DATAPOINT

DASL™

User's Guide

50807-02

October 16, 1984

Document No. 50807-02. 10/84

Preface

The DASL USER'S GUIDE is the first book to be produced in the new DATAPOINT documentation standard format. It is written using the Information Mapping writing method. The final masters for this book were produced on the DATAPOINT 9660 Laser Printer.

Please forward your comments about this document to:

DATEC Publications
DATAPOINT Corporation
9725 Datapoint Dr. MS T-72
San Antonio, Texas 78284

TABLE OF CONTENTS

5. EASL FILE SPECS AND OPTIONS

6. EASL RECOVERY MECHANISMS

7. INTRODUCTION TO RASL

8. REQUISITES FOR USING RASL

9. HOW TO USE RASL

14. CPUTIME: CPU TIMING PROGRAM

15. THE INCLUDE FILE D$INC

Chapter 1.
INTRODUCTION TO EASL

OVERVIEW

Description

This section (Chapters 1-6) of the DASL User's Guide is a reference guide for the editor EASL (Editor for Advanced Systems Languages).

Features of EASL

EASL was designed to be an easy to learn, convenient text editor for programmers of high level languages.

The features of EASL include the following:

- simplicity,
- extensive help screens,
- recovery mechanisms to prevent loss of work,
- ability to handle long lines (250 characters) and print files,
- capability to move quickly through long files,
- facility for easily modifying arbitrary parts of lines,
- provision of macros for "programming" repetitive tasks, and
- ability to work well with low-speed remote connections.

Invoking EASL features

Many of the EASL editing features can be invoked directly by using the function keys and the numeric keypad keys.

Additional features are invoked by entering the Command Key and the first letter of the name of one of the 11 EASL commands.

The basic editing features and the specific commands are discussed in Chapters 1-6.

ORGANIZATION AND USE OF THIS SECTION

Purpose of this section

Chapters 1-6 provide more information than the EASL help screens.

Different uses of this section

Different people will make different use of this section, depending on their preference for printed vs. on-line documentation.

You can learn to use EASL by relying

- primarily on Chapters 1-6 of this document,
- primarily on the on-line documentation, or
- on a combination of the two resources.

Using this section as a supplement

For people who prefer to use this section as a supplementary resource, the following chapters provide more detailed information than is included on-line:

- Chapter 2 - Beginning EASL
- Chapter 5 - EASL File Specifications and Options, and
- Chapter 6 - EASL Recovery Mechanisms.

In addition, at the beginning of Chapter 3 there is an expanded version of the on-line chart which shows the use of the numeric keypad and function keys.

Contents of section

The following table describes Chapters 1-6 of the DASL User's Guide.

THIS chapter...	DESCRIBES...
INTRODUCTION TO EASL	the features of EASL and the organization of this section.
BEGINNING EASL	• how to begin and end an EASL session and • how to access EASL help screens.
BASIC TEXT EDITING WITH EASL	how to insert, change, and format text using the numeric keypad and function keys.
EASL COMMANDS	each of the EASL commands for invoking special features.
EASL FILE SPECIFICATIONS AND OPTIONS	EASL files and options.
EASL RECOVERY MECHANISMS	how to recover from: • an aborted session or • failure of copy-back mechanism.

Chapter 2.
BEGINNING EASL

OVERVIEW

Introduction

This chapter provides the necessary information to help a new user begin to work with EASL. The user can decide whether to refer to the remaining chapters in this section or to access the help screens for further information.

Coming up

This chapter includes the following topics:

- hardware requirements for EASL,
- beginning and ending an EASL session, and
- accessing the EASL help screens.

Restriction on types of workstations

Because EASL uses the subwindowing capability of the RMS workstation interface, EASL will not run on 3600 or 6600 consoles or version 1 8200 workstations. EASL also will not run on 3800 or 5500 processors.

Two kinds of keyboards

EASL was intended for the word processing version of the keyboard, but it will also work on the data processing keyboard.

The following chart shows how three keys are marked differently on the two keyboards.

KEY	SYMBOL ON WP KEYBOARD	SYMBOL ON DP KEYBOARD
Command Key	▭	\ or '
Delete Key	→|←	DEL
Insert Key	|↔|	SP (shifted 0)

BEGINNING AND ENDING AN EASL SESSION

Introduction

The following pages briefly describe how to begin and end an EASL session.

Beginning an EASL session

To begin an EASL session, enter on the RMS command line

EASL <file name>

If a file of the name you enter does not exist, EASL asks if you wish to create a new file.

The default extension of the file is TEXT, and the default environment is blank. EASL places any new file it creates in the :W environment.

Other files and options

In addition to the source file, you may specify other files and options. See Chapter 5 for information.

Result of EASL command

After you enter the EASL command

- the screen clears,
- a welcome message appears in inverse video on the EASL command line on the bottom of the screen, and
- the first line of the file you are editing appears above the command line with the cursor positioned at the beginning of the line.

Ending an EASL session

To end an EASL session, saving the changes you have made, use the End Command.

To end an EASL session without saving changes, use the Quit Command.

To execute the End and Quit Commands

STEP	ACTION
1	Press the Command Key
2	Enter E (for End) or Q (for Quit)
3	Enter Y at the prompt
4	Press the ENTER Key

These commands are described in more detail in Chapter 4.

Recovery from an aborted session

EASL has a recovery mechanism to prevent loss of work due to an aborted session. For information see Chapter 6.

Introduction

The Help Command allows you to access on-line documentation describing

- basic text editing using the numeric keypad and function keys,
- each EASL command, and
- new features of EASL.

In this written guide, basic text editing is discussed in Chapter 3 and EASL commands are discussed in Chapter 4.

New features of EASL are documented on-line when they are implemented.

Beginning the Help Command

To begin the Help Command

STEP	ACTION
1	Press the Command Key
2	Enter H on the command line
The following prompt is displayed on the command line:	
HELP: enter a command letter (?CDEHKLMPQTW) for more information.	

Names of EASL commands

The EASL commands are:

- Clear,
- Detect,
- End,
- Help,
- Keyboard,
- Locate,
- Macro,
- Pop,
- Quit,
- Tab, and
- Window.

To list these names on the screen, press the Command Key and enter a ? or space character on the command line.

Accessing different kinds of information

TO access information on...	ENTER this character after the Help prompt...
basic text editing	H (for help)
specific EASL commands	C, D, E, K, L, M, P, Q, T, or W Note: Each letter is the first letter of a command.
new features of EASL	?

Accessing information on basic text editing

To access information on basic text editing, respond to the first Help prompt with the letter H and press the Enter Key. The following prompt appears on the command line:

```
HELP: enter H again for general help or C for chart
      of number pad:
```

If you respond with H again, general help information on text editing is displayed.

If you respond with C, the following prompt appears:

```
HELP: enter D to display chart or F to place it in
      your file:
```

If you enter D, the chart illustrating the number pad and function keys is displayed on the screen.

If you enter F, the chart is inserted below the cursor in the file you are editing. This allows you to print out the chart and attach it to your workstation for reference.

How help information is displayed

For some of the commands (Clear, End, Quit and Tab), the help information fits on just one line and is displayed on the command line itself.

Information for the other specific commands, new EASL features, and basic editing features is displayed on the full screen in place of the file you are editing.

The display scrolls automatically if it is longer than the screen.

Pausing or terminating display

When help information is being displayed on the screen, the command line displays the following information:

```
   HELP: use DSP Key to pause display or INT Key to
         terminate -->
```

If you use the DSP Key, the information stops scrolling. You may press any key on the keyboard to resume scrolling.

When all of the information has been displayed or when you have pressed the INT Key, the command line prompts:

```
   Now hit the NewLine (Enter) Key when ready to resume
   editing:
```

Pressing the Enter Key

- repaints the screen with the text you are editing and
- returns you to normal editing mode.

Chapter 3.
TEXT EDITING WITH EASL

OVERVIEW

Introduction

This chapter explains how to insert, change, and format text using the numeric keypad and the function keys.

Organization of chapter

The chart on page 7 of this chapter summarizes the uses of each of the numeric keypad and function keys.

The remainder of the chapter expands on the information provided in the chart.

Where to find information on commands

This chapter makes reference to several of the EASL commands, including

- Clear,
- Keyboard,
- Overstrike,
- Wordwrap,
- Pop, and
- Window.

If you are not familiar with these commands, you can find complete information in Chapter 4.

Subwindowing capability

EASL is a single window editor. However, EASL has commands which allow you to

- reduce the size of the window,
- replace part of your window with a "frozen" picture so that you can see part of the file while you are editing another part, and
- change the horizontal offset of the window.

For more information, see the section on the Window Command in Chapter 4.

Definitions

The following terms are used throughout the rest of this guide to describe your file and how you see it on the EASL screen.

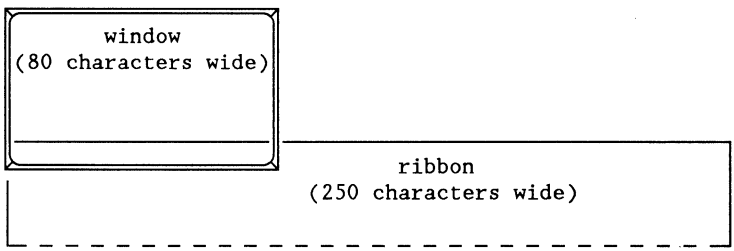
The ribbon is a 250 character wide virtual representation of the file with the modifications made in the current edit session. It is space filled allowing you to move the cursor anywhere on it.

The window is an area of the workstation screen which looks onto part of the ribbon. The window is constrained to stay on the ribbon horizontally.

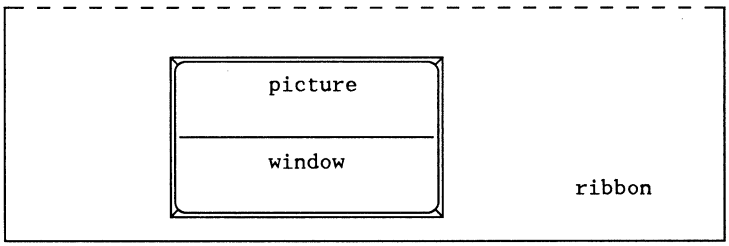
The picture is a frozen view that was once in part of a window.

Diagrams

The diagram below shows how the window appears at the beginning of an EASL session.



The following diagram shows a window which has been reduced in side and offset horizontally.



Coming up

The contents of this chapter include:

- chart of numeric keypad and function keys,
- inserting text,
- editing special text records,
- changing the position of the cursor,
- deleting text,
- copying text,
- popping text from the stacks,
- moving text,
- formatting text, and
- changing characters in place.

THE NUMERIC KEYPAD AND FUNCTION KEYS

Introduction

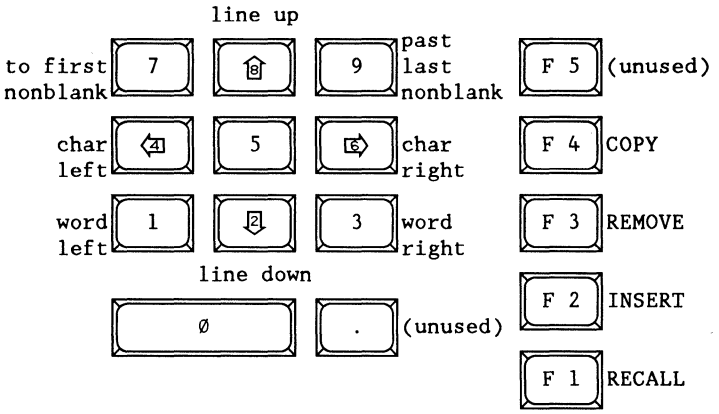
The chart presented on these pages summarizes the uses of each of the numeric keypad and function keys.

Accessing chart on-line

The Help Command allows you to access on-line a slightly abbreviated version of this chart. See the pages in Chapter 2 called Accessing EASL Help Screens.

Chart

Use the following chart as an introduction or as a reference to the use of the numeric keypad and function keys.



Use the number pad 5 key followed by

- number pad 2, 4, 6, or 8 key for express cursor movements (6 lines or 20 columns),
- number pad 1, 3, 7, or 9 key to move cursor to corners of window,
- Tab Key to move cursor to previous tab stop,
- Command Key to put out a ' or \ character (shifted or unshifted), or
- Insert or Delete Key to put out a 0177 character.

Use the DSP and KBD Keys to slide the window up or down the ribbon.

Use the ENTER Key alone to insert a blank line below the cursor.

Press the ENTER Key while holding down

- COPY to split line at cursor (cursor not moved),
- REMOVE to weld line to one above,
- INSERT to split line at cursor (cursor moved to new line), or
- RECALL to rejustify line at word wrap column.

REMOVE and COPY work with all but the 5, 8, and 0 keys to delete or copy portions of text and push them onto the stack. REMOVE and COPY work with the 0 key to pop one line from the line stack.

