with the existing operating system and software utilities (editors, compilers, utilities), and require about the same learning effort as the system functions (COPY, RENAME, ACCESS).

What Environment is Needed?

PMTs require an 8086/8088-based development system with 96K of user memory (in addition to space used by the operating system).

PMTs are ideal in a networked environment, such as NDS-II (using ISIS-III(N)) where multi-programmer software control is essential; however, they are just as useful on standalone winchester-based systems (using ISIS-II(W)).

Learning to Use MAKE and SVCS

This section shows you a very basic application of MAKE and SVCS. It has been designed to familiarize you with the most elementary commands and methodology. It further illustrates how easily MAKE and SVCS can be incorporated into an existing software project.

To develop a basic learning environment, we describe a software development project where two programmers are working on one part (e.g., an I/O subsystem) of a larger multi-programmer project.

Some of the common administrative headaches we can eliminate for you by using PMTs are

- Source contention—both programmers may attempt to make changes to a single source module simultaneously.
- Variations—several versions of a given module may be active in different prototypes, during various phases of debugging. A stable version may be used when debugging the whole system; a less stable but more functional version may be debugged independently.
- Generation—the latest version of some modules may need to go into each new generation. Past versions of other modules may be needed also.

MAKE and SVCS provide solutions to these development problems. MAKE automatically handles the software generation process; SVCS provides control to the edit/translate/debug process. SVCS is also responsible for the control of the variations of each module and for the control of the various pieces of the system.

Typically, these tools can be used in the development cycle in the following way:

1. A project data base is set up, using SVCS. SVCS provides administrative functions to install source and object modules, define variants to modules, and define the composition of modules.
2. A MAKE file is created to describe how the different pieces of the software system fit together.
3. Once the data base and MAKE file are set up, administrative changes are usually minimal. Day-to-day use of SVCS will be to check out and return modules, just as you would a library book.

   To make changes to a module, it is checked out with the SVCS GET command with WRITE privilege. The file is then edited, using ALTER, CREDIT, or another text editor.

4. After one or more modules are checked out and edited, a new version or prototype is generated. This is done with one MAKE and (usually) one submit command.
5. The new source and objects may be returned to the data base, using an SVCS PUT command. (This process does not have to occur at this point since additional changes and generations can take place.)
6. The new version is debugged and tested, using PSCOPE, ICE, or another debugger. It is probable that several edit/debug cycles will take place before returning the changed modules to the SVCS data base.
7. When the modules are returned, SVCS automatically updates the change history of the modules, recording who made the changes, what changed, when the changes were made, and why.
8. The SVCS GET, edit, MAKE, debug, SVCS PUT cycle is repeated, with GETs and PUTs being performed for each change, or perhaps just once a day. This cycle depends on how many different programmers want to make modifications to a given source module.

For simplicity, assume that the subsystem under development (IO.LNK) contains four modules (READ.SRC, WRITE.SRC, DATA.SRC, UTIL.SRC) that are compiled and linked to form IO.LNK.

The subsystem module IO.LNK is dependent on (constructed from) four source files (READ.SRC, WRITE.SRC, DATA.SRC, UTIL.SRC) that are in turn dependent on their corresponding object files (READ.OBJ, WRITE.OBJ, DATA.OBJ, UTIL.OBJ). This dependency is represented in figure 1-1 and specified by the user in a MAKE file (e.g., IO.MKE). It is essential that the MAKE file be accurate and complete to perform the correct generation. Figure 1-2 is an example of a MAKE file for IO.LNK. Figure 1-3 shows some source SVCS and MAKE invocations corresponding to the usage described previously.

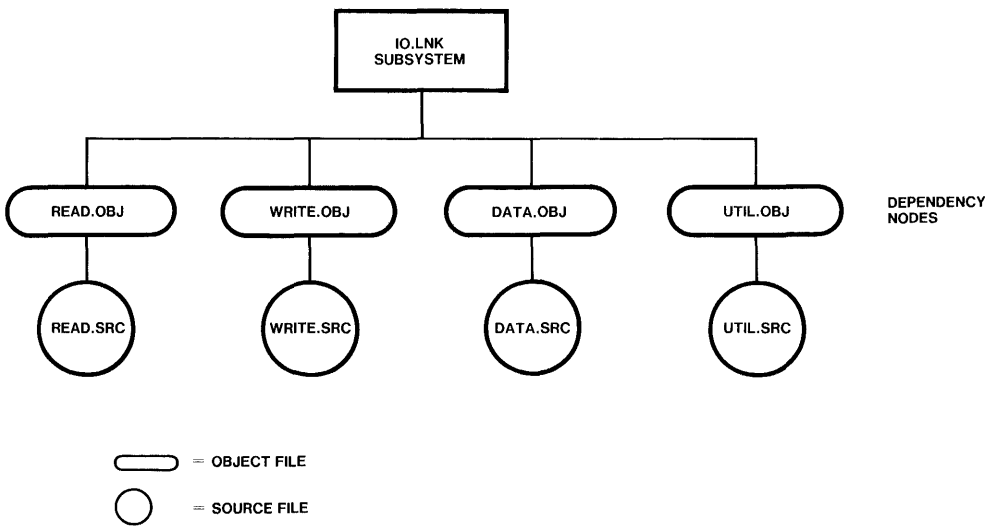

Figure 1-1. MAKE Dependency Graph 121958-1

```
SERIES-III MAKE, V1.0  07/30/82

MAKE INVOKED BY: :F1:MAKE.86 :F1:IO.MKE PRINT(:F1:IO.LST)
SUBMIT FILE: :F1:IO.CSD

     1
     2
     3  $IF IO.LNK > READ.OBJ,WRITE.OBJ,DATA.OBJ,UTIL.OBJ THEN
     4      LINK86 READ.OBJ,WRITE.OBJ,DATA.OBJ,UTIL.OBJ TO IO.LNK
     5  $END
     6
     7  $IF READ.OBJ > READ.SRC THEN
     8      PLM86 READ.SRC
     9  $END
    10
    11  $IF WRITE.OBJ > WRITE.SRC THEN
    12      PLM86 WRITE.SRC
    13  $END
    14
    15  $IF DATA.OBJ > DATA.SRC THEN
    16      PLM86 DATA.SRC
    17  $END
    18
    19  $IF UTIL.OBJ > UTIL.SRC THEN
    20      PLM86 UTIL.SRC
    21  $END
    22


    DEPENDENCY TREE

  1  IO.LNK
  2      READ.OBJ
  3          READ.SRC
  2      WRITE.OBJ
  3          WRITE.SRC
  2      DATA.OBJ
  3          DATA.SRC
  2      UTIL.OBJ
  3          UTIL.SRC


MAKE FILE SUMMARY
    NUMBER OF LINES   =    22
    NUMBER OF ERRORS  =     0
```

Figure 1-2. Sample MAKE File

```
RUN SVCS GET :F1:IO.DB(READ) TO :F5:READ.SRC WRITE ID(JNS)
; a "get" with write permission

RUN ALTER :F5:READ.SRC
; make changes to source file

RUN SVCS PUT :F1:IO.DB(READ) FROM :F5:READ.SRC ID(JNS)
; put the changed module back

RUN MAKE :F1:IO.DB.MKE
; this creates the generation procedure

RUN SUBMIT :F1:IO.DB.CSD
; submit the generation procedure created by MAKE

RUN PSCOPE
; debug the load module
```

Figure 1-3. A Sample of SVCS and MAKE

Incorporating MAKE and SVCS into an Existing Project

Changing development methodologies during a project has some inherent risks. Most project leaders will choose not to abandon current conventions when a release date is near. However, administrative headaches compound as the release data approaches; MAKE and SVCS provide the relief needed at that stage of a project.

Some guidelines for including MAKE and SVCS into an ongoing project are listed in this section. These guidelines should help you measure both the effort required and the associated risk.

1. Converting submit files to MAKE files. The most common form of software generation involves a prolific use of submit files. To alleviate the problem of unnecessary compiles and links, submit files are usually edited for every new generation. This is the effort that MAKE automates.

   To adopt MAKE in the generation process, the complete set of submit files should be converted to input files for MAKE. This means learning the command language (Chapter 2) and converting the procedures. If the submit files were just a series of compiles and links, learning the command language and converting the procedures are very straightforward (use figure 1-2 as a reference). If the submit files contain ISIS commands (COPY, DELETE, etc.) or ISIS Toolbox commands (IF, THEN, GOTO, etc.), more thought may be required. (A MAKE file can exploit some powerful macros; they just have to be learned.)

   The important point, however, is that once this MAKE file is created, you are finished with it. A MAKE file has to change only if the structure of the whole system changes. Test the MAKE file (with the GENALL option) to see if it is generating the correct submit file.

2. Setting up the SVCS data base. Installing source, object, include, link, and load modules into an SVCS data base is repetitive work. It is quite straightforward using the SVCS ADMIN commands (Chapter 3), but for a large system the process may be lengthy. You should automate it by using a submit file. It should take only a few hours to learn the syntax and create all the SVCS ADMIN commands. The actual installation, if automated, will take about twice as long as it would to copy each source and object from one file to another. (All SVCS commands may be thought of as intelligent copy commands.)

3. Day-to-day use. Once the overhead of setting up the SVCS data base and creating the MAKE file is done, the PMTs begin lowering the administrative burden.

   For software generation, the MAKE, SUBMIT sequence should replace the editing of submit files, the tracking down of latest versions, and the effort to determine what modules have changed.

   For making program changes, the SVCS GET, ALTER, SVCS PUT sequence will replace the uncontrolled editing, copying, archiving, and disk-labeling of software modules. This sequence will also reduce the problems caused by lack of control, such as deleting the wrong versions, debugging the wrong versions, and making simultaneous changes to modules.