RECALL works identically with the 1, 3, 4, 6, 7, 9 and 0 keys to pop one item from the character stack.
RECALL/ ⏎ pops an entire line from the stack.

INSERT is used with

- the 4 key to decrement the ASCII value of a character,
- the 6 key to increment the ASCII value of a character, or
- the 5 key to invert the case of an alphabetic character.

INSERTING TEXT

Introduction

Inserting text in EASL works in two different ways, depending on whether you are in overstrike or insert mode.

Every EASL session begins in insert mode. You can reverse the mode by performing the Keyboard command.

Inserting text in insert mode

To insert characters while in insert mode, enter them with the cursor in the desired place.

If there are nonblanks at or to the right of the cursor, they are shifted to the right one position for each new character.

The Insert and Delete Keys are not differentiated in insert mode. Both keys delete the character under the cursor and move the rest of the line to the left.

Inserting text in overstrike mode

In overstrike mode, each new character which is entered replaces any character or nonblank under the cursor. The cursor is then moved to the right one position.

The DELETE Key works as in insert mode, but the Insert Key inserts a blank and shifts the rest of the line to the right.

For more detailed information on the overstrike mode, see the pages titled KEYBOARD COMMAND in Chapter 4.

Side-scrolling during insertion

If you insert a character at the rightmost column of the window, then all of the lines in the window slide to the left one position.

Special text records

EASL allows you to display special text records and edit them like "normal" records. Special text record lines are not included in the line number count.

Special text records are marked by a small solid square (the octal 0177 character) appearing in the first column of the line.

Entering a special text record indicator

To enter a square (0177 character), press the number pad 5 key followed by the insert/delete key.

If you enter a square in column 1, that line will be written out as a special text record. Squares in other columns have no special meaning in EASL.

CHANGING THE POSITION OF THE CURSOR

Introduction

EASL provides the capability for moving the cursor quickly and easily to any place in a file.

The following pages expand on the information provided on the chart at the beginning of the chapter.

Window sliding during cursor movement

The following chart shows how the window slides when the position of the cursor is changed.

IF the cursor bumps into...	THEN the window slides in the appropriate direction
the left or right edge of the window	• the specified distance or • until the edge of the window is at the edge of the ribbon.
the top or bottom of the window	• the specified distance or • as many lines as exist in the direction of travel.

Moving the cursor one space or one line

The keys on the numeric keypad marked with arrows (2, 4, 6, and 8) move the cursor one space or one line at a time in the direction of the arrow.

Moving the cursor to the beginning or end of line

The number pad 7 key moves the cursor to the first nonblank of the current line.

The 9 key moves the cursor to the second column after the last nonblank on the line.

If the entire line is blank, both the 7 and 9 keys move the cursor to the leftmost column of the line.

Moving cursor by words

All word operations treat the characters . , and ; as spaces.

The 1 and 3 number pad keys move the cursor to the left or right, respectively, until a nonspace following a space is found.

If there are only spaces to the left of the cursor when 1 is pressed, the cursor moves to the left edge of the file.

If there are only spaces to the right when 3 is pressed, the cursor moves one space to the right.

Using express cursor movements

The number pad 5 key is used in combination with other keys for express cursor movements.

IF you press 5 and then...	THEN the cursor moves...
8	up 6 lines.
2	down 6 lines.
4	to the left 20 spaces.
6	to the right 20 spaces.

Moving cursor to window corners

The number pad 5 key is used in combination with other keys to move the cursor to any corner of the portion of the window which is on the screen. The window does not slide during these movements.

IF you press 5 and then...	THEN the cursor moves to this corner of the window...
7	upper left.
9	upper right.
1	lower left.
3	lower right.

Sliding the window using KBD and DSP

Pressing the KBD or DSP Key causes the window to slide up or down the ribbon, respectively, without changing the position of the cursor in the window.

Sliding continues as long as

- either the KBD or DSP Key is depressed or
- the first or last line of the ribbon appears in the window.

DELETING TEXT

Introduction

These pages describe the different methods for deleting text and the way in which stacks are used to save deletions.

Kinds of deletions

There are several different ways in EASL to perform deletions.

You may use

- the Backspace or Delete Key to delete a single character or
- the REMOVE function key together with the number pad keys to delete a portion of a line or a whole line.

Deleting characters with backspace key

To delete the character to the left of the cursor, press the Backspace Key (REMOVE/ ⌫ does the same thing). The rest of the line from the cursor to the right shifts to the left one position.

Nothing happens if you are at the left edge of the line when you press the Backspace Key.

Deleting characters with Delete Key

To delete the character under the cursor, press the Delete Key (REMOVE/ ⌦ does the same thing).

The rest of the line to the right of the cursor shifts to the left to fill the gap. The cursor is not moved by the Delete Key.

Deleting with the REMOVE function key

The REMOVE (F3) function key is used with all of the number pad keys except 5, 8, and 0 to delete different portions of text.

The following chart describes how the REMOVE function works.

IF you hold down the REMOVE key and press...	THEN this portion of text is deleted...	AND the deletion is stored on the...
• 1 (word left) • 4 (char left) • 7 (to first nonblank)	• any char (characters) the cursor moves over	character stack.
• 3 (word right) • 6 (char right) • 9 (past last nonblank)	• any char that gets pulled into the cursor	
2 (line down)	the entire line the cursor was on	line stack.

Purpose for stacking deletions

EASL stacks deletions so that the user can

- "undo" unintended deletions or
- move text from one place in a file to another.

See POPPING TEXT FROM THE STACKS on starting on page 3-22 and MOVING TEXT starting on page 3-24.

How deletions are stacked

EASL uses a character stack and a line stack to save all deletions of text.

Description of the character stack

The character stack saves all deletions of

- characters,
- words, and
- portions of lines.

The character stack is 250 characters deep. When the stack is filled, the oldest items are discarded. No indication is given when characters drop off the stack.

In addition to storing the character codes, the stack stores tags indicating what kind of delete action caused the characters to be pushed on the stack. Therefore, only one "undo" function serves to undo all kinds of deletions.

Description of the line stack

The line stack is used for storing all deletions of complete lines.

The line stack is stored in a work file which can contain up to 65,535 lines.

The stack work file is preserved between EASL sessions and can therefore be used for moving lines from one file to another. When you start an EASL session, the command line states how many lines are currently on the line stack.

COPYING TEXT

Introduction

EASL provides a way to copy different portions of text and store the copies on the character or line stacks.

Copying with the COPY function key

The COPY function key is used with all of the number pad keys except 5, 8, and 0 to copy different portions of text.

The COPY function works in the same way as the REMOVE function, except that the characters moved over by the cursor are not deleted. The following chart describes how the COPY function works.

Copying

IF you hold down the COPY key and press...	THEN this portion of text is copied...	AND the copy is stored on the...
• 1 (word left) • 4 (char left) • 7 (to first nonblank)	• any chars which the cursor would move over and • the char which ends up beneath the cursor	character stack.
• 3 (word right) • 6 (char right) • 9 (past last nonblank)	• the char beneath the cursor and • any char which the cursor would move over	
2 (line down)	the entire line which the cursor was on	line stack.

How copies are stacked

EASL stacks copies of text as if a deletion were performed.

For a description of the character and line stacks, see the preceding pages titled DELETING TEXT.

POPPING TEXT FROM THE STACKS

Introduction

Once you have deleted or copied text onto the character or line stacks, you can then place the text back into the file by using either the function keys or the Pop Command.

Popping text with function keys

The following chart shows how the function and number pad keys are used to pop text from the stacks.

IF you want to pop...	THEN either...
the most recent deletion or copy from the character stack	• hold down RECALL and press 1,3,4,6,7,9 or 0, or • press 0 by itself.
one line from the line stack	• hold down RECALL and press 2 or • hold down either COPY or REMOVE and press 0.

Using the Clear and Pop Commands

The CLEAR Command clears both the line stack and the character stack.

The Pop Command pops all the lines on the line stack.

For more information on using these commands, see the pages called CLEAR COMMAND and POP COMMAND in Chapter 4.

Introduction

By using the procedures for deleting, copying, and recalling text in combination, you can move portions of text either

- within a file or
- from one file to another.

Moving portions of lines

To move portions of lines within a file use the following procedure:

STEP	ACTION
1	Delete or copy the text
2	Place the cursor in the position to the left of where you want the text to be deposited
3	Pop the text from the character stack

Moving lines within a file

To move lines within a file, use the following procedure:

STEP	ACTION
1	Perform the Clear Command to clear the stack
2	Delete or copy the desired lines
3	Position the cursor on the line above which you want the line to be deposited
4	Perform the Pop Command to pop the entire line stack in the new location

Moving lines across files

You can move lines from one file to another since the line stack is preserved between EASL sessions.

Perform the steps listed above, except after deleting or copying the desired lines, you must exit the EASL session and enter the new session before popping the stack.

Introduction

EASL has several formatting functions which make it easy to create the indentation used to show the structure of high level language programs.

These functions are performed by

- pressing the ENTER Key or
- using the ENTER Key in combination with the function keys.

Using the ENTER Key alone

To insert a blank line below the cursored line with automatic indentation, press the ENTER Key.

Result: The cursor moves to the newly created blank line. The horizontal position of the cursor is under the first nonblank of the previous line.

If the previous line is blank, the line previous to that is tried. If that line is also blank, then the horizontal position of the cursor is the first column of the line.

Using ENTER with the function keys

The following chart describes the different formatting effects of using the ENTER Key in combination with the function keys.

IF you press the ENTER Key while holding down this function key...	THEN...
RECALL	line rejustification is performed around the wrap column. Note: The word wrap mode must be on or nothing happens.
INSERT	the line that the cursor is on is split at the cursor. The part of the line to the left of the cursor remains on the current line. The part of the line from under the cursor to the right is inserted below the first nonblank of the current line. The cursor also moves to the new line. The horizontal position of the cursor is below the first nonblank of the previous line.

IF you press the ENTER Key while holding down this function key...	THEN...
REMOVE	the line that the cursor is on is welded onto the previous line. The horizontal position of the cursor has no effect. The first nonblank of the cursored line is placed one space after the last nonblank of the previous line. An extra space is inserted if the last nonblank is a . ? or !
COPY	the same result occurs as with INSERT except the cursor does not move. This is useful for inserting blank lines at the beginning of the file.

CHANGING THE VALUE OF A CHARACTER

Introduction

There are three ways to change the value of individual characters:

- incrementing the ASCII value,
- decrementing the ASCII value, and
- inverting the case.

Changing the ASCII value can be useful in repetitive macros.

Changing the value

Move the cursor to the character you want to change and...

IF you want to...	THEN hold down the INSERT function key and press the number pad key...
increment the ASCII value of a character	6. The sequence wraps from 127 to 0.
decrement the ASCII value	4. The sequence wraps from 0 to 127.
invert the case	5. After the function is performed, the cursor moves one space to the right.

ASCII functions with digits

If you use the increment or decrement function with a digit, a decimal carry or borrow is performed.

Examples:

- If you increment the digit 9 in 169, the result is 170.
- If you decrement the last 0 digit in 200, the result is 199.

Chapter 4.
EASL COMMANDS

OVERVIEW

Introduction

EASL provides 11 commands which allow you to perform various functions.

You may also write up to ten macros which are invoked by the commands 0 through 9.

Kinds of commands

The following chart briefly describes each of the EASL commands.

COMMAND NAME	DESCRIPTION
End	Ends EASL, updating source file with changes
Quit	Quits EASL, not updating source file
Help	Accesses on-line documentation for EASL (See Chapter 2)
Locate	Locates a line in the file by number or content
Clear	Clears the line and character stacks
Pop	Pops the entire line stack
Macro	"Programs" repetitive editing tasks
Detect	Detects a string on a line for aborting macro playback (especially loops)
Tab	Sets tab stops
Keyboard	Changes five different keystroke modes: keystroke click, shift key inversion, overstrike, video, and word wrap
Window	• Sets horizontal offset of screen and • changes the "working" part of the file to a smaller vertical or horizontal window.

Listing the names of EASL commands

To list the names of all of the EASL commands on the screen

STEP	ACTION
1	Press the Command Key
2	Enter a ? or space character
3	Press the ENTER Key
EASL lists the command names and then awaits the entry of the command letter.	

Leaving command mode

To begin any command, press the Command Key and the cursor moves to the command line.

If you change your mind and do not want to perform a command, you can press

- the Command Key before you enter a command or
- the ENTER Key before you have responded to a command prompt.

The cursor then leaves the command line and EASL returns to the normal editing mode.

Coming up

The rest of this chapter provides specific information about each of the EASL commands.

END COMMAND

Description

The End Command allows you to

- update your source file with changes made in the current editing session and
- leave EASL.

Performing the End Command

To perform the End Command

STEP	ACTION
1	Press the Command Key
2	Enter E on the command line
3	Enter Y at the prompt END?
4	Press the ENTER Key

Results of the End Command

When the End Command is executed, EASL first checks to see if you have made any changes in the file during the current editing session.