   Learning to use MAKE (day-to-day) is like using SUBMIT—the command is very short, the options few. Learning how to use SVCS in a controlled manner is slightly more complicated (due to its flexibility) and represents the only ongoing investment in getting a large project converted to a new approach. To reduce the confrontation of SVCS, SVCS prompts the user for any required command option left off, rather than reporting an error. Learning SVCS is no more difficult than learning the collection of utilities it replaces: COPY, ATTRIB, DELETE, etc.

The next sections provide a tutorial on getting started with MAKE and SVCS; they cover the important and most frequently used commands and options. Chapters 2 and 3 serve as reference material for the whole command set.

Starting with MAKE

MAKE takes an input file and generates an output (submit) file and a listing.

This section illustrates how you would construct a MAKE input file and invoke the MAKE command.

Invoking MAKE

To activate MAKE processing, type:

```
MAKE make_file <cr>
```

The following message is then displayed:

system-id MAKE, V*x.y*

Constructing a MAKE File

The file to be constructed (usually a load module) is referred to as the target. Its constituent files are known as dependency files. A MAKE file consists of a series of dependency nodes.

A dollar sign must appear as the first non-blank (or tab) character on a MAKE command line.

The format of each dependency node in a MAKE file is as follows:

```
$IF target_file : dependency_files THEN
        task lines
            .
            .
            .
$END
```

where

target_file	is the name of the file that is constructed by the task lines.
dependency_files	are object or source files that are used to construct the target (e.g., via a compile or link).
task_lines	designate what processing needs to be done to bring target file up-to-date.

MAKE then looks at the characteristics (such as last-modify time and date) of the named files and constructs a submit file with those user-specified task lines that are needed to generate the up-to-date version of the target file.

Example 1 shows a complete MAKE file that states the dependency (shown in figure 1-1) as well as the tasks to be executed if any of the dependencies are not fulfilled. If the files in the dependency list are dependent on other files, those dependencies are also declared.

Example 1:

```
$IF IO.LNK > READ.OBJ,WRITE.OBJ,DATA.OBJ,UTIL.OBJ THEN
 RUN LINK86 READ.OBJ,WRITE.OBJ,DATA.OBJ,UTIL.OBJ TO IO.LN'
$END

$IF READ.OBJ > READ.SRC THEN
    RUN PLM86 READ.SRC
$END

$IF WRITE.OBJ > WRITE.SRC THEN
    RUN PLM86 WRITE.SRC
$END

$IF DATA.OBJ > DATA.SRC THEN
    RUN PLM86 DATA.SRC
$END

$IF UTIL.OBJ > UTIL.SRC THEN
    RUN PLM86 UTIL.SRC
$END
```

The first node of this file checks to see if IO.LNK is older than any of the .OBJ files. If it is, MAKE places the task lines into the output file. The last four dependency nodes compare the age of each source and corresponding object module. If the object module is older than the source, MAKE also places those task lines in the output file (submit file).

The resulting submit file contains the task lines exactly as MAKE reads them. Since MAKE is dependent on the accuracy of this file, it is suggested that the PRINT option be issued in a MAKE invocation so that you can verify the accuracy of the dependency information.

After the dependency specifications are checked for accuracy, the output file is created. This file is the submit file that consists of the appropriate task lines.

If MAKE detects an error in the dependency specification, it will place a message in the listing file and report the number of errors in the sign-off message. The form of the error message is

***ERROR *nnn* IN LINE *lll*, NEAR "*ttt*" : *message*

where

nnn	provides the error number.
lll	provides the line number.
ttt	provides input text near where the error was detected.
message	provides an explanation of the error.

Chapter 2 contains more detailed information about MAKE and its more complex commands. Of key importance are the macros that simplify the MAKE file (e.g., show how the four object files may be represented by a single identifier).

Starting with SVCS

An SVCS data base contains all the modules that make up a software system. Normal operation of SVCS will be to retrieve modules (GET) and replace modules (PUT or RETURN). Some data base administrative commands (ADMIN) also handle the addition and deletion of modules to the data base.

Invoking SVCS

SVCS invocations are fairly lengthy, involving several (logical) options. Wherever possible, SVCS will prompt the user for missing options.

GET Command

The GET command is used to check out a module from the data base. The user can issue the GET command to read information from the data base (e.g., to print a listing or look at the change history) or to obtain write permission so that the module can be modified.

GET with Permission to Read

The optional command given here retrieves a module from the data base. There is no write privilege.

```
SVCS GET :f1:io.db(util) TO :f5:util.src <cr>
```

In this case, :f5:util.src is the name of the file that will receive the module copied out of the data base.

GET with Permission to Modify a Module

The optional command given here requests write permission on the unit that is to be retrieved from the data base.

```
SVCS GET :f1:io.db(data) TO :f5:data.src WRITE ID(jns) <cr>
```

If the ID portion of the command is left off, SVCS will prompt the user for an identification. The identifier is mandatory for write permission and is used by SVCS to identify the person modifying the module.

PUT Command

The PUT command enables the user to return the modified module to the data base.

```
SVCS PUT :f1:io.db(read) FROM :f5:read.src ID(jns)
HISTORY (text) <cr>
```

where

text provides commentary as to why the module was changed

The user can either supply the history information in the command line or wait for SVCS to prompt for it. The history option is required if the PUT command is for a source module.

If you attempt to issue the PUT command without having write permission, SVCS will issue an error message.

RETURN Command

The RETURN command enables the user with write permission to return a module to the data base without having modified it.

```
SVCS RETURN :f1:io.db (WRITE) ID(jns) <cr>
```

SVCS Completion

When SVCS is completed, it will sign off

```
SVCS COMPLETED
```

You have now learned the basic commands to modify the units with a data base controlled by SVCS. Chapter 3 contains more detailed information about SVCS and its options.

SVCS Data Base Administration

Since modification of the data base is controlled, it is not necessary for the average user to become familiar with the following section. Its purpose is to familiarize those particular users having data base responsibility with the administrative functions associated with a data base controlled by SVCS.

ADMIN Command with CREATE Option

The ADMIN command with the CREATE option permits an SVCS data base to be created.

```
SVCS ADMIN IO.DB CREATE <cr>
```

This instruction sets up an empty data base. SVCS issues an error message if the named data base already exists.

IMPORTANT

If you want the SVCS database to be publicly shared in an NDS-II environment, you ***MUST*** change the WORLD access rights of the database file named with the ADMIN command. Do this immediately after you create it; SVCS will automatically use the same access rights when it creates its auxiliary database files.

ADMIN Command with ADD Option

The ADMIN command with the ADD option allows the administrator of the data base to add units (modules) to the data base. These units are empty (a PUT command stores its initial contents).

```
SVCS ADMIN IO.DB ADD (UNIT = READ,WRITE,DATA,UTIL) <cr>
```

This instruction adds four units (READ, WRITE, DATA, UTIL) to the data base, IO.DB.

ADMIN Command with DELETE Option

The ADMIN command with the DELETE option allows the administrator of the data base to delete units from the data base at any time.

```
SVCS ADMIN IO.DB DELETE (UNIT = READ,WRITE,DATA) <cr>
```

This instruction deletes three units (READ, WRITE, DATA) from the data base IO.DB.

ADMIN Command with ADD Option/Initialized Source

This instruction allows the administrator of the data base to add units to the data base and initialize the source.

```
SVCS ADMIN IO.DB ADD (UNIT = read FROM :f5:read.src) <cr>
```

These commands are used to provide basic adminstrative functions. Chapter 4 gives a detailed command description for those users with data base responsibility.

CHAPTER 2
PROGRAM CONSTRUCTION (MAKE)

This chapter presents a more in-depth presentation of MAKE. It provides you with more detailed information on the structure of the MAKE file and invocation controls.

What Is MAKE?

The MAKE program is designed to generate a submit file that can be used to construct the most current version of the requested software. The construction of the most current software is based upon the dependency of one file upon another.

MAKE performs the following functions:

- Parses the user-specified dependency file
- Checks the relative ages of the specified files
- Creates a submit file containing only the user-specified tasks necessary to generate a new version of software.

MAKE begins processing the input file by checking the modification characteristics and interrelationships of the files you select. Generally, linked object files are dependent upon source files and a load module is dependent upon object modules. These dependencies can best be represented by a dependency graph (see figure 2-1). At each dependency node in the graph you can designate one or more tasks that need to be accomplished to bring the file up-to-date. These tasks are typically compiles (to bring the object file up-to-date with the source) and links (to bring the link or load module up-to-date with the object files).

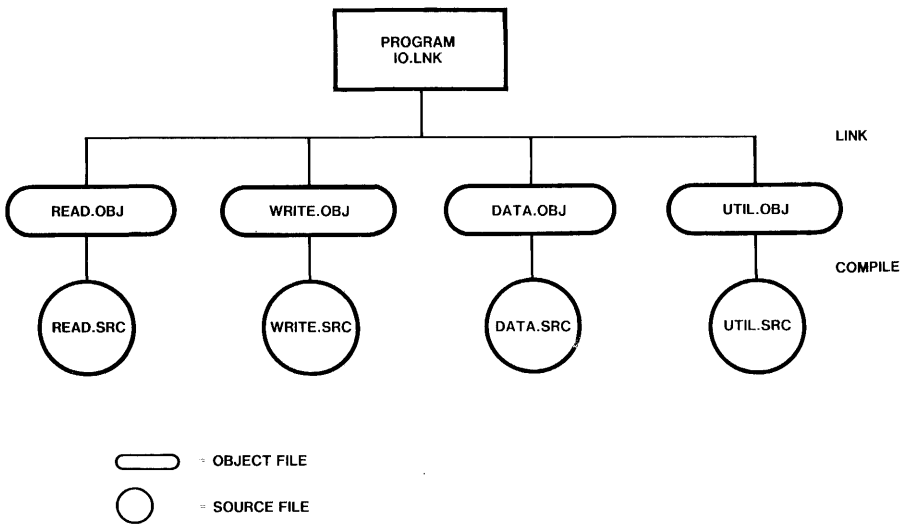

Figure 2-1. Software Development File Dependencies 121958-2

Based upon the dependency data supplied by you and modification information in the directory, MAKE automatically specifies which tasks need to be performed to generate the current or requested version of a program or program module. Thus, if an include file is used by only five of the twenty modules used to construct an entire program, modifications made to the include file would subsequently cause recompilation of only those five modules. This process avoids costly recompilation of software modules that have not been modified since the last time they were compiled and yet assures you that the software created is current.

Make Structure

MAKE translates the user specified MAKE dependency file (containing the interrelationships between files and the set of tasks to be performed if the file dependencies are not fulfilled) into the submit file that contains only those tasks required to create the latest version of a program.

Dependency File

A dependency file consists of one or more MAKE commands. These commands are either macro definitions, SVCS data base access definitions, or dependency nodes. Each of these is explained in the following sections.

Each section of the dependency file contains the specifications of a node in the dependency graph. Each node consists of the following:

- Target—indicates the files being constructed. If more than one target is specified, the oldest is used to compare modification attributes.
- Dependency—indicates those files that are used for construction of the particular target or targets.
- Task—the set of program invocations and commands that are to be performed for reconstruction of the target.

The following rules apply to the specifications in the dependency file:

1. A dollar sign ($) must appear as the first non-blank (or tab) character on a MAKE command line (as opposed to task lines).

   This character differentiates command lines (macro definitions, target/dependency specifications, SVCS access definitions, and iteration commands) from task lines (those lines that are passed through the submit file). Leading blanks and tabs may be used to format the MAKE command lines since they are ignored by MAKE.
2. A semi-colon (;) that is not enclosed by quotes (" ") can appear in a MAKE command line, causing MAKE to ignore all characters appearing after the semicolon to the end of the line. This feature can be used to include comments in the MAKE file.
3. Only task lines are placed in the submit file that MAKE generates.
4. A macro definition must appear before the macro is used.

The target, dependency, and task information is specified as a unit in such a sequence that, following the specification of the target and dependency information, is a series of zero or more task lines. Such a unit is referred to as a dependency node.

A MAKE dependency file consists of up to five types of constructs described as follows:

- Dependency nodes
- Task lines
- Macro definitions
- Iteration commands
- SVCS access definitions

Dependency Nodes

The target/dependency specification (established at each dependency node) matches up a target file (or list of target files) with a file (or list of files) on which it depends. This specification takes the form of the keyword IF, followed by the target files, followed by the special character > (meaning is older than), followed by the dependency files, and then the keyword THEN. This specification is followed by zero or more task lines and finally, the keyword END. For example,

```
$IF READ.OBJ > READ.SRC THEN <cr>
```

task lines

```
$ END
```

If any dependency file within a dependency list has been modified more recently than the target file (or the oldest of the target files at this node), the tasks are required for updating the target and are placed in the submit file. Other circumstances that also cause the tasks to be placed in the submit file are as follows:

- If the target file does not exist, it is not an error. The intent of the tasks would most likely be to construct the target regardless. This condition would cause the task lines to be added to the submit file.
- If one or more dependency files do not exist, one of the following outcomes occur. If the files are targets in other dependency nodes, no error is generated, because those files would be created at run time by the other dependency nodes. However, if the files are not targets in other dependency nodes, an error is generated.

Task Lines

Task lines begin with any character other than a $ as the first non-blank (or tab) character on the line.

These lines are not inspected except to perform macro substitution; they are tokenized so that the macros can be found and replaced. The lines are, however, reconstituted if they are written to the submit file. Since these lines might then appear different than those in the task lines, you may wish to indicate that the lines are to remain exactly as they are (including spacing). This can be accomplished by placing quotes (either single or double) around the line. MAKE would then strip away the quotes and simply place the line in the submit file with no further modification and no macro substitution.

Macro Definitions

A macro is a command that represents a string of frequently used instructions. Macro definitions are MAKE command lines that define macros (string substitutions) to be used throughout the remainder of the file. You define macros to add readability to the MAKE file and/or to simplify the specification and modification of this file.

The syntax of a macro definition is as follows:

```
$SET macro_name TO macro_specification
```

where

macro_name	is the name of the macro.
macro_specification	is either a substitution macro or an enumeration macro, as described later.

A macro is used by naming it. The name must be preceded by the macro character %. In order to distinguish a macro from the surrounding context, the macro name can be delimited by a matching pair of either single or double quotation marks. For example,

```
%"SOURCEDEVICE"FILE.SRC
```

references the macro named SOURCEDEVICE.

Substitution Macros. A substitution macro defines a string of text that is to replace the macro reference whenever it occurs in either a MAKE command line or task line. A string consists of all characters contained between a set of matching double or single quotes.

A string is not inspected until it becomes the alternate input at the time of substitution (is invoked). Macro references can be nested in this way to a depth of 16 levels.

Example

```
$SET SOURCEDEVICE TO :F1:
```

or

```
$SET OPTIONS TO 'COMPACT OPTIMIZE(1)' DEBUG XREF
```

Enumeration Macros. An enumeration macro is used to specify a list of substitution strings (such as filenames or fragments of filenames) that are to be referenced together or iterated upon. The enumeration macro is extremely helpful for handling groups of files that are treated in the same manner. The syntax of the enumeration macro is as follows:

```
$SET identifier TO (enumeration list)
```

Example

```
$SET FILES TO (READ,WRITE,DATA,UTIL)
```

Parameter Macros. These macros are used with the parameters option to allow run time specification of values within the dependency specification.

When MAKE is invoked, actual parameters are specified through the parameters option. Within the dependency specification, there can be formal parameter references (parameter macros) of the form %*n*, where *n* is a decimal digit (0-9). When the dependency file is parsed, the actual parameters are substituted for the formal parameters in the same manner as for substitution macros. The formal parameter %*n* is then replaced by that element of the list of actual parameters specified in the parameter option (%0 is replaced by the first list element, etc.). If there is no actual parameter for the formal parameter, the replacement is performed using the null string.

Specification of a MAKE dependeny file does not need to be tied to fixed device numbers or directory names. Instead of specifying a file as

```
:F1:DATA.SRC
```

it is far more flexible to specify it as

```
%0DATA.SRC
```

and to suppy it with the device number or directory name when MAKE is invoked.

Special Macros and Macro Constructors. These macros are useful for shorthand specification of dependency nodes (target files, dependency file, task lines). They allow a concise notation that is often more readable and maintainable than typing filenames throughout. The macro constructor %ALL is used in conjunction with an enumeration macro to specify a concatenation of all of the files specified by the list. The special macros %TARGET and %DEPEND are shorthand specifications for the target file list and the list of dependency files, respectively. References to %TARGET and %DEPEND can only appear in task lines. However, references to %ALL are allowed in target lists and dependency lists, as well as task lines.

%ALL. %ALL can be used in a target file list, a dependency file list, or a task line, and is replaced by a concatenation of the enumeration macro elements (separated by commas). A header (the characters preceding the enumeration macro reference, e.g., directory name) is concatenated to the beginning of each element and the trailer (the characters following the enumeration macro reference, e.g., file extension) to the end of each element. Therefore, the same enumeration list can be used for source and object files' different directories.

Using %ALL in the following example:

```
$SET FILES TO (READ,WRITE,DATA,UTIL)
```

the macro constructor %ALL(:F1:%"FILES".SRC) would be expanded as :F1:READ.SRC,:F1:WRITE.SRC,:F1:DATA.SRC,:F1:UTIL.SRC, while the constructor %ALL(%"FILES".OBJ) would be expanded as READ.OBJ, WRITE.OBJ,DATA.OBJ,UTIL.OBJ.

This allows the user to maintain a file list in a single place and yet reference that list in several different forms.

%TARGET. The %TARGET macro represents the target file list. It is replaced in the input stream by the list of target file names (separated by commas) for the dependency node in which it occurs.

%DEPEND. The %DEPEND macro represents the list of dependency files. It is replaced by the list of dependency filenames for the dependency node in which it occurs (separated by commas). It can only be used in task lines.

Using %TARGET and %DEPEND in the standard example:

```
$SET FILES TO ("READ","WRITE","DATA","UTIL")

$ ; The root of the dependency graph
$IF IO.LNK > %ALL(%"FILES".OBJ) THEN
    RUN LINK86 % DEPEND TO %TARGET BIND
$END

$FOR I IN %FILES
  $IF %"I".OBJ > %"I".SRC THEN
      RUN PLM86 % DEPEND OPTIMIZE(3) XREF
  $END
$END
```

In the above case, the first task line is treated the same as

```
RUN LINK86 A.OBJ, ...
```

The second task line is treated the same as

```
RUN PLM86 A.SRC OPT ...
```

Iteration Command

The iteration command is useful in performing a set of MAKE commands over a group of files. The command takes the form of the keyword FOR, followed by the index macro (a name that should not be previously defined), followed by the keyword IN, and a reference to an enumeration macro (generally a list of filenames).

This MAKE command and an END MAKE command line bracket a set of lines that will be iterated over once for each element in the enumeration macro list. These lines can be dependency nodes, macro definitions, or SVCS access definitions.

Example

Given the enumeration macro

```
$SET FILES TO (READ,WRITE,DATA,UTIL)
```

the iteration command would be

```
$FOR I IN %FILES
    $IF %'I'.OBJ > %'I'.SRC THEN
        PLM86 %'I'.SRC
    $END
$END
```

which is the same as

```
$IF READ.OBJ > READ.SRC THEN
    PLM86 READ.SRC
$END
```

```
    .
    .
    .
$IF UTIL.OBJ > UTIL.SRC THEN
    PLM86 UTIL.SRC
$END
```

and the iteration command

```
$FOR I IN %FILES
    $SVCS %'I'.OBJ = IO.DB(%I,,OBJECT)
$END
```

is the same as

```
$SVCS READ.OBJ = IO.DB(READ,,OBJECT)
    .
    .
    .
$SVCS UTIL.OBJ = IO.DB(UTIL,,OBJECT)
```

SVCS Access Definitions

Within the dependency file is a special form of file reference called an SVCS reference. This is a reference to a particular module and variation within a SVCS data base and requires four pieces of information:

- The name of the data base file
- The name of the module
- The variation
- The class of information (source, object, etc.)