IF changes...	THEN this message appears on the command line...	AND the WORK/E$ file...
have been made	Work file complete and is being copied back to source file.	is completed properly and copied back to source file.
have not been made	There were no changes; work file NOT completed.	does not have a complete copy of the file and is of no use.

QUIT COMMAND

Description

The Quit Command allows you to quit EASL without updating the source file.

Performing the Quit Command

To perform the Quit Command

STEP	ACTION
1	Press the Command Key
2	Enter Q on the command line
3	Enter Y at the prompt QUIT?
4	Press the ENTER Key

Results of the Quit Command

When the Quit Command is performed, EASL first checks to see it you have made any changes in the file during the current editing session.

If no changes were made, then EASL displays the message:

```
There were no changes; work file NOT completed.
```

If changes were made, then EASL displays this warning and prompt:

```
Modifications were made.  Still want to quit?
```

If you enter a Y at the prompt, then EASL writes out a complete WORK/E$ file, but does not update the source file with changes made during the current editing session.

LOCATE COMMAND

Description

The Locate Command allows you to locate a specific line or a string in a file.

You can locate

- a specific line by line number,
- a line which is a specified number of lines toward the end (or beginning) of the file from the current cursor position,
- a string which occurs after (or before) the current cursor position,
- a string which occurs at the beginning of a line, or
- an entirely blank line.

The Locate Command also allows you to

- position the cursor to the place where the previous Locate was performed or
- display the current line number.

Effect of Locate Command

After a Locate, the cursor is positioned on the specified string or line in the file.

The window is repainted with the cursor in the same relative vertical position. The window is offset horizontally as necessary to accommodate the cursor.

Note:

You can reduce the window repaint time by reducing the size of the window. See the pages titled WINDOW COMMAND.

Beginning the Locate Command

To begin the Locate Command

STEP	ACTION
1	Press the Command Key
2	Enter L on the command line
The prompt LOCATE: is displayed on the command line.	

Specifications for line Locate

This table shows the specifications for different kinds of line Locates. The optional character t (or T) specifies that the located line will appear at the top of the window.

TO locate...	ENTER this after the LOCATE: prompt...
a specific line by line number	[t]<number>
a line which is a specific number of lines toward the beginning of the file from the current cursor position	[t]-<number>
a line which is a specific number of lines toward the end of the file from the current cursor position	[t]+<number>

Notes:

- Special text records are not counted in the line numbering.
- If the file contains fewer lines than the specified line number, then the cursor moves to the last line in the file.
- Line numbers are modulo 65,536, so line 65,536 appears to be line 0, line 65,537 appears to be line 1, and so on. This can cause confusion when locating by line number in extremely large files.

Cursor movement in line Locate

When a line Locate is performed, the cursor is placed at the first nonblank of the line.

Normally, the vertical position of the cursor does not change. However, if you enter a t at the beginning of the specification, the cursor appears at the top of the window.

Displaying current line number

To find out the number of the line where the cursor is positioned, enter ? after the LOCATE: prompt.

The line number is displayed on the command line, and the cursor returns to its original position.

Specifications for string Locate

This table shows the specifications for different kinds of string Locates.

Specifications inside brackets are optional. The character t (or T) specifies that the located string will appear at the top of the window.

The character - causes the string search to travel toward the beginning of the file.

The character + causes the string search to travel towards the end of the file.

No - or + implies + in a string Locate.

Specifications for string Locate,

TO Locate...	ENTER this after the LOCATE: prompt...
the first match of a string	[t][- or +] <string>
the first match of a string which is preceded by nothing or only spaces on the line	[t][- or +]$<string>
an entirely blank line	[t][- or +]$
the next match of the string specified in the previous Locate or Detect	[t][- or +]

How strings are stored

The space or $ character is saved along with the specified string in a "previous string" storage area each time a nonnull string is given.

The Locate and Detect commands share the "previous string" storage.

Cursor movement in string Locates

When a string Locate is performed, the cursor is placed at the beginning of the string.

Normally, the vertical position of the cursor does not change. However, if you enter a t at the beginning of the specification, the cursor appears at the top of the screen.

If the specified string cannot be located, the message "-- STRING NOT FOUND --" is displayed on the command line and the cursor returns to where it was in the file prior to the Locate Command.

Moving back to previous Locate positions

To position the cursor to the place where the previous line or string Locate was performed, enter a period after the LOCATE: prompt.

Only the last five locations are saved. When all have been popped, the . Locate simply does nothing.

Examples of LOCATE specifications

The following examples illustrate different specifications for the Locate command.

TO Locate...	ENTER this after LOCATE: prompt...
line number 53	53
a line which is 10 lines toward the beginning of the file from the current cursor position	-10
a line which is 12 lines toward the end of the file from the current cursor position	+12
line number 53 and have it appear at the top of the window	t53
the string XYZ toward the end of the file	XYZ or + XYZ
the next occurrence of the string specified in the previous LOCATE	+
the string XYZ at the beginning of a line toward the beginning of the file	- XYZ
an entirely blank line toward the beginning of the file	-$
the position where the previous Locate was done	

CLEAR COMMAND

Description

The Clear Command clears both the line stack and the character stack at the same time.

When to use

The Clear Command is useful if you want to clear the line stack before moving a group of lines.

Performing the Clear Command

To perform the Clear Command

STEP	ACTION
1	Press the Command Key
2	Enter C on the command line
3	Enter Y at the prompt CLEAR STACKS?
4	Press the ENTER Key

POP COMMAND

Description

The Pop Command allows you to unstack all the lines on the line stack.

This command also tells you the current depth of the stack.

Performing the Pop Command

To execute the Pop Command

STEP	ACTION
1	Press the Command Key
2	Enter P on the command line
3	Enter Y at the prompt POP all <n> lines from the stack?
4	Press the ENTER Key

Suggestion

For the following reasons, it is a good idea to use the Clear Command regularly to keep the stack cleared of lines that you do not need to save.

- When you use the Pop Command, there is no way to stop the unstacking process short of aborting the session.
- The stack is copied into the LOG file at the beginning of every session. See Chapter 5 for information on the LOG file.

MACRO COMMAND

Description

The Macro Command allows you to "program" repetitive editing tasks.

You can define up to ten different macros, which can be saved for use in later EASL sessions.

Definition of macro

A macro in EASL is a recording of up to 250 keystrokes.

Beginning the Macro Command

To begin the Macro Command

STEP	ACTION
1	Press the Command Key
2	Enter M on the command line

Result: The following prompt appears on the command line:

```
MACRO DEF (0..9, R or W):
```

STEP	ACTION
3	Enter one of the following: • a digit to begin the macro definition, • W to write a macro to a disk, or • R to read a macro from a disk.

Beginning a macro definition

To begin a macro definition, respond to the Macro prompt with a digit from 0 through 9. The digit which you enter appears at the left of the command line (in normal video) to indicate that recording of the macro is in progress.

Example: If you enter the digit 5, the following message appears on the command line.

```
-- starting macro 5 definition --
```

Defining a macro

While the macro definition is in progress, you may enter a sequence of up to 250 keystrokes, including function keys and commands.

The only two keys which are not allowed are the DSP and KBD Keys. These keys are disabled during macro definition.

You can invoke another macro during definition. You may also invoke the same macro which you are defining, but this causes the definition to be terminated (the indicator goes out).

If you overflow the 250 character limit, the definition is nullified and the definition mode is terminated.

If there is a syntax error in a command, that command is erased from the macro definition.

Currently, there is no way to edit a macro except to reenter the complete definition.

Example of a macro definition

Suppose that you want to replace the word "good" with the word "fine" in a file.

The following chart shows how you can create a macro (let's say macro 5) to accomplish the task.

To perform this step...	ENTER these keystrokes during macro definition
locate the word "good"	Command Key L + good
delete the word "good"	Delete Key four times
insert the word "fine"	fine
invoke the macro which is being defined	Command Key 5
This macro will loop or call itself until the LOCATE Command fails.	

Terminating a macro definition

To terminate a macro definition in a nonlooping macro,

STEP	ACTION
1	Press the Command Key
2	Enter M on the command line
3	Enter Y or . after the prompt MACRO STOP?

The command line now reads

```
MACRO STOP?  -- macro <n> definition completed --
```

If you type in any character other than y or ., the command line reads

```
<n>  MACRO STOP?  * MACRO DEFINITION CONTINUES *
```

Displaying macro contents

Currently, there is no way to display the contents of a macro. You must remember the definitions.

Invoking a macro

Before invoking a macro, position the cursor at the desired location in the file.

To invoke a macro

STEP	ACTION
1	Press the Command Key
2	Enter a digit from 0 through 9 on the command line

When a macro is invoked, its contents are pushed onto a 1000 character playback stack. It is possible to make this stack overflow by having several macros looping on each other, but this does not usually happen.

Aborting a macro

To abort a macro which is in progress, press the KBD key or any other key. This causes the remainder of the stack to be discarded when the end of the top macro definition is reached.

Macros also terminate if there is an error condition, such as

- running off the beginning or end of the file or
- having the Detect or Locate Command fail to meet a specified condition.

Writing a macro to a disk

New macros defined in an editing session are not saved on a disk unless you enter a specific sequence of commands.

To write a macro definition to a disk

STEP	ACTION
1	Enter the M Command
2	Enter W (for write) after the Macro prompt
3	Enter a digit from 0 through 9 to indicate which macro definition to save

The message "-- macro <n> definition written out --" appears on the command line.

Pressing the ENTER Key at any point in this sequence aborts the Macro Command.

All macros are written to the storage file specified in the EASL command or to the default file MACNSTACK/E$.

Automatic reading of macro definitions

When an EASL session is started, all ten macros are automatically read from the macro storage file. The default storage file is MACNSTACK/E$:W. If the storage file does not exist when the EASL session is started, the file is created with all entries null.

Reading a specific macro definition from disk

Sometimes you might want to read a macro definition from a disk if you have created a different definition for that macro during the current editing session.

To read a macro definition from a disk

STEP	ACTION
1	Enter the M Command
2	Enter R (for read) after the Macro prompt
3	Enter a digit from 0 through 9 to indicate which macro definition to read

The message "-- macro <n> definition read in --" appears on the command line.

Pressing the ENTER Key at any point in this sequence aborts the Macro Command.

The definition read from the disk replaces any definition which you may have created in the current editing session.

Description

The Detect Command determines the presence or absence of a specified string on the current line.

The Detect Command is usually used in macros. It causes looping macros to terminate when the specified condition of string presence or absence is met. For more information on the use of the Detect Command in macros, see the pages titled MACRO COMMAND in this chapter.

Beginning the Detect Command

To begin the Detect Command

STEP	ACTION
1	Press the Command Key
2	Enter D on the command line

The prompt DETECT: is displayed on the command line.

Specifications for Detect Command

The following table shows the different kinds of specifications for the Detect Command.

Specifications inside brackets are optional.

The character t (or T) specifies that the detection is for the presence of a string.

The characters - or + specify that the detection scan is to the left or right of the cursor, respectively.

No - or + implies + in a Detect.

TO detect the presence or absence of...	ENTER this after the DETECT: prompt...
a string on the current line	[t][- or +] <string>
a string at the start of the current line	[t][- or +]$ <string>
the string specified in the previous Locate or Detect Command	[t][- or +]
an entirely blank line	[t][- or +]$

Results of Detect Command

The Detect Command does not affect the position of the cursor in the file.

This table shows the possible results of a Detect Command.

IF you are detecting..	AND the string is..	THEN...
the presence of a string [t]	found	a looping macro continues.
	not found	• a looping macro terminates and • the message -- STRING NOT FOUND -- appears on the command line.
the absence of a string	not found	a looping macro continues.
	found	• a looping macro terminates and • the message -- STRING FOUND -- appears on the command line.

Examples of Detect specifications

The following examples illustrate different specifications and results for the Detect Command.

TO detect...	ENTER this after the DETECT: prompt	RESULT...
the presence of the string XYZ to the right of the cursor	t+ XYZ or t XYZ	A looping macro terminates if XYZ is not present.
the absence of the string XYZ with nothing but zero or more blanks before it on the line	-$ XYZ	A looping macro terminates if XYZ is present.
the presence of an entirely blank line	t$	A looping macro terminates if the current line is not blank.

Description

The Tab Command allows you to set tab stops at any column.

Description of default tabs

At the beginning of an EASL session, EASL initializes the tab stops to every third column, starting in column 1.

Once you change the tab stops, there is no way to reset to the default tabs, except by exiting and reentering EASL.

Setting new tab stops

To set new tab stops

- enter in your file a line of text which has a nonblank at each column where you want a tab stop to be set and
- perform the Tab Command:

STEP	ACTION
1	Press the Command Key
2	Enter T on the command line
3	Enter Y at the prompt SET TABS?
4	Press the ENTER Key

Results of Tab Command

The results of the Tab Command are to

- erase any existing tab stops,
- set the new tabs to the columns indicated, and
- delete the line of text which you created to set the tabs, pushing it onto the line stack.