Example

```
$SVCS READ.OBJ = IO.DB(READ,,OBJECT)
```

This access definition tells SVCS that all references to the file named READ.OBJ are really references to the object of unit READ in the data base named IO.DB.

In this reference, all fields except the data base name are optional, since SVCS supports the concept of defaults.

When an SVCS file is referenced in a target or a dependency list, the modification information is retrieved from the named data base where it is stored for each variant. In this way, modification to one variant in the data base will not affect the dependencies on the other variants.

The following is the same example used for special macros and macro conductors; however, it includes SVCS access definitions and illustrates the use of many flavors of macros and the macro constructor.

```
$SET FILES TO (READ,WRITE,DATA,UTIL)
$SET VARIANT TO "V3.5"
$FOR I IN %FILES
    $SVCS %"I".OBJ = IO.DB(%I,%VARIANT,OBJECT)
    $SVCS %"I".SRC = IO.DB(%I,%VARIANT,SOURCE)
$END
```

```
; The root of the dependency graph
$IF IO.LNK > %ALL(%"FILES".OBJ) THEN
     $ ; SVCS get of object files
     RUN LINK86 %DEPEND TO %TARGET
$END

$FOR I IN %FILES
  $IF %"I".OBJ > %"I".SRC THEN
       $ ; SVCS get of source file
       RUN PLM86 %DEPEND OPTIMIZE(3) XREF
       $ ; SVCS put of object
  $END
$END
```

Submit File

The submit file created by MAKE contains the tasks (as supplied by the user in the dependency file) necessary to bring the target up-to-date.

Lines added to the submit file will not, in general, make any assumptions about the command line format. These lines will be constructd from the task lines specified by the user (with macros expanded). The following are additional rules concerning the lines in this file:

- No line will exceed 78 characters in length (including the continuation character and the carriage-return and line-feed).
- If a line is to be continued, it will end with a space followed by the character &. Due to expansion of macros, some lines may have to be broken up as continuation lines.

MAKE Invocation

The form of the invocation as well as the method for passing a command line to the program is host specific. The general invocation of MAKE is as follows:

`MAKE` *file_name options*

MAKE Syntax

The general syntax for the MAKE program is presented here for reference.

make_command = *make_pgm command_tail* .

make_pgm = / * *the O.S. dependent specification of the executable object of the program MAKE or MAKE.86* * / .

command_tail = *dependency_file* [*control_list*] .

dependency_file = / * *the O.S. dependent specification of the dependency file, created by the user, to describe the construction of his program (see Chapter 1)* * / .

control_list = {*control*} .

```
control = "TO" submit_file_name
        | attrib_option
        | "GENALL" /* don't check dependencies, gen everything */
        | "PAGELENGTH" "(" length ")"
        | "PAGEWIDTH" "(" width ")"
        | page_option
        | "PARAMETERS" "(" argument_list ")"
        | print_option
        | "TARGET" "(" target_name ")" .

attrib_option = "NOATTRIB"
              | "ATTRIB" [ "(" attrib_string ")" ] .

print_option = "NOPRINT"
             | "PRINT" [ "(" list_file_name ")" ] .

page_option = "NOPAGING"
            | "PAGING" .

submit_file_name = path_name .

attrib_string = /* an O.S. dependent invocation of the ISIS ATTRIB program */ .

list_file_name = path_name .

path_name = /* an O.S. dependent file name */ .

target_name = /* the name of the root of the dependency tree that is to be made */ .

argument_list = argument "," ... .

length = /* a non-zero, unsigned integer */ .

width = /* a non-zero, unsigned integer */ .
```

MAKE Command Options

MAKE accepts the following invocation command options.

TO *submit_file_name*

The TO clause directs MAKE to write the submit file to the named file. The form of the filename is object/source dependent. If the filename is omitted, the extension of the input filename is changed to CSD to create the submit filename.

Example

```
MAKE IO.MKE TO IO.SUB
```

This clause would place the task lines into the file IO.SUB instead of IO.CSD.

GENALL

The GENALL (GA) option specifies that dependencies are not to be checked. If this option is specified, MAKE assumes that all dependencies will fail and therefore puts all of the user specified tasks into the submit file. This permits the user to run the entire generation without relying on partial generations that may exist.

Example

```
MAKE IO.MKE GENALL
```

This option would result in all of the task lines in IO.MKE being placed into the submit file IO.CSD (the default name) regardless of file modification characteristics.

TARGET *target name*

The TARGET (TG) option specifies the complete name of a target file that specifies a dependency node on one of the target lists in the dependency file. It instructs MAKE to use that dependency node as the root of the dependency tree. This option enables the user to specify that only part of the unit be created and the remainder ignored. The default is to use the first dependency node specified in the file as the root.

Example

```
MAKE IO.MKE TARGET(READ.OBJ)
```

This option instructs MAKE to investigate that portion of the dependency graph starting with READ.OBJ as the target. In our example, READ.OBJ would limit the task lines to either compiling READ.SRC or no lines at all.

PRINT/NOPRINT (*list file name*)

The PRINT/NOPRINT (PR/NOPR) option specifies the placement of the listing file. If the optional list filename is omitted, the listing file is printed to a file with the same name as the input file but with the extension changed to LST. The NOPRINT command suppresses the generation of the listing file. NOPRINT is the default.

Examples

```
MAKE IO.MKE PRINT(IO.LST)

MAKE IO.MKE PRINT
```

Both options place a listing file into the file IO.LST.

PARAMETERS

The PARAMETERS (PAR) option matches actual parameters (specified in the invocation line) with formal parameters (specified in the MAKE dependency file).

- If there are more formal than actual parameters, the remaining formal parameters have the value of the null string.
- If there are more actual than formal parameters, the extra actual parameters are ignored.

Example

```
$SET FILES TO (READ,WRITE,DATA,UTIL)
$IF IO.LNK > %ALL(%0%'FILES'.OBJ) THEN
    RUN LINK86 %DEPEND TO %TARGET
$END
```

```
$FOR I IN %FILES
    $IF %0%'I'.OBJ > %0'I'.SRC THEN
        RUN PLM86 %'I'.SRC %1
    $END
$END
```

The MAKE file EXAMPL.MKE has two parameter macros: %0, which specifies a drive for the object files, and %1, which specifies options for the compiles.

Invocation of MAKE to process this MAKE file would take the form

```
MAKE EXAMPL.MKE PAR(:F1:,'COMPACT OPTIMIZE(1)')
```

which would substitute :F1: for each occurrence of %0 and COMPACT OPTIMIZE(1) FOR %1.

PAGELENGTH *length*

The PAGELENGTH (PL) option sets the maximum number of lines per page in the listing file.

- The length specified must be an unsigned integer from 5 to 65,535.
- If either the NOPAGING or NOPRINT option is being used, this option is ignored.

PAGEWWIDTH *width*

The PAGEWIDTH (PW) option sets the maximum number of bytes for a line in the listing file.

- The width specified must be an unsigned integer from 60 to 255.
- If the NOPRINT option is being used, this option is ignored.

PAGING/NOPAGING

The PAGING/NOPAGING (PI/NOPI) option specifies that page ejecting and page headers should or should not exist in the listing file at every page according to the PAGELENGTH command.

ATTRIB/NOATTRIB

The ATTRIB/NOATTRIB (AT/NOAT) option permits the user to reset the modification (dirty) bit for files that are in ISIS directories. The ATTRIB option allows the user to state the name of the program that will perform the resetting. The NOATTRIB option supresses the resetting of the modification bit. The default string is ATTRIB.

NOTE

This modification bit is available only on ISIS-II(W), the winchester ISIS. This option is ignored for all files that reside on the Network Resource Manager (NRM) for systems on NDS-II.

MAKE FILES

Make opens an input file, an output file, and optionally a listing file.

Input File (MAKE File)

The input file is the file of dependency information specified by the user on the command line.

Output File (Submit File)

The output file is always created. This file is the submit file that is built up of user-specified tasks. The user can direct this file to a specified file by using the TO command.

Listing File

The listing file contains a header summary, a dependency file listing, a dependency graph listing, and a file summary.

- This file is not constructed unless the user requests it through the PRINT command.
- Continuation lines in the listing are marked with a dash just to the left of the continuation text.
- Only this file contains error messages resulting from improper specifications in the dependency file.

Header Summary

The header summary contains the name of the MAKE dependency file, the name of the constructed submit file and a list of the controls specified by the user.

Dependency File Listing

This section of the listing contains the dependency file as specified by the user.

Dependency Graph Listing

This section of the listing contains a representation of the dependency graph with levels of indention used to denote dependency. Each line, representing a target, a dependency, or both, have the following fields:

- Depth—(3 digits) depth from the root (root is depth 1)
- Separator—(2 blanks)
- Indentation—(4 (DEPTH-1) blanks)
- Name—name of the target/dependency file at this node
- Separator—(blank,colon,blank) only if there are attributes
- Attributes—attributes relating to this file

File Summary

The file summary lists the number of lines and number of errors within the MAKE file.

If MAKE detects an error in the dependency specification, a message is placed in the listing file.

Example

```
***ERROR nnn IN LINE lll "ttt"; message
```

where

nnn	provides the error number.
lll	provides the line number.
ttt	provides the input text near where the error was detected.
message	provides an explanation of the error.

At program completion, MAKE will return the following completion code:

0 if no errors were detected

2 if errors were detected

MAKE Error Messages

The following is a list of error messages generated by MAKE.

```
 1. ENUMERATION MACRO REQUIRED IN %ALL EXPANSION

 2. UNEXPECTED END OF FILE ENCOUNTERED IN %ALL
    EXPANSION

 3. UNEXPECTED END OF FILE ENCOUNTERED IN MACRO
    DEFINITION

 4. UNEXPECTED END OF FILE ENCOUNTERED IN FOR LOOP

 5. UNEXPECTED END OF FILE ENCOUNTERED IN TASK LINE

 6. UNEXPECTED END OF FILE ENCOUNTERED IN DEPENDENCY
    SPEC

 7. REFERENCED MACRO IS UNDECLARED

 8. SET COMMAND REQUIRES A MACRO NAME

 9. FOR STATEMENT REQUIRES AN ENUMERATION MACRO

10. MATCHING SINGLE OR DOUBLE QUOTE NOT FOUND

11. STRING LENGTH LIMIT IS 255 BYTES

12. 'TO' EXPECTED IN MACRO DEFINITION
```

```
13. 'IN' EXPECTED IN FOR STATEMENT

14. MAKE STATEMENT NOT ALLOWED IN TASK LINES

15. UNKNOWN MAKE COMMAND

16. CONTINUATION OF A MAKE COMMAND EXPECTED, FIRST
    CHAR MUST BE '$'

17. MAKE ERROR: MACRO STACK UNDERFLOW
```

An error in the MAKE program occurred. Please document it and report the error to your Intel representative.

```
18. MACRO EXPANSION SUPPORTED TO A DEPTH OF 16

19. COMMA EXPECTED IN NAME LIST

20. 'THEN' REQUIRED IN DEPENDENCY SPECIFICATION

21. NAME USED IN TARGET OPTION NOT FOUND, FIRST
    TARGET USED
```

Fatal Command Errors

The first fatal command error encountered causes MAKE to report the error to the console and halt processing. Fatal errors result from either an illegal or unknown option/option value in the invocation and are as follows:

```
UNKNOWN OPTION

ILLEGAL OPTION VALUE FOR OPTION:

DEPENDENCY FILE REQUIRED AS FIRST OPTION;

RESPECIFICATION OF OPTION NOT ALLOWED:

TOO MANY ARGUMENTS IN PARAMETERS OPTION

FATAL OBJECT/SOURCE CODE INTERFACE ERRORS
```

The fatal object/source code interface errors occur from calls to the operating system that cannot be handled because of user error or lack of system resources. If the error occurs during an I/O operation, the following will be displayed:

MAKE *supervisor* SYSTEM CALL ERROR

FILE: *file name*

```
ERROR: object or source error message

MAKE TERMINATED
```

If the error occurs during an non-I/O system call, the message is as above without the line containing the filename.

CHAPTER 3
SOFTWARE VERSION CONTROL SYSTEM (SVCS)

This chapter presents a more in-depth presentation of SVCS. Its purpose is to provide you with more detailed information on the structure of SVCS.

What Is SVCS?

SVCS allows you to set up a complete data base to manage software projects. SVCS provides the capability to track changes to program source code, maintains variations of the source and object code modules for a program, and controls access to these modules in a multi-programmer environment. Essentially, you can group related software modules, as well as variations of those modules, within a single data base.

SVCS automatically retains history information on every change to a software module, including who made the change and when and why the change was made. It also allows different versions of a module to be uniquely identified. This makes SVCS well suited to applications that require customized software.

SVCS Structure

SVCS maintains a data base of units that may be checked out and either modified or returned. An SVCS unit is a reference to a given data base and contains the following constructs (see figure 3-1):

- Program units
- Unit classes
- Variations

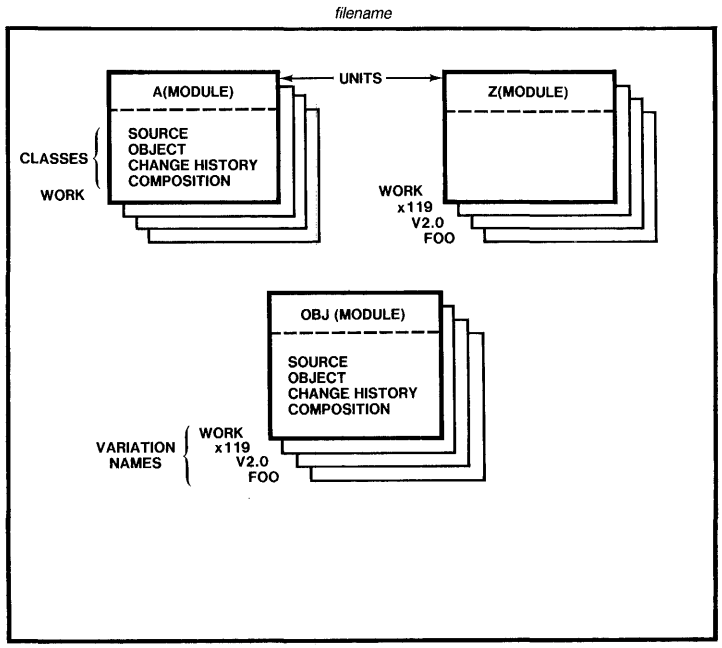

Figure 3-1. Software Version Control 121958-3

Program Units

Program units are an expression of the granularity of the data base. They represent not just one piece of information (such as the text of a source module) but several pieces of information concerning a single piece of the program that is being developed.

Developing a program requires manipulation of several different fragments of that program. These may include source files and their related object files, include files, and linked objects (e.g., complete programs, overlays, and tasks). One of these fragments is referred to as a unit within SVCS. A unit is not just one piece of information, but is all information relating to a particular fragment of the program. The majority of these will be module units that contain a piece of source (such as a module), a related object file (if one exists), change history for the unit, and optionally some composition information.

In addition to the source fragments, a project will need to keep track of the linked objects (such as tasks, overlays, and complete programs) that are generated. Linked objects are stored in system units that are identical to module units but do not store any source.

Module Units

A module unit is a unit of source (either a source module or an include file). Within a data base there can be any number of such units; however, the best use of the data base would be to include only those sources that are related to form a logical subset of the program being developed (refer to the section on optimal use of SVCS in this chapter).

System Units

System units are the synthesis of one or more source modules created during the construction of a program. Each of these units is usually the linkage of the objects generated from the source modules by a translator or the linkage of several link files. The units provide the following:

- A place to accumulate internal interface information
- A place to permit outside interface to the modules
- A place to store information that is useful for functional documentation of the modules as a group
- A place to store generation information

SVCS makes no distinction between system units and module units. They are treated in identical fashion. The distinction is made only through their usage.

Unit Classes

Each program unit has one to four or more classes of information associated with it. For a module unit, these classes would be the source module, the object module that results from compiling the source, the change history for the source, plus one class for any related information, such as a list of include files that are used by the source, or documentation describing either the module or the interface exported by the module. For a system unit, the classes would be the same (without the source class).

SVCS predefines some of the characteristics of these classes, as defined below:

- Source (SO)—This class holds the source module. It is generally present only for module units.
- Object (OJ)—For a module unit, this class is the object that was generated from the source module by some translator. For a system unit, it is the object module that was generated from combining the object modules of several module units or several system units.
- History (HT)—This class contains the who, what, when, and why of a source (or object) change and is available to the programmer. It is logged automatically every time a change is made to the source class of a unit. Changes to the history information can be made any time, not just when the source module of an object module is returned to the data base with a PUT command.
- Composition (CP)—his class can be used arbitrarily by the user. It is available for any purpose that will help document the system. Typically, it might be used to contain a list of unit names that are used for the construction of a particular unit and/or the tasks required for construction. For a module unit, this list could be a list of include files to document dependencies. For a system unit, this list includes the names of the module units whose objects are to be linked together to form the system object. Generation procedures, interface specifications, or module documentation (as well as the MAKE file) could also be stored in this class.

In summary, classes contain the information associated with a unit (module or system) that is being either developed or maintained.

Variations

The variation (variant) mechanism allows the programmer to have copies (versions) of the same unit that serve different developmental needs, markets, or end uses. SVCS supports such development by allowing these different sources to be stored as variations on a single source. A variant represents different flavors of a system or consequent version of a module. Each variation has its own name and its own share of the classes. This enables all of the objects for the different variations to be stored in the same data base.

The variation construct allows the programmer to have "shadow" copies of each unit to fulfill various requirements of development and maintenance of the program. The variations are called shadow copies because they do not actually take up separate space in the data base, but are merged together, resulting in substantial space savings. One history file (class) serves for all variations of a unit, as does one source file. This technique enables the user to spin off a new variation from an old one at any time.

The names of the variations have no inherent meaning; thus they serve merely as a means of identification. The variation construct can either be used as a version tracking mechanism or as a mechanism for tracking variations that are created to fulfill special requirements of different end products.

For each unit, SVCS recognizes the default variant to be the current working version, WORK. A GET command of a source module with no variant specified will retrieve the WORK variant. Specifically, named variants may be retrieved, modified, and returned, just like the current working version. Variants are created with the ADMIN command and FROM existing variants.

Optimal Use of SVCS

Because the SVCS data base is built upon the existing file system, it is fragmented into many files (depending on the number of units, classes, and variants). There will be at least three files per unit in the data base plus one per variant. For large projects, therefore, there will be many files, with potential for programmer contention accessing the data base.

For this reason, it may be wise to break up a large project into more than one data base. There is no penalty for having more than one data base in a project, especially if they are broken along logical lines (nucleus, I/O system, application, or overlay boundaries).

There is no physical limit to the number of units in a data base (other than those of the file system). However, keeping the number of units and variants under 50 or 100 should limit the number of files (and lengthy disk access) and reduce programmer contention for the data base.

It is recommended that SVCS data bases occupy entire NDS-II sub-directories, rather than mix user-accessible files with SVCS-accessible files.