Introduction

The Keyboard Command includes five subcommands which allow you to reverse the state of a particular mode.

Kinds of subcommands

The following table lists the Keyboard subcommands and the modes which they control.

THIS subcommand...	REVERSES this mode...
Click	keystroke clicking.
Invertcaps	shift key inversion.
Overstrike	overstrike.
Reversevideo	reverse video.
Wordwrap	word wrap.

Description of mode states

Each mode has two states, which can be thought of as off or on.

If a particular mode is off, then the appropriate Keyboard subcommand turns the mode on and vice versa. An exception to this rule is the word wrap mode, which is described later.

At the beginning of an EASL session, all of the modes are off.

Performing a Keyboard Command

To perform a Keyboard Command

STEP	ACTION
1	Press the Command Key
2	Enter K on the command line
3	Select a subcommand by entering at the prompt either C, I, O, R, W [column number]
4	Press the ENTER Key

Description of keystroke clicking mode

In keystroke clicking mode, a click sounds with every keystroke.

Description of shift key inversion mode

In shift key inversion mode, keystrokes result in uppercase letters. To get lowercase letters, you must hold down the Shift Key.

Description of overstrike mode

In overstrike mode, a newly entered character replaces any character under the cursor and the cursor is moved to the right one position.

Each overstruck character is pushed onto the character stack. The "undo" key (number pad 0)

- moves the cursor one position to the left and
- pops a character from the stack, which replaces any character under the cursor.

Overstrikes caused by "undo" cannot be undone.

The Insert Key inserts a blank and shifts the rest of the line to the right.

Description of reverse video mode

In reverse video mode, the entire screen appears in reverse video.

Description of word wrap mode

In word wrap mode, a wrap margin is set at the column number specified in the Wordwrap subcommand. Column numbering starts with 1 at the beginning of the line.

If no column number is specified, the word wrap mode is forced off.

The columns beyond the wrap margin are considered to be the "hot zone".

The following are allowed in the hot zone:

- spaces,
- cursor movements,
- characters which are popped from the character stack, and
- characters that are pushed over due to insertion.

Rejustifying a line at wrap column

To rejustify a line such that no characters remain in the hot zone, hold down the RECALL Key and press the ENTER Key.

Welding of lines

The line which the cursor is on is welded to the previous line if a backspace is performed which

- leaves the cursor line blank and
- occurs when the line above contains only blanks from the cursor position to the left.

Definitions

The following terms are used in describing the effects of the Window Command.

The ribbon is a 250 character wide virtual representation of the file with the modifications made in the current edit session. It is space filled so you can move the cursor anywhere on it.

The window is an area of the workstation screen which looks onto part of the ribbon. The window stays on the ribbon horizontally. However, the window can only stay on one vertical line of the ribbon.

A picture is the part of the screen that is frozen when a window is formed.

Description of Window Command

The Window Command allows you to

- create a smaller active window,
- cause one or more portions of the file to become frozen pictures on the screen, and
- change the horizontal offset of the window.

Beginning the Window Command

To begin the Window Command

STEP	ACTION
1	Press the Command Key
2	Enter W on the command line
The prompt WINDOW (* for restore): appears on the command line.	

Reducing window size and taking pictures

At the beginning of an EASL session, the window is the size of the full screen.

The following chart shows how you can reduce either the height or width of the window. At the same time a portion of the file becomes a picture on the screen.

TO cause the active window to be...	ENTER after the Window prompt...	RESULTS
from the cursored line down	space	• A horizontal separator line is inserted at the cursored line, causing the lines from the cursor down to be scrolled down one line and • the lines above the separator become a picture.
from the cursored column to the right	| (vertical bar)	• A vertical separator line is inserted at the cursored column, causing the lines from the cursor to the right to be scrolled to the right one column and • the lines to the left of the separator to become a picture.

Restoring the window size

To restore the window to its original height and width, enter * after the Window prompt.

Taking multiple pictures

By repeating the Window Command with the cursor at different locations, you can create a number of different frozen pictures on the workstation screen at once.

Advantages of the Window Command

One advantage of reducing the window size is that small windows reduce window redisplay time after a Locate Command. This is useful when doing many Locates.

Freezing portions of the file allows part of the file to be viewed while another part is being accessed.

Freezing a portion of the file vertically is useful when dealing with lines longer than the screen width.

Changing the horizontal offset

At the beginning of an EASL session, the first column of the file is at the left edge of the screen. EASL starts with an offset of 1.

To change the horizontal offset of the window, respond to the Window prompt with a number between 1 and (250 - window width + 1).

This causes the file to be offset horizontally such that the specified column number in each file line appears at the left edge of the window. When the window is offset from column 1, the mode indicator "<" appears at the left edge of the command line.

Chapter 5.
EASL FILE SPECS AND OPTIONS

OVERVIEW

Introduction

When you invoke EASL from the RMS command line, you may specify

- names for any of the five files which EASL uses and
- four options.

Syntax

The standard RMS command line syntax is used to specify EASL files and options.

Example:

```
EASL POWER LOG=SAVE; SCAN, TEST
     ↑           ↑     ↑  ↑
name of      name of      options
IN file      LOG file
```

Coming up

This chapter provides information about the use and purpose of each file and option.

Description

EASL deals with five different files during every editing session.

If you do not specify a name for any of these files on the RMS command line, then EASL specifies a default file name. EASL also specifies default extensions and environments if they are not provided.

The following chart shows the default specifications and describes how each file is used.

The files are listed in the order in which they must be listed on the command line.

SYMBOLIC FIELD NAME	DEFAULT SPECIFICATION	DESCRIPTION
IN	.?./TEXT	Source text file
MACSTACK	MACNSTACK/E$	Non text file which stores macro definitions and the line stack
LOG	RECOVERYLOG/E$:W	Non text file which is used for recovering from an aborted session This files stores • macro definitions and line stack as of the beginning of the session, • all input during session, and • output from invoked macros.
WORK	WORK/E$:W	Text file which stores results of edit session before copying back to the IN file
RWORK	RWORK/E$:W	Non text file which stores lines that disappear off the top of the screen if they do not all fit in memory

Additional information about the IN file

If you do not have WRITE access to the IN file, the SCAN option is automatically invoked. This option is described on the following page.

You cannot edit a file which has an end-of-file pointer that does not point to a $LEOF character in the file. If your file has this error, you will have to run a utility program to fix the file before you can edit it with EASL.

Additional information about the WORK file

The WORK file is written out whenever you have made changes during an edit session and then enter the End or Quit Command.

In case something happens to the source file, you have a complete backup in the WORK file.

However, if you End or Quit without having changed anything, the WORK file does not have a complete copy of the source file and is of no use.

EASL OPTIONS

Introduction

There are four options in EASL which you may specify on the RMS command line.

Description

The following chart describes the four EASL options. The TEST and RASL options are used for certification and development of the EASL Program.

OPTION	DESCRIPTION
HELP	Displays on the screen a brief summary of the info on files and options.
SCAN or S	Forces the source file to be opened in read-only mode. This option is automatically assumed if you only have read access to the source file. You are still allowed to make changes in the file, but they will only alter the WORK file, not the source file.
TEST	Forces recovery if a LOG file is found and saves the LOG file for use as a test script.
RASL	Invokes the RASL debugger resident that is built into EASL. The pipe name is EASLRASLPIPE. The source code for EASL is required to make any use of this option.

Chapter 6.
EASL RECOVERY MECHANISMS

OVERVIEW

Introduction

EASL has recovery mechanisms to prevent loss of work due to

- an aborted EASL session, or
- failure of the copy-back from the WORK file to the source file.

Coming up

This chapter discusses

- recovery from an aborted EASL session, and
- recovery when copy-back fails.

RECOVERY FROM AN ABORTED EASL SESSION

Introduction

Whenever an EASL session is aborted, EASL automatically saves the work and gives the user the option of recovering the aborted file.

Function of the LOG file

The LOG file is used to recover from an aborted EASL session.

Whenever you use EASL, the LOG file keeps track of everything you are doing.

The LOG file stores

- macro definitions and the line stack as of the beginning of the session,
- all input during the session, and
- output from invoked macros.

The LOG file is updated

- every 256 keystrokes or
- whenever the keyboard is idle for more than five seconds.

Role of the LOG file in recovery

When you start an EASL session, EASL searches for the specified LOG file or the default file RECOVERYLOG/E$:W.

If EASL cannot find the file, it creates a new one.

If EASL finds the file, it searches for a flag that indicates whether the previous session was aborted.

If the previous session was aborted, EASL displays the following prompt:

```
A previous EASL session was aborted, do you wish to
recover?
```

Responding to the recover prompt

If you respond with N to the recover prompt, then the LOG file is reset and you lose your work.

If you enter Y, the previous EASL session will "replay"

- up to the point where it was aborted or
- until you press the INT key.

Pressing the INT key causes the LOG file to be truncated on the spot. It is a good idea to make a backup copy of the LOG file if it is important.

Warning: Be careful not to replay the recovery log from one edit session into a new session in which you are working with a different file.

RECOVERY WHEN COPY-BACK FAILS

Introduction

When an EASL session is terminated normally, but the copy-back mechanism fails, you can recover your work from the WORK file.

Function of the WORK file

When you end an editing session with the End Command

- the entire file is first stored in the WORK file and then
- the WORK file is copied back on top of the source file.

How to recover

It is possible for something to go wrong during this copy-back. In this case, you can recover your source file from the WORK file by

- renaming the WORK file to something other than WORK/E$,
- checking the file to make sure everything is there, and
- copying the renamed file into your original file.

Chapter 7.
INTRODUCTION TO RASL

OVERVIEW

Description

RASL (Roam Among Symbolic Locations) is the symbolic debugger for DASL programs.

Purpose

RASL allows you to stop the execution of a program at particular points and examine variables.

Advantages

Some of the important advantages of RASL are listed below.

- With RASL, DASL programs can be debugged without the need for assembly code listings or absolute address calculations.
- RASL permits the user to set breakpoints by function name or line number and to examine variables by name.
- RASL makes it easy to debug programs with complex workstation I/O. It also allows debugging of independent tasks.
- There is no increase in the amount of generated code for debugging because all of the debugging information is in files.

Parts of RASL

RASL consists of two parts which communicate over a pipe.

- RASLRES$ is a small (400 byte) relocatable routine which is linked with the program being debugged.
- RASL is a separate command file which accepts debugging keyins from the user and displays debugging information.

ORGANIZATION AND USE OF THIS SECTION

Introduction

This section is organized to provide programmers with reference information that

- addresses all the aspects of RASL and
- is easy to retrieve.

Coming up

The following table describes Chapters 7-10 of this book.

THIS chapter...	DESCRIBES...
INTRODUCTION TO THE RASL REFERENCE SECTION	the basic features of RASL and the organization of the RASL section of the guide.
REQUIREMENTS FOR USING RASL	• the required hardware and software for RASL and • what the user must do to set up debugging files for RASL.
HOW TO USE RASL	all aspects of using RASL to debug a DASL program.
ASSEMBLY LANGUAGE DEBUGGING WITH RASL	how to use RASL to debug assembly language programs.

Chapter 8.
REQUISITES FOR USING RASL

OVERVIEW

Introduction

Before you can use RASL, you need to

- make sure that you have the appropriate software and hardware and
- set up the debugging files which RASL requires.

Coming up

This chapter describes the following five requirements for running RASL:

- appropriate versions of SNAP and LINK,
- properly formatted DASL functions,
- two workstations,
- a pipe for communication between the parts of RASL, and
- debugging information files.

VERSIONS OF SNAP AND LINK

Description

RASL requires the following versions of SNAP and LINK:

- SNAP 1.2.D or later and
- LINK 1.4.1 or later.

PROPERLY FORMATTED FUNCTIONS

Description

In order for RASL to work properly, the first executable statement of a function must be on a separate line from the assignment operator following the function header.

TWO WORKSTATIONS

Description

RASL normally requires two workstations, one to run RASL and one to run your program.

If your program can run without a workstation, you may be able to run it as an independent background task.

Using two different processors

If the two workstations are running on two different processors, the one running RASL must be a data resource processor because of the way pipes work. See the RMS System Software Installation and Configuration User's Guide, Document # 50624. It describes how to use the CONFIG utility to set up a processor to be a resource processor.

The other processor must have access to the resource processor which contains the pipe local resource. In order to provide this access, you insert an environment on the other processor and press the Enter Key at the RESOURCE prompt.

PIPE FOR COMMUNICATION

Description

The two parts of the RASL program, RASL and RASLRES$, require a pipe in order to communicate.

Supplying the pipe name

Follow these steps to supply the name of the pipe to be used for communication between RASL and RASLRES$.

STEP	ACTION
1	Enter on the RMS command line: COPY RASLPIPE,RASLPIPENAME This command copies RASLPIPE to a file called RASLPIPENAME in your catalog.
2	In the file RASLPIPENAME, locate the pipe name HSPRASLPIPE: • in the comment line at the beginning and • in the line starting with RASLPNAME$. Change the initials HSP to your own name, initials, or project name. This name should be unique so that it will not conflict with anyone else's pipe.
3	Enter on the RMS command line: SNAP RASLPIPENAME;N

Description

The RASL command uses debugging information files to determine code and data addresses and variable types. These debugging information files must be available to the RASL command.

RASL is more useful if the module source files are also available, because then the text of the program can be displayed.

How debugging files are produced

The debugging information files are produced by the DASL compiler and the linker when the DEBUG option is given with DASL, SNAP, and LINK.

The following chart shows the specific effects of the DEBUG option.

THE DEBUG option causes...	TO...
the DASL compiler	• produce a symbol table file with the name <source>/SYM1 and • include directives in the assembly language output code relating source file line numbers to code addresses.
the SNAP assembler	• include this line number information in the output relocatable file.
LINK	• produce for each output segment a file called /SYM2 which contains • the line number information from the relocatable file for each PROG, • the addresses of PROG's PABs (code and data areas), and • the addresses of the PROG's external references.

Using the DASL/CHN file

The DASL/CHN file can be used to compile, assemble, and link a single DASL source file.

Two of the DASL/CHN options, DBUG and RASL, both set up a program to be run with RASL.

A complete listing of the DASL/CHN file is included in Chapter 11: INTRODUCTION TO RUNNING A DASL PROGRAM.

)omparison of DASL/CHN options

The DBUG and RASL options are similar in that they both

- cause the DEBUG option to be given with DASL, SNAP, and LINK in order to produce the debugging information files and
- INCLUDE the file, RASLPIPENAME, which supplies the pipe name for communication between RASL and RASLRES$.

The difference between the two options is shown in the following table.

THIS option	CAUSES this module to be included...	CONSEQUENCE
DBUG	D$LIB.D$STARTS	D$STARTS calls the RASLRES$ routine as soon as the program begins executing. If the pipe does not exist, then the program executes normally without RASL.
RASL	D$LIB.START	To run RASL you have to call RASLRES$ as a subroutine from your program, perhaps conditionally based on a command line option.

Calling RASLRES$

If you use the RASL option, you must call RASLRES$ as a subroutine from your program in order to run RASL.

The RASLRES$ subroutine returns a D$CCODE indicating the result of the $OPEN of the pipe. This permits a program to take special action if the pipe cannot be opened. However, if the program does not check for an error and the pipe cannot be opened, the program will continue just as if RASLRES$ had not been called.

Example: The following code causes the program to take special action if the pipe cannot be opened.

```
IF raslOpt THEN
    IF RASLRES$() && D$CFLAG THEN $ERMSG();
```

Calling RASLEND$

You can call RASLEND$ as a subroutine from your program in order to close the the pipe and turn off the error traps before the end of the program. If you do not call RASLEND$, the program runs to the end with RASL.

Warning: Before calling RASLEND$, you should have cleared any breakpoints or tracepoints.

Modifying DASL/CHN

The DASL/CHN file can serve as a prototype for setting up more complex programs to be run with RASL.

The DEBUG option must be specified

- on the DASL and SNAP command for each module for which you want symbolic information available to RASL and
- on the LINK command for each segment for which you want symbolic information available.

If you are debugging several overlays, RASLRES$ should probably be linked into your root segment.

The file RASLPIPENAME must be INCLUDEd in the directives to LINK.

Chapter 9.
HOW TO USE RASL

OVERVIEW

ntroduction

This chapter provides detailed information on how to use RASL to debug a DASL program.

nteraction between RASL and program

At any time, either one or the other of the following is happening:

- your program is running and RASL is inactive or
- your program is suspended and you are interacting with RASL.

When you first invoke RASL, the program is suspended.

Process for using RASL

The following table outlines a typical use of RASL. The purpose of the table is to provide an overall view of how the different features of RASL work together. There can be variations of this typical use. They are described in this chapter and in Chapter 10.

STAGE	DESCRIPTION	SEE PART(S)
1	You invoke RASL.	A
2	You enter a segment, module, and function on the RASL menu lines.	B
3	You set one or more breakpoints or tracepoints on the RASL screen.	C
4	You use the RASL command Execute to begin execution of the program which you want to debug.	D
5	The program is suspended when it reaches a breakpoint and RASL is reentered. At this time you may examine variables on the RASL screen or use any of the RASL commands.	B,D
6	You repeat the process as often as necessary, beginning at Stage 2, 3, or 4.	
7	You quit RASL.	D

Coming up

The following chart shows how the information in this chapter is organized.

PART	TITLE
A	Invoking RASL
B	Description and Use of the RASL screen
C	Breakpoints and Tracepoints
D	Description and Use of RASL Commands

Procedure for invoking RASL

Follow this procedure to invoke RASL.

STEP	ACTION
1	Enter the command RASL at workstation 1. This workstation must be running on a resource processor if the two workstations are running on different processors.
Results: RASL checks to make sure that no one else is already using the pipe with the name specified in the RASLPIPENAME file. RASL creates the pipe if it doesn't already exist and displays the message Pipe XXXRASLPIPE:L is being waited on	
2	Enter the name of the program to be debugged at workstation 2.
Result: Once the program enters RASLRES$, the program is suspended and the RASL menu is displayed at workstation 1.	

PART B: THE RASL SCREEN

Introduction

This part describes what the RASL screen looks like and how to enter and interpret information on the screen.

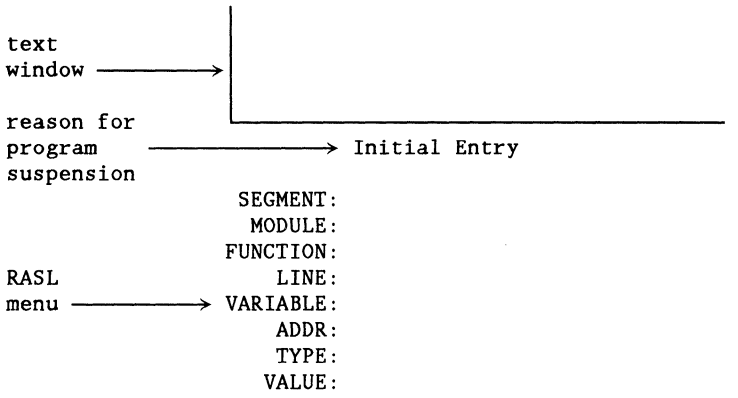
Coming up

This part includes the following topics:

- description of the RASL screen,
- accessing parts of the screen,
- entering information on menu lines,
- how RASL updates the menu display, and
- purpose and description of menu lines.

Diagram of screen

This diagram shows what the RASL screen looks like upon initial entry.



Note:

On 6600 type processors with 12-line screens, the text window is not displayed with the menu.

Parts of the screen

The following table describes the parts of the RASL screen.

PART	FUNCTION
text window	Shows 11 lines of the currently selected source module
reason for program suspension	Explains the reason why the program being debugged is suspended and RASL is active
RASL menu	Displays eight lines which can be used for two purposes: • to allow the user to make selections or • to display information about the state of the program.

Introduction

These pages describe procedures for accessing different parts of the screen, including

- moving the cursor,
- scrolling the text window, and
- changing the size of the text window.

RASL uses the 12-line screen of 6600 type processors in a modified way.

The use of the 24-line screen is described first, followed by modifications for the 12-line screen.

Moving the cursor

The following chart shows how to move the cursor to the beginning of any line on the screen.

TO move the cursor...	PRESS on the numeric keypad...
up or down to any menu line	8 (up) or 2 (down).
back and forth from the menu lines to the text window	5.
up or down in the text window	8 or 2.

Scrolling the text window

To scroll the text window up or down through the source module, use the KBD (up) or DSP (down) Keys.

Changing to a larger text window

To display a 22-line text window (on a processor which supports a 24-line screen), press the ENTER Key while the cursor is in the text window.

The bottom line of the screen shows just the LINE menu line.

To restore the complete RASL menu, repress ENTER.

Accessing the text window on 6600 processors

On 6600 type processors with 12-line screens, the text window cannot be displayed at the same time as the menu.

You can display either

- the full RASL menu or
- the text window (11 lines) and just the LINE line of the menu.

To switch back and forth between these options, press the number pad 5 key and the KBD Key at the same time.

Moving the cursor on 6600 processors

On 6600 type processors which do not encode the numeric keypad, you simulate the up and down cursor keys by using the number pad 2 and 8 keys and the KBD Key at the same time.

Scrolling on 6600 processors

To scroll through the source program on 6600 processors

- hold down the DSP Key and then press KBD (scroll down)
  or
- hold down the KBD Key and then press DSP (scroll up).

ENTERING INFORMATION ON MENU LINES

Introduction

One of the functions of the menu lines is to allow the user to enter information.

Procedure

To enter information on a menu line

STEP	ACTION
1	Move the cursor to the line
2	Enter the information
3	Press the ENTER Key

The following chart explains what happens when you enter various keystrokes.

IF you...	THEN RASL...
backspace to the beginning of the line or press CANCEL	restores the old information on the line.
enter invalid information	beeps and restores old information.
press the ENTER Key by itself	responds as if you had retyped the current information on the line.
enter a space followed by the ENTER Key	clears the line.

Entering numeric information

When you type in numbers, they are in

- octal if they begin with zero or
- decimal if they do not begin with zero.

HOW RASL UPDATES THE MENU DISPLAY

Introduction

One of the functions of the RASL menu lines is to display information about the state of the program.

RASL automatically updates the display whenever RASL is reentered after the initial entry to RASLRES$.

Description of normal display

In most cases of reentry to RASL, RASL updates the MODULE, FUNCTION, and LINE lines and displays on the VALUE line the current value of the selected variable or address.

Display update for special cases

If the program address belongs to a module without a symbol table file, <module>/SYM1, then RASL

- clears the FUNCTION, LINE, and VARIABLE lines and
- sets the ADDR line to the program address and displays its VALUE.

If the program address does not belong to a module in the current segment, RASL does the same thing but it also clears the MODULE line.

Display update when a run-time error occurs

RASL sets the ADDR line to the program address if the RASL entry occurred because of

- an undefined or privileged instruction or
- memory access or write violation.

USING MENU LINE: SEGMENT

Purpose

On the SEGMENT menu line, you enter the name of the segment in which you are interested.

For single segment programs, you enter the program (command) name.

Requirement

The LINK output debug file /SYM2 must be available.

Results of selection

If the segment selection is successful, RASL

- blanks the information on the MODULE, FUNCTION, and LINE lines,
- moves the cursor to the MODULE line, and
- attempts to select a module with the same name as the segment.

The module selection is successful for programs which have the same name for the source file and command.

USING MENU LINE: MODULE

Purpose

On the MODULE menu line you must select a module.

Description of module

A module consists of

- a DASL source file,
- its associated relocatable PROG in a relocatable file or library, and
- the same PROG within an absolute code command or segment.

Requirements

There should normally be a compiler debug file <module>/SYM1. The one exception is described on the pages called Assembly Language Debugging.

There must have been a PROG with the module name LINKed into the current segment.

Results of selection

If the module selection is successful, RASL blanks the current information on the FUNCTION and LINE lines and moves the cursor to the FUNCTION line.

If RASL can find a file with the module name and extension TEXT, then the text window is initialized to the beginning of this file. Otherwise, the text window area is not used for the module.

USING MENU LINE: FUNCTION

Purpose

On the FUNCTION menu line, you select a function within the current module.

Results of selection

If the function selection is successful, RASL

- moves the cursor to the LINE menu line and
- displays the line number of the first line of code in the function after the function entry code.

The function selection also determines the scope of variable references.

No function selection

If you do not want to select a function, enter one space and press the ENTER Key.

Result: Any information on the LINE menu line is cleared.

This can be useful if you are interested in examining a global variable which has the same name as a local variable.

USING MENU LINE: LINE

Purposes

The LINE menu line displays the line number for a function selection. It can also be used to select a line number within the current module.

Results of selection

If the line selection is successful, that line appears in the text window. An arrow appears next to any line you select.

Changing the current LINE

There are three ways to change LINE information:

- enter a function on the FUNCTION line, or
- enter a line number on the LINE menu line, or
- change the cursor position in the text window using the cursor, KBD, or DSP Keys.

  Note: If the cursor position is changed

  - the line indicator moves and
  - the LINE menu line displays the line number of the current cursor position.

Restriction

The line numbers in RASL always refer to the source file. Therefore, you cannot use RASL to debug executable code in an INCLUDE file.

USING MENU LINE: VARIABLE

Purpose

On the VARIABLE menu line, you can select a variable in the current module.

Kinds of variables allowed

The selected variables can be

- a simple name or
- a complex variable containing ∧, [number] and .field components.

Examples: You can select

- num
- a∧.b[1].c

If ∧ is used, the value of the preceding pointer must not be NIL (0).

The meaning and type checking of variables is as in DASL.

Variables in recursive functions

Because of the way recursive functions are allocated from a stack, you cannot select a recursive function on the FUNCTION line and then look at its variables.