Major Functions of SVCS

SVCS allows you to perform the following functions on the data base:

- ADMIN function—allows you to create and delete units and unit variations. It also allows you to manipulate attributes associated with the variants.
- GET function—allows you to retrieve a unit from the data base either to read or to modify. Any variant and class of information for a unit is available to the GET command.
- PUT function—lets you return a unit to the data base (with WRITE permission) and have the unit's change history update. This allows you to get a source with WRITE permission, edit it, and then return it to the data base. When PUT of source occurs, information concerning the changes is automatically logged. At this time, you can also request that commentary text about the change be placed with the change history.
- RETURN function—enables you to return modification permission for a source that was acquired from the data base with WRITE permission. This treats the GET and RETURN transactions as though they never occurred (no record exists in the change history). Both the GET and the PUT function act as intelligent COPYs.

SVCS Commands

SVCS accepts the following commands for processing.

GET

The GET command is used to retrieve information from the data base.

TO *file_name*

This clause names the file that is to receive the information requested in the GET command. It is not optional.

Example

```
SVCS GET pgm.db(main) TO :f1:main.src
```

The options associated with a GET command (WRITE, IDENTIFIER, HISTORY, COMMON) are described as follows.

WRITE [(*variant_list*)]

The WRITE (WR) option requests write permission on the piece of information that is to be retrieved (checked out) from the data base for the purpose of modification.

- Permission is granted if no one has it checked out and if the requester has write permission on the data base.
- The optional variant list allows the requester to retrieve multiple variants at the same time.

Example

```
SVCS GET pgm.db(main) TO :f1:main.src WRITE
```

IDENTIFIER

The identifier (ID) clause is used to designate the person requesting the information from the data base. It is required for WRITE permission.

- If the ID is omitted on a GET command for WRITE permission, SVCS prompts for it with ID:. The ID is used to identify who has the module.
- The ID entered in response to the above prompt will consist of all the characters between the prompt and the carriage return.

Example

```
SVCS GET pgm.db(main) TO :f1:main.src WRITE ID (Sheila)
```

HISTORY

In a GET command, the HISTORY (HT) option annotates the source code with the information from the change history file.

- In a GET command, HT is valid only for the source class and only for read permission so that your source file cannot be corrupted with history information.
- It allows the programmer to find out when, why, and by whom changes were made to the source code.

Example

```
SVCS GET pgm.db(main) TO :f1:main.src HISTORY
```

COMMON

The COMMON (CM) option directs SVCS to add lines to the retrieved source code to delineate any lines that are common to all of the variants listed in the WRITE option variant list and the variants requested by the GET command.

- It is valid only for the source class.

Example

```
SVCS GET pgm.db(main) TO :f1:main.src &
 WRITE (x119,v1.2) ID (user) COMMON
```

PUT

The PUT command enables you to return information to the data base that was checked out for WRITE permission.

- The PUT command is not valid if the requester did not check it out with WRITE permission (verified through ID comparison).
- When the PUT command fails, it is ignored.

The options associated with a PUT command (FROM, WRITE, IDENTIFIER, HISTORY) are described as follows:

FROM *file_name*

The FROM clause places the named file into the data base.

- It is not valid if the piece of information specified is not checked out by the person designated by the ID option.

Example

```
SVCS PUT pgm.db(main) FROM :f1:main.src
```

This command causes SVCS prompt for the ID and the history information as explained below.

WRITE [(*variant_list*)]

The WRITE (WR) option is used as a counter-check for the GET command.

- If the variant list is not the same as specified in the GET command, the PUT command fails.
- In general, the user will not choose to use this command but rather rely on the default. The default is to use the list specified on the GET command.

Example

```
SVCS PUT pgm.db(main) FROM :f1:main.src WRITE (X119,V1.2)
```

IDENTIFIER

The identifier (ID) option is used to designate the person replacing the information into the data base.

- The options is used to verify that the person replacing the information in a PUT command is the same person who checked it out.
- If the ID is not provided, SVCS prompts with ID:. The ID entered consists of all the characters between the prompt and the carriage return.
- If the PUT command is used for the source class, the ID is logged with the date, time, and optional data supplied by the user with the HISTORY option.

Example

```
SVCS PUT pgm.db(main) FROM :f1:main.src ID (Andy)
```

HISTORY

The HISTORY (HT) option allows the user to supply information about why the file was modified.

- It is terminated by the first non-quoted right parenthesis encountered.
- The user-supplied information can be retrieved by either doing a GET command of the history class associated with a unit or by using the HISTORY option on a GET of the source class for a module unit.
- If the HISTORY option is omitted from a PUT command on a source class, SVCS prompts for it with the prompt HISTORY:. This prompt is displayed at the beginning of each line until a zero-length line is encountered (a line containing only a carriage return/line-feed).

RETURN

The RETURN (RT) command returns write permission for the designated variants. This command is used if the programmer gets a file and then decides not to modify it. As with the PUT command, the ID option is required and the WRITE option can be specified. If the WRITE option is used, the variant list must match what was specified in the GET command.

Example

```
SVCS RETURN pgm.db(main) ID (Stu)
```

ADMIN

The ADMIN command allows SVCS to perform the various administrative functions required for maintaining a data base.

The options associated with ADMIN (CREATE, ADD, DELETE, WRITEACCESS, DEFAULTACCESS, PRINT) are described as follows.

CREATE

The CREATE (CA) option permits the administrator of the data base to create a dat base. If the file given as the data base name already exists, it is an error. SVCS will then prompt, asking whether to overwrite the existing file or to simply abort. If the named data base already exists, it is overwritten. Note that after creating a database that is to be shared, be sure to change its world access rights so that others may modify it.

Example

```
SVCS ADMIN pgm.db CREATE
```

ADD and DELETE

The ADD option permits the administrator of the data base to add either units or variants to the data base at any time. Units added will automatically have all variants defined for that data base.

Example

```
SVCS ADMIN pgm.db ADD (VARIANT=projA,projx,proj1)

SVCS ADMIN pgm.db ADD (UNIT=main,data,init)
```

This example adds three units (main, data and init) to the data base.

The DELETE (DL) option allows the administrator to delete units and variants from the data base at any time.

Example

```
SVCS ADMIN pgm.db DELETE (VARIANT=projB)
```

Both the ADD and DELETE options allow the administrator to specify the creation or deletion of one or more units (module and system) and one or more variants.

- UNIT. A unit can either be created in the data base as empty (if the FROM clause is not used) or with the source class file initialized to the contents of a named file (if the FROM clause is used). It is created with an empty object class file, a variant with the name WORK, and a history class file with an entry for the creation. It has both read and write permission.

  ### Example

  ```
  SVCS ADMIN pgm.db ADD (UNIT = main FROM :f1:main.src
  ```

  This command adds the unit main while initializing it to the contents of the file :f1:main.src.

  NOTE

  The FROM clause initializes the WORK variant of the unit. All other variants of the unit remain uninitialized. Thus the user should define units before defining variants.

- VARIANT. When a variant is either created or deleted, the action occurs on the entire data base. A variant is always created from an existing variant and has the identical contents of that variant. In creation, the variant is stated in the FROM clause. If not, it is created from the WORK variant. The variant is created with write access enabled but with no associated default accesses.

WRITEACCESS

The WRITEACCESS (WA) option allows the administrator of the data base to allow or disallow writing to a specified variant within the data base. Any GET command requesting permission to write on a variant where write access is disallowed will fail.

Example

```
SVCS ADMIN pgm.db ADD(VARIANT = x119 FROM WORK) &
  WRITEACCESS (x119=FALSE)
```

The preceding example shows that the administrator wishes to create a prototype variant x119 from the WORK variant with write access disallowed. By disallowing write access, the owner of the data base has rendered the variant unmodifiable.

DEFAULTACCESS

The DEFAULTACCESS (DA) option allows the administrator of the data base to set up default accesses to any variant in the data base.

- If the ID list is not specified, the specified variant becomes the default for all identifiers that are not in any other variant default list.
- The word NONE eliminates all identifiers from the list for that variant.
- The word ALL causes the specified variant to become the default for all identifiers. This global default can be modified for individual users through further DEFAULTACCESS definitions.
- Identifiers in the ID list are stored with the variant in the data base file.
- An identifier will never appear on the default list of more than one variant of a unit because if it is added to one list while on another, it is automatically deleted from the old list.

Example

```
SVCS ADMI pgm.db DEFAULTACCESS (x119=ALL) &
     DEFAULTACCESS (WORK=CHRIS,ELSA,MATT,ERIC)
```

This command would direct all accesses to the data base in which the variant was not named to the variant named x119. The exception is that the default variant for the four named programmers would be the one named WORK. In this way, all outside references to the data base (generally from groups requiring prototypes) can be directed without the outside groups needing to know specifics. At the same time, it permits the programmers working on the program to use the default.

PRINT

The PRINT (PRI) option provides the user with a formatted directory of the data base.

Example

```
SVCS ADMIN pgm.db PRINT (db.lst)
```

SVCS Command Options

The following options can be applied to any of the SVCS commands.

PROMPT/NOPROMPT

The PROMPT/NOPROMPT (PRO/NPRO) option tells SVCS whether to prompt on certain error conditions or simply issue the error message and exit. The default for this command is PROMPT.

TIMEOUT/NOTIMEOUT

The TIMEOUT/NOTIMEOUT option establishes the defined period of time SVCS will try to gain control of the data base before giving up and exiting to the command level. Since more than one person may wish to access the data base at any time, SVCS will pause and try again if another SVCS command is executing.

- The default time is 300 seconds.
- TIMEOUT can override this default with a user-specified number of seconds.
- NOTIMEOUT instructs SVCS to wait forever.