Instead, you can use the Caller command repeatedly to step out to the desired function and then examine its variables. The Caller command is described in PART D of this chapter on the pages titled USING RASL COMMAND: CALLER.

Scope of variables

RASL will first look for a variable declared locally to the current function if there is one. Then it will look for variables declared globally.

The variable may be external if it

- was declared EXTERN in the current function or globally and
- was actually referenced by the code in the current module.

Results of selection

If the variable selection is successful, RASL displays the variable's

- absolute address on the ADDR menu line,
- type on the TYPE line, and
- value on the VALUE line.

Clearing lines

To clear the VARIABLE, ADDR, TYPE, and VALUE lines, enter a space and the ENTER Key on the VARIABLE line.

USING MENU LINE: ADDR

Purposes

The ADDR menu line normally displays the address of a selected variable.

You can also select an address on the ADDR line.

Selecting an address

The following chart shows the different ways that you can select an address.

TO select...	ENTER on the ADDR menu line...
a particular absolute address	the absolute address (preceded by zero if octal).
the previously displayed address plus or minus one	the right or left cursor keys.
the previously displayed address incremented or decremented by a specific number	+ or - followed by the number.
the two byte value at the current address as the new address	^.

Results of selection

When a new address is selected,

- the VARIABLE line is cleared,
- the TYPE line is changed to null for unknown, and
- the VALUE of the selected location is displayed.

Clearing lines

To clear the ADDR, TYPE, and VALUE lines, enter a zero and the ENTER Key on the ADDR line.

Purposes

The TYPE menu line normally displays the type of a selected variable.

You can also enter information on the line to change the type.

Changing the type

The following examples illustrate two common situations when it is useful to change the type.

- You have a character stored as a BYTE and want to see it as a CHAR.
- You are passing a string to a function by using a pointer to a CHAR and want to see the whole string.

Changing the type

You may change the type by entering

- any DASL predefined type,
- a TYPDEF name defined in the current module,
- ^ for pointer,
- [] for array,
- STRUCT, or
- space and ENTER Key for null.

The information on the TYPE line determines the format of the VALUE line.

Purposes

The VALUE menu line displays the value of a selected variable or address.

You can also enter a value on this line to change the value of a variable or address.

Format of value

The format of the value is determined by the displayed TYPE.

The following chart shows the value format for different data types.

TYPE	FORMAT of value
CHAR	character
BOOLEAN	FALSE or TRUE
BYTE	one byte unsigned
UNSIGNED	two bytes unsigned
INT	two bytes signed, if negative
LONG	four bytes signed
POINTER	octal address of pointer
CHARACTER ARRAY	string (possibly truncated to fit on the line)
MISCELLANEOUS TYPES: • other array types • STRUCT • UNION • NULL	they have one and two byte values at the current address that are displayed in several formats: • character • one byte octal • two bytes octal • two bytes reversed octal • one byte unsigned • two bytes unsigned • two bytes signed, if negative. Additionally, the entire value is displayed as a sequence of octal bytes and as a string.

Changing the value

You can enter a value on the VALUE menu line to change the value of a variable or address.

The value may be

- an octal or decimal number with optional preceding sign or
- a string in single quotes with DASL's double quotes forcing rules.

If the current type is an array or STRUCT, then the size of the type and the size of the value are compared. The smallest of the two determines the number of bytes which will be modified. The size of a numeric value is one byte if the value fits in eight bits, two if 16 bits, and otherwise four.

PART C: BREAKPOINTS AND TRACEPOINTS

Description

A breakpoint or tracepoint may be set at any line of a program which actually generates object code. Breakpoints and tracepoints modify the program at the line where they are set.

A breakpoint causes an executing program to

- suspend execution before the breakpoint line is executed and
- reenter RASL.

A tracepoint also causes the program to reenter RASL before the line is executed. However, RASL resumes execution of the program after updating the RASL menu display.

Coming up

This part discusses the following topics:

- using breakpoints and tracepoints, and
- restrictions on placement of breakpoints and tracepoints.

USING BREAK AND TRACEPOINTS

Introduction

These pages describe how to set and use breakpoints and tracepoints.

When to use

Breakpoints allow the user to stop program execution, examine values of variables, and trace the logic flow.

Tracepoints allow the user to see a variable changing without stopping the program. The user can also set several tracepoints and watch which ones are executed.

Setting a breakpoint

There are two ways to set a breakpoint. You can enter B

- on the LINE menu line, or
- in the text window area with the cursor positioned on the desired line.

Results: If the chosen line does not actually generate object code, then the LINE is set to the next line which does.

The LINE menu displays the word "Breakpoint".

A "B" appears to the left of the line in the text window.

If the breakpoint is set in the text window area, then the LINE is set to the following line.

Removing a breakpoint

To remove a breakpoint, enter "B" again on the desired line.

Setting and removing a tracepoint

To set or remove a tracepoint, use "T" and follow the same procedure as for setting and removing breakpoints.

Association of variables with breakpoints and tracepoints

RASL permits different variables to be associated with different breakpoints and tracepoints.

Whenever a breakpoint or tracepoint is set, the current state of each of the RASL menu line displays is remembered and associated with that breakpoint or tracepoint.

Exception: the VALUE menu line display is not remembered.

IF the LINE display is later set to or execution resumes at a line which has a breakpoint or tracepoint set, THEN

- the display is restored to what it was when the breakpoint or tracepoint was set and
- the current value of the selected variable or address is displayed on the VALUE line.

Examining breakpoints or tracepoints

To examine all of the breakpoints or tracepoints which have been set, press the left or right cursor keys on the LINE menu line.

Each time you press a cursor key, the RASL menu is updated to display all of the information associated with the next (or previous) breakpoint or tracepoint in the program.

Using the Execute command with breakpoints

The Execute command causes the debugged program to resume from where it was last suspended. See the pages titled USING RASL COMMAND: EXECUTE in PART D of this chapter.

If a program has been suspended due to a breakpoint or tracepoint, the Execute command causes the breakpoint or tracepoint to be temporarily removed so that execution can resume.

This is indicated on the screen by

- the message "Breakpoint pending" or "Tracepoint pending" on the LINE menu line and
- a lowercase "b" or "t" to the left of the line in the text window.

The next time RASL is reentered, the breakpoint or tracepoint is restored.

Entering RASL through a loop

If you want to enter RASL each time through a loop, you need to set more than one breakpoint or tracepoint in the loop.

Introduction

There are several restrictions on the placement of breakpoints and tracepoints. If you try to set a breakpoint or tracepoint where one is not allowed, then RASL automatically sets the breakpoint or tracepoint at an acceptable position.

In each of the following examples, the user tries to set a breakpoint at line X, but RASL actually sets the breakpoint at line B. The same resetting would occur for tracepoints.

Example 1

If the user tries to set a breakpoint on the right brace at the end of a LOOP statement and the last statement before the right brace is a WHILE statement, then RASL sets the breakpoint on the next statement after the loop.

The reason for this is that the end of the loop has been optimized into the WHILE.

```
        LOOP {
           y += x;
         WHILE ++x <= 10;
X          };

B       IF y > 40 THEN z := 0;
```

Example 2

If the user tries to set a breakpoint on the right brace at the end of a function and the last statement of the function is an assignment to RESULT, then RASL sets the breakpoint at the beginning of the next function.

The end of the function has been optimized into the assignment.

```
        funct() :=
        {
           x := a + b;
           y := x * 3;
           RESULT := y;
X          };

B       mod() :=
        {
           c := r + s;
           s++;
           };
```

Example 3

If the user tries to set a breakpoint on a line which starts with an ELSE, then RASL sets the breakpoint after the end of the statement following the THEN.

```
        IF x = y THEN {
           b := 0;
           a := 1;
B          }
X       ELSE b := 5;
```

Example 4

If the user tries to set a breakpoint on the closing brace of a CASE statement, then RASL sets the breakpoint on the code which evaluates the CASE expression.

```
B       CASE x {
           one : a := 3;
           four : a := 5;
X          };
```

PART D: RASL COMMANDS

Introduction

This part describes how to execute and use the five commands in RASL:

- Execute,
- Caller,
- Where,
- Quit, and
- Invert.

Executing commands

Follow this procedure to execute a RASL command.

STEP	ACTION
1	Press the Command Key. Result: The five commands are displayed on the bottom line. The Execute Command is displayed in inverse video to show that it is the selected command.
2	Change the selected command by entering either • its first letter, • the Right Cursor Key or a space to move to the next command, or • the Left Cursor Key to move to the previous command.
3	Execute the command by pressing the ENTER Key.
You can cancel the command mode at any time by repressing the Command Key.	

EXECUTE COMMAND

Purpose

The Execute Command causes execution of the debugged program to resume from where it was last suspended.

Special condition of reentry

If there is a breakpoint or tracepoint set at the program reentry address, the breakpoint or tracepoint is temporarily removed from the program so execution can resume. See the pages titled USING BREAKPOINTS AND TRACEPOINTS in PART C of this chapter.

Conditions for suspending execution

While the program is running, RASL displays the message "Executing...". Execution continues until one of the following occurs:

- breakpoint,
- tracepoint,
- undefined or privileged instruction,
- access or write violation, or
- debug key sequence (KBD-CAN-KBD) typed at the keyboard running the program.

Purpose

The following chart describes the uses of the Caller Command.

IF you execute the Caller Command...	THEN RASL displays...
for the first time after an entry to RASL or after the Where Command	the module, function, and line number which called the function where execution was suspended.
again	the next outer function call. Repeated use of the Caller Command reveals the current program call nesting.

Use of Caller with recursive functions

The Caller Command can be used to examine variables in recursive functions. See the pages titled USING MENU LINE: VARIABLE in PART B of this chapter.

Restrictions on use of Caller Command

You cannot use the Caller Command again if

- the function line was blank on the reentry display or after the previous Caller Command (meaning the return address is unknown) or
- the program suspension occurred in the function header before the return address was stored.

In the second case, you can set a breakpoint on the next line and enter the Execute Command.

Module name restriction

Module names which are longer than eight characters can cause problems with the Caller Command, because module names are truncated to eight characters by SNAP and LINK.

If the Caller Command gets a truncated module name, then RASL cannot find the /SYM1 or source file.

Description of uses

The following chart describes the use of the remaining three RASL commands.

COMMAND...	DESCRIPTION
Where	Displays the program reentry point, just as when RASL is entered
Quit	Terminates RASL The debugged program remains suspended and must be aborted from the workstation
Invert	Reverses the shift key inversion, which is initially off

Chapter 10.
ASSEMBLY LANGUAGE

USING RASL TO DEBUG ASSEMBLY CODE

Introduction

RASL was designed as a debugger for DASL programs.

However, RASL can also be used to debug programs written in assembly language.

Description

RASL allows you to use the ADDR and LINE menu lines in alternate ways for assembly language debugging.

You can perform either of these alternatives:

- set the current address of the ADDR line to the machine state save area in RASLRES$ or
- change the mode on the LINE line so that entries on this line are absolute addresses instead of line numbers.

These uses of the LINE and ADDR lines are only meaningful if you have used assembly language code in your program.

Using the ADDR line

To set the current address on the ADDR line to the machine state save area in RASLRES$, enter an M on the ADDR line.

You may examine or modify this area as any other memory area.

The following table indicates the number of bytes used and the contents of each item in the 32-byte machine state save area.

NUMBER OF BYTES	CONTENTS
1	internal use
13	RMS interrupt state save area, which includes • condition code (1 byte) • A, B, C, D, E, H, L, X registers (8 bytes) • program reentry address (2 bytes) • internal use (2 bytes)
2	address of recursive frame pointer (D$RFRAME)
16	top 8 stack entries (excluding reentry address)

Jsing the LINE line

To toggle a mode in which entries on the LINE line are absolute addresses instead of line numbers, enter an M on the LINE line. (Remember to precede octal numbers with a zero.)

Breakpoints and tracepoints work the same as for line numbers.

Exceptions to use of ADDR and LINE

If there is no /SYM1 file for a name entered on the MODULE line,

- the ADDR line is set to the address of the second PAB of the module (usually data) and
- the LINE line, if toggled to the absolute address mode, shows the address of the first PAB of the module (usually code).

Chapter 11.
RUNNING A DASL PROGRAM

OVERVIEW

Scope of this section

This section briefly describes the steps involved in going from DASL source code to executable object code.

What is not covered: This section does not include information on compiling and linking programs with more than one module or programs which have overlays.

For complete and detailed information on linking see the section on LINK in the RMS Utility User's Guide.

Coming up

This section provides introductory information on

- the process of going from DASL source code to executable code and
- the use of the chain file, DASL/CHN, to perform this process.

The complete DASL/CHN file is listed at the end of this chapter.

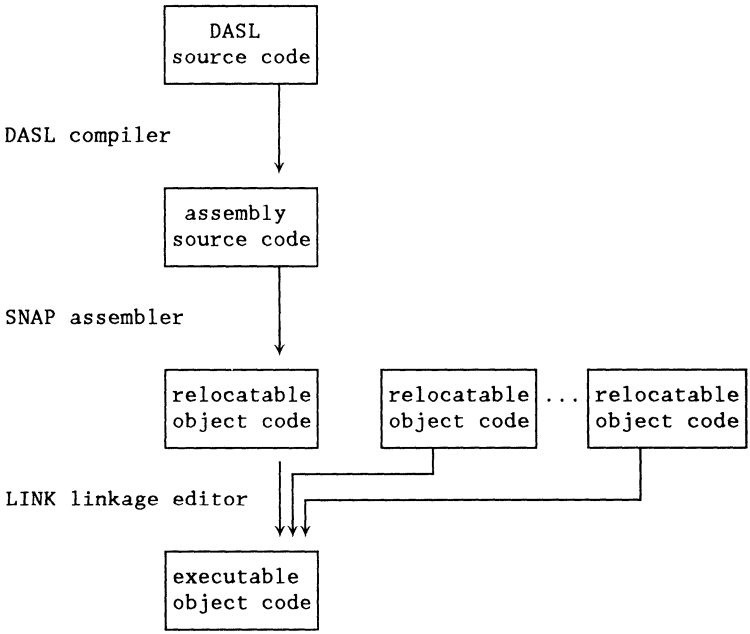
Stages of process

The process of going from DASL source code to executable object code involves the following three stages:

STEP	ACTION
1	Compilation by the DASL compiler
2	Assembly by the SNAP assembler
3	Linkage by the LINK editor

Diagram

The diagram below outlines the stages of the process.



Function of the DASL compiler

The DASL compiler consists of two serial passes.

- Pass one transforms DASL source code into a machine independent, intermediate logical representation.
- Pass two generates the actual assembly output code.

Because pass one is machine independent, different code generators may be hooked onto it for different machines.

Function of SNAP

The SNAP assembler generates a relocatable code module from the assembly source code produced by the compiler.

Description of relocatable code modules

The additional relocatable code modules in the diagram represent either

- modules which have already been compiled and assembled by the user, or
- modules which are part of some standard RMS package, such as for run-time support, I/O routines, User Function Routines, and so on.

Function of LINK

LINK performs the final blending of relocatable code modules required to make an executable command file.

The directives to the linker specify which modules are required to resolve all of your program's references.

USING THE DASL/CHN FILE

Introduction

DASL programmers frequently use a chain file to accomplish the three steps of compiling, assembling, and linking a program module.

The following pages describe how to use the standard chain file, DASL/CHN.

Definition of DASL/CHN

DASL/CHN is a chain file which is designed to work for programs which have only a MAIN module.

DASL/CHN can serve as a prototype for more complex programs.

A listing of DASL/CHN is included at the end of this chapter.

Description of DASL/CHN options

DASL/CHN has five options which the user may specify on the DASL/CHN command line. These options determine

- which options are given to DASL, SNAP, and LINK, and
- which files and libraries are included in the LINK directives.

The following chart describes the function of each option.

THIS option...	PERFORMS this function...
DBUG	compiles and links a program so that it produces information needed by the debugger, RASL. When you use the DBUG option, the program tries to activate the debugger every time it is run. If the pipe needed by RASL cannot be opened, the program executes without RASL.
RASL	compiles and links a program for debugger, RASL. The program itself must call RASLRES$ if it wants to use the debugger.
TRACE	compiles and links a program so that it produces information needed by TRACE.
CODE	determines whether the code generator for the 6600 or the 5500 instruction set is used. The default is CODE = 6600.
PRT	generates a print file of the LINK map.

Running DASL/CHN

To run DASL/CHN, enter on the command line

DASL/CHN[;IN=<filename>][,<options>]

DASL/CHN prompts you for the name of your program if you do not include it on the command line.

If no errors are encountered, an executable file is produced called <filename>/CMD.

Display of compilation errors

If the compiler finds any errors, it describes each error and indicates the line number where it occurred.

The error information is displayed on the screen and is also written to a text file called DASLERR/TEXT.

Description of LINK directives

The directives to LINK specify to the linker which modules are required to locate all of your program's references.

The INCLUDE directive takes the entire /REL module specified and plugs it into the /CMD file, locating any references.

The LIBRARY directive searches the /REL module only for those entities that are required to locate references.

LINK directives in DASL/CHN

Some of the directives in DASL/CHN are conditional and depend upon which options are given.

The following table briefly describes the LINK directives which appear in DASL/CHN.

DIRECTIVE	DESCRIPTION OF FILE OR LIBRARY
INCLUDE D$LIB.D$START	D$START is responsible for starting up the execution of a DASL program. D$START calls the MAIN function of the program.
INCLUDE D$LIB.D$STARTS	D$STARTS first calls the RASLRES$ routine and then calls MAIN.
LIBRARY RASLRES$ INCLUDE RASLPIPENAME	This library and file are needed in order to run RASL.
LIBRARY D$IO	D$IO is one of the many I/O packages which are available. Other I/O packages are discussed in the SYSTEM PROGRAMMERS REFERENCE MANUAL.
LIBRARY D$LIB	D$LIB contains the run-time support needed by DASL programs, including in part • recursion support, • software divide and multiply for the 5500, and • block move/comparison.
LIBRARY RMSUFRS	RMSUFRS is the standard RMS package of User Function Routines and other definitions. Some of these are called by D$LIB and must, therefore, be included. Library RMSUFRS should be the last library directive to LINK

Modifying DASL/CHN

If your program consists of more than one module or references other function libraries, you may add the appropriate INCLUDE or LIBRARY directives to DASL/CHN.

For information on making more extensive modifications, see the CHAIN and LINK sections of the RMS Utility User's Guide.

LISTING OF DASL/CHN FILE

Introduction

The following is a listing of the complete DASL/CHN file.

DASL/CHN file

```
/& DASL/CHN ;IN=file,DBUG,RASL,TRACE,CODE,PRT
/&
/&     DBUG  - Compile and link for debugger
/&     RASL  - Compile and link for debugger, but
/&             program must call RASLRES$
/&     TRACE - Compile with debug options for TRACE
/&
/&     CODE  = 5500 or 6600 (6600 assumed)
/&     PRT   - Generate print file of LINK map
/&
// ASSIGN TIME=CLOCK()
// IF ~ IN
//   IF ~ OUT
//     KEYIN "ENTER PROGRAM NAME: " OUT
//   XIF
//   ASSIGN IN = OUT
// XIF
// IF DBUG | RASL | TRACE
DASL #IN#;CODE=#CODE#,DEBUG
SNAP DASLASM;N,DEBUG
// IF PRT
LINK,#IN#/CMD,#IN#;ERR,FASTLIB,NEW,PRNT,DEBUG
#TIME#
// ELSE
LINK ,#IN#/CMD,#IN#;ERR,FASTLIB,NEW,DEBUG
// XIF
// ELSE
DASL   #IN#;CODE=#CODE#
SNAP DASLASM;N
// IF PRT
LINK ,#IN#/CMD,#IN#;ERR,FASTLIB,NEW,PRNT
#TIME#
```

```
// ELSE
LINK ,#IN#/CMD,#IN#;ERR,FASTLIB,NEW
// XIF
// XIF
SEGMENT #IN#,,$LOADTOP
INCLUDE DASLASM
// IF DBUG
INCLUDE D$LIB.D$STARTS
// ELSE
INCLUDE D$LIB.D$START
// XIF
// IF DEBUG | RASL
INCLUDE RASLPIPENAME
LIBRARY RASLRES$
// XIF
// IF CODE = 5500
LIBRARY D$IO/REL5
// ELSE
LIBRARY D$IO
// XIF
LIBRARY D$LIB
LIBRARY RMSUFRS
SIGNON #IN# - #TIME#
```

Chapter 12.
MAKE REFERENCE SECTION

OVERVIEW

Purpose

MAKE is a program to aid in automating the maintenance of a set of interrelated files.

Common uses of MAKE

MAKE is commonly used to detect updates to the various source and INCLUDE files that implement a user command. MAKE then recompiles only those source modules which have changed or that INCLUDE text which has changed.

MAKE can also be used to make backups or listings of files which have changed.

How MAKE works

The user provides a rules file to MAKE which specifies

- the file interdependencies for a system of files and
- the actions to be taken in order to update any dependent files.

MAKE uses this rules file in order to create a chain file. The chain file includes a minimal set of commands which will assure that a specified file is up-to-date. MAKE automatically executes this chain file.

Coming up

In this chapter, we discuss

- Limitations of MAKE,
- The MAKE Command Line,
- MAKE Rules File Format,
- How MAKE Decides What to Remake,
- An Example Illustrating the Rules File and Chain File, and
- An Example Illustrating How MAKE Decides What to Remake.

LIMITATIONS OF MAKE

Introduction

MAKE is only as good as the file time stamps stored in the files. The user must try to ensure that the file times are accurate and up to date.

File times of RMS 1 and 2

In RMS 1, the only file time available is the RMS create time.

There are commands, such as LINK, which do not update the create time of the file being modified.

Later versions of RMS may have a last modified time which MAKE will use.

Use of the command TOUCH

The command TOUCH, an RMS utility, takes a file name as its only parameter and changes the file's create time to the current time.

If you are using RMS 1, it is important to run TOUCH before invoking MAKE.

MAKE RULES FILE FORMAT

Description

The user provides a rules file to MAKE which specifies

- the file interdependencies for a system of files and
- the actions to be taken in order to update any dependent files.

Description of rules file format

The rules file format is an extension of the chain file format which the user might normally use to cause the required actions to be accomplished.

THE MAKE COMMAND LINE

Introduction

The following pages describe each of the file specifications and options for the MAKE command line.

Format for MAKE command line

The format for the MAKE command line is:

```
MAKE [TARGET=target][RULES=rules][CHN=chain];
     [options]
```

Description of file specifications

The chart below describes each of the optional file specifications and shows the default name for the file.

FILE SPECIFICATION	DESCRIPTION	DEFAULT
target	Specifies the files that are to be updated	<unit 1>/* The name <unit 1> means that every dependent in the first make-unit is to be updated. The extension * means that the target is a pseudofile, which is a name used to identify a group of predecessors.
rules	Specifies the file containing the MAKE rules	RULES/MAKE The rules environment defaults to the target environment.
chain	Specifies the chain file that MAKE creates and executes	MAKE$/CHN:W

Options

The MAKE command line may include any combination of the following options.

THIS option...	Causes MAKE to...
NORUN	create the required chain file but not execute it.
UPDATE	create a chain file which updates all target file times as if the commands in the make-units for the out-of-date targets were executed.
ALL	assume that all targets are out-of-date and must be remade.
CHNOPT = "chain options" or CHNOPT= chain_option	pass the indicated chain options through to the chain file MAKE creates. That is, MAKE will execute the chain file • MAKE$/CHN:W;chain options, or • MAKE$/CHN:W;chain_option.
IGNORERR	ignore any errors that occur and attempt to generate the chain file.

Use of UPDATE and ALL

UPDATE and ALL are essentially opposites.

UPDATE is useful to update the file times when you know that nothing is really out-of-date but the file times are not indicating that.

ALL is useful when you have reason to believe that the file times do not indicate the actual state of "up-to-date-ness."

MAKE RULES FILE FORMAT

Description

The user provides a rules file to MAKE which specifies

- the file interdependencies for a system of files and
- the actions to be taken in order to update any dependent files.

Description of rules file format

The rules file format is an extension of the chain file format which the user might normally use to cause the required actions to be accomplished.

Kinds of syntactic elements

The rules file consists of three basic MAKE syntactic elements. The following chart defines each element and shows its delimiters.

SYNTACTIC ELEMENT	DEFINITION	DELIMITERS
make-unit	A group of commands that must be executed to update any of the files listed in one of the rules file's dependent-list based on the files listed in one of the rules file's predecessor-lists Every make-unit must have at least one dependent-list and one predecessor-list.	{* *}
dependent-list	A list of one or more files which would change if any file in the predecessor-list changed	<* *>
predecessor-list	A list of one or more files which are depended on by the file(s) in the dependent list	(* *)

Syntax for make-unit

A make-unit is delimited by the symbols {* and *}.

All text outside of a make-unit is discarded. The remainder of the line on which {* or *} appears is ignored and discarded.

Make-units may not be nested.

Syntax for dependent- and predecessor-lists

Dependent-lists and predecessor-lists have the same structure.

A dependent-list is delimited by the symbols <* and *>. A predecessor-list is delimited by the symbols (* and *).

If a list contains more than one element, the elements are delimited by any combination of blanks, commas, and line ends.

Dependent-lists and predecessor-lists may not be nested.

The same file specification may appear in more than one dependent- or predecessor-list in the same make-unit.

Example of make-unit

The following example shows one make-unit which is part of a longer rules file.

```
{*
/& (* DEFS/TEXT *)
DASL (* MAIN/TEXT *)
SNAP DASLASM <* MAIN/REL *>
*}
```

This make-unit includes one dependent-list,

```
<* MAIN/REL *>,
```

and two predecessor-lists,

```
(* DEFS/TEXT *) and (* MAIN/TEXT *).
```

File specification format

Each element of a dependent- or predecessor-list is a file specification in standard RMS format.