Example

```
TIMEOUT (30)
```

SVCS Invocation

The form of the invocation, as well as the method for passing a command line to the program, is host specific. The general form of invocation is as follows:

```
SVCS { GET    }
     { PUT    } data_base_spec options
     { RETURN }
     { ADMIN  }
```

SVCS Syntax

The general syntax for the SVCS program is presented here for reference.

```
SVCS_command = SVCS_pgm command_tail .

SVCS_pgm = /* the O.S. dependent specification of the executable object of the
             program SVCS.86 */ .

command_tail = command_type [ common_options ] .

command_type = "GET" db_spec GET_options
             | "PUT" db_spec PUT_options
             | "RETURN" db_spec RETURN_options
             | "ADMIN" db_spec ADMIN_options .

db_spec = db_name [ "(" [unit_name] [ "," [variant]
                    [ "," [class] ] ] ")" .

db_name = file_name .

unit_name = identifier .

variant = identifier .

class = "SOURCE"
      | "OBJECT"
      | "HISTORY"
      | "COMPOSITION" .

GET_options = GET_option ... .

GET_option = "TO" filename
           | "WRITE" [ "(" variant_list ")" ]
           | "ID" "(" identifer ")"
           | "HISTORY"
           | "COMMON" .
```

```
PUT_options = PUT_option ... .

PUT_option = "FROM" filename
           | "WRITE" [ "(" variant_list ")" ]
           | "ID" "(" identifer ")"
           | "HISTORY" "(" history_string ")" .

RETURN_options = RETURN_option ... .

RETURN_option = "WRITE" [ "(" variant_list ")" ]
              | "ID" "(" identifier ")" .

ADMIN_options = ADMIN_option ... .

ADMIN_option = "CREATE"
             | "ADD" "(" add_item ")"
             | "DELETE" "(" delete_item ")"
             | "PRINT" "(" list_file_name ")"
             | access_options .

add_item = "UNIT" "=" add_unit_list
         | "VARIANT" "=" variant [ "FROM" variant ] .

delete_item = "UNIT" "=" unit_list
            | "VARIANT" "=" variant_list .

unit_list = unit_name "." ... .

add_unit_list = add_unit "," ... .

add_unit = unit_name [ "FROM" file_name ] .

access_options = "WRITEACCESS" "(" variant_assign ")"
               | "DEFAULTACCESS" "(" variant [ "=" id_spec ] ")" .

id_spec = "ALL"
        | "NONE"
        | id_list .

variant_assign = variant_list "=" boolean .

variant_list = variant "," ... .

boolean = "TRUE"
        | "FALSE"

id_list = identifier "," ... .

file_name = /* O.S. dependent file name */ .

identifier = /* an argument as defined by [2] */ .

common_options = PROMPT_option
               | TIMEOUT_option .

PROMPT_option = "PROMPT"
              | "NOPROMPT" .
```

```
TIMEOUT_option = "TIMEOUT" "(" num_seconds ")"
               | "NOTIMEOUT" .

num_seconds = /* number of seconds to wait before timing out */ .
```

SVCS Files

SVCS opens a number of different files depending on the options specified by the user. The various files are described as follows.

Data Base File

The data base file is the heart of SVCS. It contains all of the information about the various units, variants, classes, and access rights within the overall data base.

This file is generally centrally controlled during a software development process. Any user can request permission to read the information within the file; however, if the user wants to modify the data, he must have write access (as designated by the host operating system). For write requests, the data base file acts as a lock, allowing only one write request to be honored at any one time. However, once that request has control of the data base file, the user can open any of the other files seeking write permission without fear or deadlock.

If the request to get control of the data base file fails because someone else is either reading from or writing to that file, SVCS will wait a short time (approximately one second) and then retry the request. SVCS will wait until that control is relinquished. The amount of time that SVCS will wait for the data base can be controlled by the TIMEOUT option as discussed in the summary of SVCS commands contained in this chapter.

Auxiliary Files

The SVCS data base is not constructed as one large file but as a central data base file that references several auxiliary files. These files contain information that is specific to each unit. They have the same name as the data base file but with the extension changed. The extension on these files is three characters long with one of the characters being a decimal digit (0—9) and the remaining two being upper case alphabetic. The extensions are allocated as needed and never reused. This allows for archiving, deleting, and then restoring from the archives.

Retrieved Files

Either a retrieved object or history file results in a straight copy of the auxiliary file to the file specified in the TO command. These files are never encoded by SVCS.

A retrieved source class file or composition class (both are stored in an encoded format) is decoded and then placed into the file specified by the user. For source, the user is also able to specify either or both of two options (HISTORY and COMMON) that will affect what is placed in the output file. The results of these options are discussed in the following paragraphs.

History Option

If the history option is specified on a GET command, the history file is written to the output file preceding the source file lines. It looks like a listing, which is why it cannot be changed. Each line of the source file is preceded by a five-character change number (representing the change that created that line and referencing one of the entries in the change history) followed by a blank.

Example

```
#      1    07/14//82
VARIANTS:
TEXT: INITIALIZED

#      2    07/30/82              CHRIS
VARIANTS: WORK
TEXT: This change reflects corrections to the
third, fourth, and fifth lines.

     1    This example shows the result of using the
     1    HISTORY option on a GET of source. The
     2    first, second, and sixth lines show up as
     2    the original lines (from when the file was
     2    initialized). The other lines were changed
     1    at a later time to correct errors.
```

Common Option

If the common option is specified on a GET command (valid only on a source class), lines that are common to all variants in the variant list are preceded by the line shown in the example.

Example

```
    SVCS allows the user to GET more than
    one variant at the same time. The
$SVCS COMMON
    COMMON option can be used on such a
    GET to delineate the lines that are
    common to all variants. This example
$SVCS END COMMON
    shows the result of just such a GET.
    The third, fourth, and fifth lines are
    common to all variants of the GET. The
    other lines only exist in the main
    variant of the GET.
```

Stored Files

A stored object or history file results in a straight copy to the auxiliary file from the file specified in the FROM clause. These files are never encoded by SVCS.

A stored source class file or composition class file is compared against the lines in the associated auxiliary file and merged in with those lines in an encoded format to save space.

If the file being stored was retrieved with the COMMON option, the lines inserted to designate the common source are stripped out before the store actually takes place.

SVCS Error Messages

This section lists all the possible error messages associated with the SVCS program.

Command Errors

Command errors result from either an illegal or unknown option/option value in the invocation line. SVCS prompts for respecification of the erroneous information if the prompt option is in effect; otherwise, it merely displays the error message and exits. There are messages that indicate internal errors; when one of these appears contact your Intel representative.

The SVSC command error messages are as follows:

1. DATA BASE FILE REQUIRED

2. FILE IS NOT AN SVCS DATA BASE FILE: *file_name*

3. UNKNOWN VARIANT NAME: *variant*

4. CANNOT DELETE THE WORK VARIANT

5. CANNOT CREATE, UNIT ALREADY EXISTS: *unit_name*

6. CANNOT CREATE, VARIANT ALREADY EXISTS: *variant_name*

7. Only 16 VARIANTS may be added in one ADMIN.
   Please break up the SVCS invocation into several
   invocations.

8. ADMIN not interactive, please respecify with
   complete command

9. UNKNOWN VARIANT IN WRITE LIST *variant_name*

10. FILE NOT CURRENTLY CHECKED OUT

11. CANNOT RETURN WRITE PERMISSION, OWNER IS *owner_name*

12. CANNOT PUT FILE, OWNER IS *owner_name*

13. WRITE ACCESS NOT ALLOWED ON VARIANT: *variant_name*

```
14. Duplicate occurence of variant : variant_name
    Occurrence in WRITE listing ignored.

15. SVCS INTERNAL ERROR #1
    Please contact your Intel representative

16. SVCS DIRECTORY EXTENSIONS EXHAUSTED
    Please contact your Intel representative

17. ILLEGAL OPTION VALUE FOR OPTION: option_name

18. UNKNOWN UNIT NAME: unit_name

19. ID OPTION REQUIRED

20. CANNOT PUT FILE, IT IS NOT CHECKED OUT

21. HISTORY OPTION REQUIRED

22. SVCS INTERNAL ERROR #2
    Please contact your Intel representative

23. UNKNOWN OPTION: option_name

24. INSUFFICIENT ROOM FOR LARGEST RECORD

25. INSUFFICIENT ROOM TO SYNCHRONIZE

26. REQUESTED FILE IS CHECKED OUT TO: id_name
```

Fatal Object/Source Interface Errors

These errors occur from calls to the operating system that cannot be handled because of user error or lack of system resources. If the error occurs during an I/O operation, the error message is as follows:

```
SVCS supervisor I/O ERROR
FILE: file_name
ERROR: message
SVCS TERMINATED
```

If the error occurs during a non-I/O system call, the message will the same as the preceding one without the line containing the filename.

SVCS Prompt Messages

If the PROMPT option is in effect, certain error conditions cause SVCS to prompt the user for correct input. The following prompt messages are supported by SVCS:

```
A data base name is required, please enter one.
DATA BASE:

An SVCS command is required. The command choices are:
      GET - to retrieve information from SVCS
      PUT - to put information into SVCS
      RETURN - to return WRITE permission
      ADMIN - to perform administrative functions on SVCS
Please enter one of these commands:

Unknown variant (variant$name), please respecify
VARIANT:

An identifier is required for this operation. Please enter
your id
ID:

Please specify the name of the file that is to be copied to
"TO" FILE NAME:

A unit name is required for this operation.
Please specify one
UNIT:

Please specify the name of the file that is to be copied
from
"FROM" FILE NAME:

An explanation of this PUT is required for the modification
history.
Please type in an appropriate entry
HISTORY:
```

APPENDIX A
SUMMARY OF MAKE/SVCS COMMANDS AND PROMPTS

Table A-1. Summary of MAKE Commands

Command	Explanation
TO	Directs the writing of the submit file
GENALL (GA)	Specifies that dependencies are not to be checked and that everything is to be generated
TARGET (TG)	Specifies the target file specified in a target list (results in a partial generation)
PRINT (PR)	Specifies placement of listing file
NOPRINT (NOPR)	Suppresses generation of the listing file
PARAMETERS (PAR)	Matches actual parameters with formal
PAGELENGTH (PL)	Sets the number of lines per page for the listing file
PAGEWIDTH (PW)	Sets the maximum number of bytes for a line in the listing file
PAGING/NOPAGING (PI/NOPI)	Controls page ejecting and headers in the listing file
ATTRIB/NOATTRIB (AT/NOAT)	Permits the user to reset the modification bit of files in the ISIS directory

Table A-2. Summary of SVCS Commands

Command	Explanation
GET	Retrieves information from the data base
TO	Names the file that is to receive the information
WRITE (WR)	Requests write permission on information
IDENTIFIER (ID)	Designates person requesting information
HISTORY (HT)	Includes history information in the retrieved file
COMMON (CM)	Allows for manipulation of common information
PUT	Returns information (checked out with write permission) to the data base
FROM	Name of file to be copied into the data base
WRITE (WR)	Countercheck for GET command
IDENTIFIER (ID)	Verifies that the person replacing the information is the same person who checked it out
HISTORY (HT)	Designates why changes were made
RETURN (RT)	Returns write permission for the designated variant
WRITE (WR)	Same as for PUT
IDENTIFIER (ID)	Same as for PUT
ADMIN	Allows for the administrative functions associated with a data base
CREATE (CA)	Creates a data base
ADD	Permits the adminstrator to add units or variants to the data base
DELETE	Permits the adminstrator to delete units or variants from the data base

Table A-2. Summary of SVCS Commands (Cont'd.)

Command	Explanation
WRITEACCESS (WA)	Allows the administrator to allow/disallow writeaccess to a given variant
DEFAULTACCESS (DA)	Allows the administrator to establish default access to any variant
PRINT (PRI)	Provides formatted directory of the data base
These options apply to the GET, PUT, RETURN and ADMIN commands	
PROMPT (PRO) NOPROMPT (NPRO)	Instructs SVCS as to whether to prompt or simply exit on error conditions
TIMEOUT	Establishes time period SVCS will try to gain control of data base before giving up
NOTIMEOUT	Instructs SVCS to attempt to gain control without giving up

Table A-3. Summary of SVCS Prompt Messages

Prompt	Explanation
PLEASE ENTER ONE OF THESE	SVCS command missing
GET	Retrieve information
PUT	Put information into SVCS
RETURN	Return write permission
ADMIN	Perform adminstrative functions on SVCS
DATA BASE:	Data base name either missing or incorrect
UNIT:	Unit name missing or incorrect
VARIANT:	Variant name missing or incorrect
TO FILENAME:	Filename where information is to be copied is missing
FROM FILENAME:	Filename that contains information to be put into the data base is missing
ID:	Identification of user is required
HISTORY:	Explanation of change is required

APPENDIX B
ADDITIONAL INFORMATION FOR THE SERIES III USER

This appendix contains information specific to the Intellec Series III Microcomputer Development System. It covers the following subjects:

- Series III literature
- Hardware and software required
- System resources used by PMTs
- System-specific examples of invocation lines and commands

Series III Literature

Information describing the general operation of the Series III is provided in the following manuals:

- *Intellec Series III Microcomputer Development Product Overview*, 121575
- *Intellec Series III Microcomputer Development System Console Operating Instructions*, 121609
- *Intellec Series III Microcomputer System Programme's Reference Manual*, 121618

Hardware and Software Required

To run the PMTs, the following hardware and software are required:

- Intellec Series III development system (with RUN version 1.5 or later)
- ISIS-II Operating System (version 4.1 or later)
- 8086-based Utilities

System Resources

The amount of memory available depends upon the amount of memory in your system. The Series III can be expanded up to one megabyte of memory addressable by the 8086. PMTs require approximately 96K bytes. More memory must be added to accommodate additional workspace and programs.

Invocation Line

To invoke PMTs in the 8086 execution environment of the Series III, preface the invocation line with the RUN command. The ISIS-II operating system prompt is a hyphen (-).

The general format of the MAKE invocation is

```
-RUN MAKE
```

MAKE signs on with the following message:

```
Series-III MAKE, Vx.y
```

The general format of the SVCS invocation is

```
-RUN SVCS
```

SVCS signs on with the following message:

```
Series-III Software Version Control System, Vx.y
```

INDEX

ADD, 1-9, 3-7
ADMIN, 1-5, 1-8, 1-9, 3-4, 3-7
ampersand, 3-8
AT/NOAT, 2-11
ATTRIB/NOATTRIB, 2-11
auxiliary files, 3-12, 3-13

classes, 3-1
command errors, 2-14, 3-14
command line, 1-6, 2-2, 2-3
COMMON option, 3-5, 3-13
completion code, 2-13
composition, 1-2, 3-3
continuation lines, 2-12, 3-8
CP, 3-3
CREATE, 1-9, 3-7

DA, 3-9
data base, 1-2, 1-3, 1-5, 1-8, 1-9, 3-1, 3-2, 3-12
DEFAULTACCESS, 3-9
DELETE, 1-10, 3-7, 3-8
dependency file, 1-6, 2-1, 2-2, 2-10, 2-12
dependency graph, 1-3, 2-1, 2-10, 2-12
dependency nodes, 1-3, 1-6, 2-1, 2-3, 2-6, 2-10, 2-12
DL, 3-8
dollar sign, 1-6, 2-2, 2-3

END, 1-6, 2-3
environment, 1-2
error message
  MAKE, 1-7, 2-13
  SVCS, 3-14

fatal command errors, 2-14, 3-15
FROM, 3-6

GENALL, 1-5, 2-9
generation, 1-2
GET, 1-2, 1-8, 3-4, 3-6

history, 1-1, 1-3, 3-3, 3-5, 3-7, 3-13
HT, 3-3, 3-5, 3-7

ID, 3-5, 3-6
IDENTIFIER, 3-5, 3-6
incorporating MAKE and SVCS, 1-5
input file, 2-12
invocation
  MAKE, 1-6, 2-8, 2-11, 2-13, B-1
  SVCS, 1-8, 3-10, B-2
iteration commmand, 2-2, 2-6
ISIS-II, v, B-1
ISIS-II(W), v, 1-2, 2-11
ISIS-III(N), v, 1-2

listing file, 2-12

macro definitions, 2-2, 2-3, 2-6
MAKE, 1-1, 1-6, 2-1
MAKE command line, 1-6
MAKE Command Options, 2-9
MAKE commands, 2-2, A-1
MAKE file, 1-3 thru 1-5, 1-6, 2-8, 2-12
module housekeeping, 1-1
module unit, 3-2

NDS-II, v, 1-2, 2-11
Network Resource Manager, 2-11
Notational Conventions, vi

object, 1-2, 3-3
object modules, 1-1
OJ, 3-3
options, 3-9
output file, 1-7, 2-12

PAR, 2-10
PARAMETERS
  actual, 2-10
  formal, 2-10
PAGELENGTH, 2-11
PAGEWIDTH, 2-11
PAGING/NOPAGING, 2-11
PI/NOPI, 2-11
PL, 2-11
PMTs, v, 1-1, B-1
PR/NOPR, 2-10
primary class, 3-2
PRINT, 1-7, 2-12, 3-9
PRINT/NOPRINT, 2-10
PRO/NPRO, 3-9
programmable secretaries, 1-1
prompt, 3-7, 3-16
PROMPT/NOPROMPT, 3-9
PUT, 1-3, 1-8, 3-4, 3-6
PW, 2-11

read, 1-8, 3-5
Related Publications, v
retrieved files, 3-12
RETURN, 1-8, 1-9, 3-4, 3-7
RUN, B-1

semi-colon, 2-2
Series III, v, B-1
shadow, 3-3
sign-off message, 1-9
SO, 3-3
software generation, 1-1, 1-5
software management, 1-1
source, 1-2, 3-2
source class, 3-2, 3-5
source contention, 1-2
source modules, 1-1, 3-2
submit file, 1-1, 1-5, 1-7, 2-2, 2-8, 2-12

SVCS, 1-1, 1-8, 3-1
SVCS access definition, 2-6, 2-7
SVCS command line, 1-8
SVCS commands, 1-8, 3-4, 3-9, A-1
SVCS files, 3-12
SVCS prompts, 1-5, 3-16, A-2
syntax
  MAKE, 2-8
  SVCS, 3-10
system unit, 3-2

TARGET, 2-10
target file, 1-6, 2-2
task lines, 1-6, 1-7, 2-2, 2-3
TIMEOUT/NOTIMEOUT, 3-9, 3-12
TO, 2-9
tracking changes, 1-1, 3-3

units, 3-1,3-2, 3-8

variants, 3-3, 3-8
variations, 1-2, 3-1, 3-3

WA, 3-8
winchester disk, 1-2, 2-11
WORK, 3-3, 3-8
WR, 3-6
WRITE, 1-2, 1-8, 3-4, 3-6
WRITEACCESS, 3-8

REQUEST FOR READER'S COMMENTS

Intel's Technical Publications Departments attempt to provide publications that meet the needs of all Intel proc users. This form lets you participate directly in the publication process. Your comments will help us correct improve our publications. Please take a few minutes to respond.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of publication. If you have any comments on the product that this publication describes, please contact your I representative. If you wish to order publications, contact the Intel Literature Department (see page ii of manual).

1. Please describe any errors you found in this publication (include page number).

2. Does the publication cover the information you expected or required? Please make suggestions improvement.

3. Is this the right type of publication for your needs? Is it at the right level? What other types of publicati are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this publication on a scale of 1 to 5 (5 being the best rating).

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____

CITY _____ STATE _____ ZIP CODE _____

(COUNTRY)

Please check here if you require a written reply. ☐

'E'D LIKE YOUR COMMENTS ...

ıis document is one of a series describing Intel products. Your comments on the back of this form will ɜlp us produce better manuals. Each reply will be carefully reviewed by the responsible person. All ›mments and suggestions become the property of Intel Corporation.

NO POSTAGE
NECESSARY
IF MAILED
IN U.S.A.

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1040 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
Attn: Technical Publications M/S 6-2000
3065 Bowers Avenue
Santa Clara, CA 95051

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95051 (408) **987-8080**

Printed in U.S.A.