The name portion of the file has no default and should be specified.

The extension defaults to /TEXT and the environment defaults to blank.

Generic file specifications are not interpreted, and, therefore, will probably not work as intended.

Exception to format

The one exception to the RMS format is that the extension * may be specified.

The * extension indicates to MAKE that this file specification is not really an RMS file but is instead a "pseudofile."

Definition of pseudofile

A pseudofile is a name that is used in a make-unit to identify a group of predecessors.

Example: In the following make-unit, a pseudofile, defs/*, is used to group the two INCLUDE files, SUB2DEFS and DEFS.

```
{*
/& <* defs/* *>
/& (* SUB2DEFS DEFS *)
*}
```

Description of process

MAKE uses the rules file to decide what dependent files need to be remade.

MAKE first scans the entire rules file looking for all dependencies.

Then MAKE applies a set of rules to determine which make-units must be output to the MAKE chain file.

Description of terms

Several terms are used in describing the set of rules which MAKE applies. These terms are described below.

- Dependent. File X is a dependent of file Y if X is in a dependent-list make-unit which has Y in one of its predecessor-lists.
- Predecessor. File A is a predecessor of file B if A is in a predecessor-list of a make-unit which has B in one of its dependent-lists.
- Target. A file is a target if it is one of the files specified by "TARGET" on the command line or it is a predecessor of a target.
- Real file. Any file which is not a pseudofile is called a real file.
- Remade. A file is to be remade if it is a dependent of a make-unit which is to be output.

Rules for determining file age

MAKE usually determines the age of a file by using the file's create time. Additionally, the following rules apply.

- If a real file is to be remade, then it is considered to be the youngest possible file.
- A real file which cannot be found is considered the oldest possible file.
- The age of a pseudofile is determined from the age of its dependents or predecessors, as shown on the following page.

WHEN comparing a pseudofile with one of its...	AND...	THEN the pseudofile is considered...
dependents	at least one predecessor exists	as young as its youngest predecessor
	no predecessors exist	the oldest possible file.
predecessors	at least one dependent which is a target exists	as old as its oldest dependent which is a target.
	no dependents which are targets exist	the youngest possible file.

Rules for outputing make-units

MAKE applies the following rules to determine which make-units must be output to the MAKE chain file. These rules are applied in parallel, along with the rules for determining file age.

- A make-unit will be output to the chain file if one of its dependents is a target and either
  - the target does not exist or
  - one of the target's predecessors is newer than the target.
- Make-units will appear in the chain file in an order which assures that all predecessors which are to be remade are remade before any dependents are remade.
- If a target is to be remade by more than one make-unit, then the make-units will appear in the chain file in the same order they appeared in the rules file.

EXAMPLES: THE RULES AND CHAIN FILES

Introduction

The example on the following pages illustrates

- a typical rules file and
- the chain file which MAKE creates.

Description of the problem

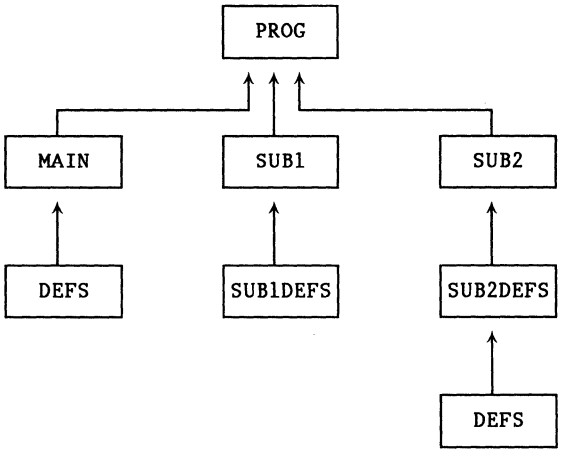
Consider the problem of keeping up to date an RMS command, PROG, which is implemented from three DASL source files, MAIN, SUB1, and SUB2.

Each source file INCLUDES a file of definitions:

- MAIN INCLUDES the file DEFS,
- SUB1 INCLUDES the file SUB1DEFS, and
- SUB2 INCLUDES the file SUB2DEFS.

SUB2DEFS, in turn, INCLUDES DEFS.

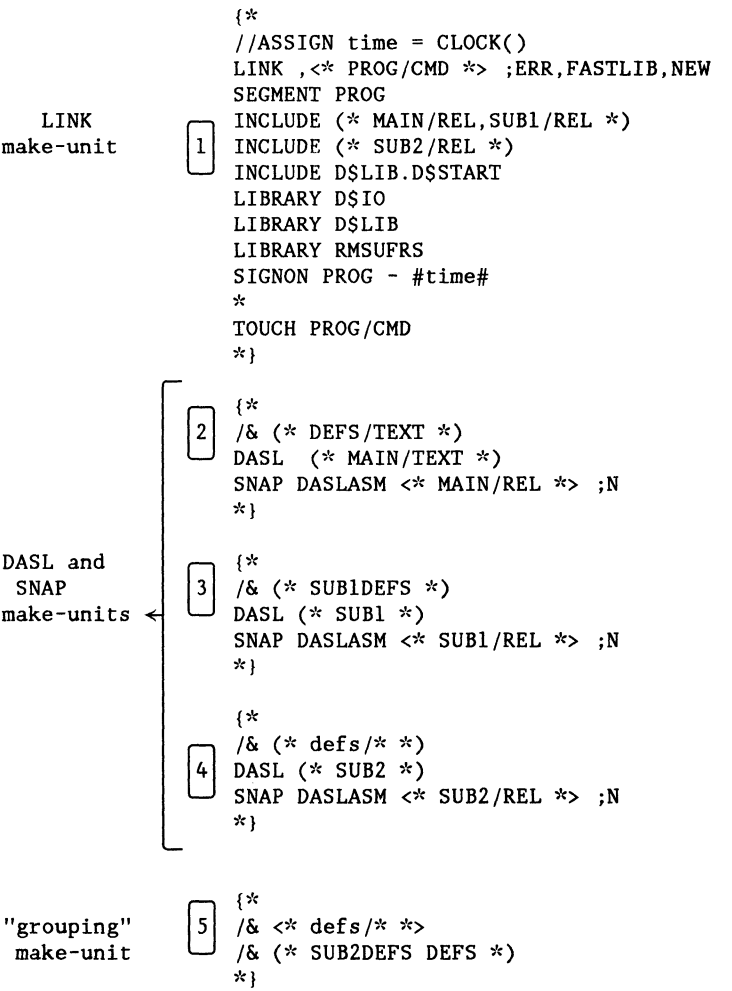
The configuration looks like this:



The Rules file

To use MAKE to keep the command PROG up to date, a user must first create a rules file which describes the dependencies for the system of files.

The following rules file assumes a straightforward LINK structure.



Parts of the rules file

The rules file on the previous page consists of five make-units which are described below.

PART	DESCRIPTION
1 LINK make-unit	Indicates that PROG/CMD • is a dependent of the unit and • has as predecessors the three /REL files
2-4 DASL and SNAP make-units	Indicate that the /REL file in each unit • is a dependent of its DASL unit and • has as predecessors its source file and some set of INCLUDE files
"grouping" make-unit	Uses a "pseudofile," defs/*, to group the two INCLUDEs, SUB2DEFS and DEFS

Using MAKE to update files

Suppose in our example that the user changes SUB2DEFS and wants to make sure that PROG/CMD is up to date.

The user can invoke MAKE, using the rules file that was listed previously. The file PROG/CMD will be the default target since it is a dependent of the first MAKE unit.

MAKE will then create a chain file which takes care of any necessary recompiling and linking.

MAKE chain file

MAKE creates the following chain file if the user changes SUB2DEFS and then invokes MAKE with PROG/CMD as the target.

```
/& defs/*
/& SUB2DEFS DEFS
/& defs/*
DASL SUB2
SNAP DASLASM SUB2/REL
//ASSIGN time = CLOCK()
LINK ,PROG/CMD ;ERR,FASTLIB,NEW
SEGMENT PROG
INCLUDE MAIN/REL,SUB1/REL
INCLUDE SUB2/REL
INCLUDE D$LIB.D$START
LIBRARY D$IO
LIBRARY D$LIB
LIBRARY RMSUFRS
SIGNON PROG - #time#
*
TOUCH PROG/CMD
```

By comparing this chain file with the rules file, you can see that MAKE outputs the make-units labeled 5, 4, and 1 to the chain file.

HOW MAKE DECIDES WHAT TO REMAKE

Introduction

The example on the following pages builds on the previous example of keeping the command PROG up to date.

These pages explain how MAKE applies its set of rules to the MAKE rules file to create the chain file.

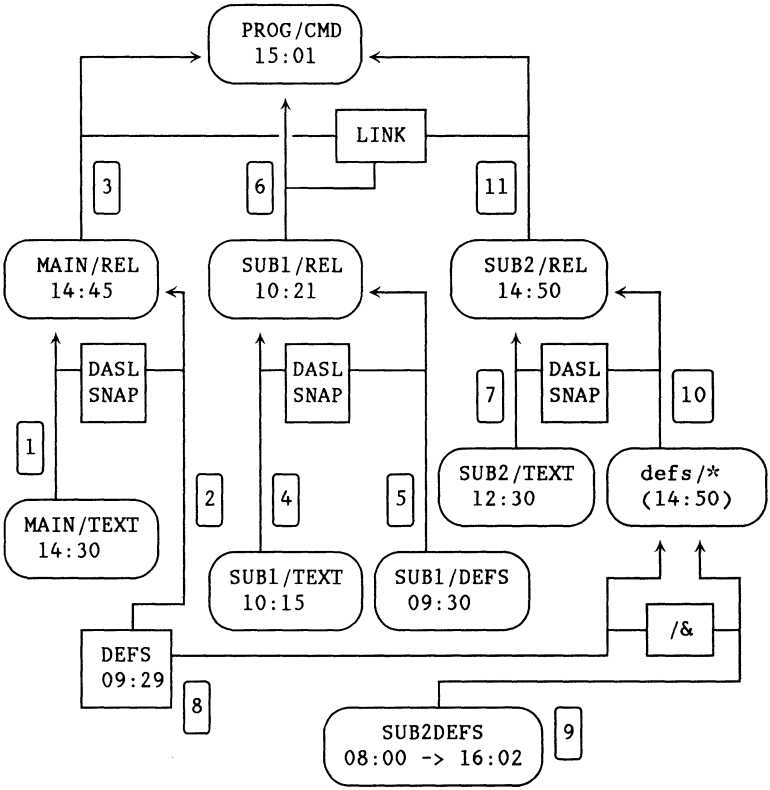
Diagram of file interdependencies

The diagram on the next page illustrates all of the file interdependencies for the system of files on which the command PROG depends.

The following symbols are used in the diagram:

- ovals depict files,
- arrows point from the predecessors of a make-unit to the dependents of the same make-unit,
- labels inside boxes indicate the associated make-units,
- times shown are file times, and
- numbers next to arrows indicate the order in which MAKE compares the file times of files.

Note that the file time of SUB2DEFS changed from 8:00 to 16:02.


PROG/CMD
15:01
LINK
3
6
11
MAIN/REL
14:45
SUB1/REL
10:21
SUB2/REL
14:50
DASL
SNAP
DASL
SNAP
7
DASL
SNAP
10
1
2
4
5
SUB2/TEXT
12:30
defs/*
(14:50)
MAIN/TEXT
14:30
SUB1/TEXT
10:15
SUB1/DEFS
09:30
/&
DEFS
09:29
8
SUB2DEFS
08:00 -> 16:02
9

How MAKE decides what to output

The previous diagram helps to show how MAKE determines what to output to the chain file.

MAKE compares the file times of files in the order indicated by the numbers in the diagram.

MAKE does not find any targets which are out of date until it gets to comparison (9).

The following chart explains why MAKE outputs to the chain file the make-units associated with comparisons (9), (10), and (11).

For each comparison, the following rule applies:
a make-unit is output to the chain file if one of its dependents is a target and one of the target's predecessors is newer that the target.

AT comparison ...	MAKE compares the file times of...	AND outputs the associated make-unit to the chain file because...
9	defs/* and SUB2DEFS	SUB2DEFS is newer than defs/*. Rule: When comparing a pseudofile (defs/*) with one of its predecessors, the pseudofile is considered as old as its oldest dependent which is a target (SUB2/REL).
10	SUB2/REL and defs/*	defs/* is newer than SUB2/REL. Rule: When comparing a pseudofile (defs/*) with one of its dependents, the pseudofile is considered as young as its youngest predecessor (SUB2DEFS).
11	SUB2/REL and PROG/CMD	SUB2/REL is newer than PROG/CMD. Rule: If a real file (SUB2/REL) is to be remade, then it is considered to be the youngest kind of file.

Chapter 13.
TRACE REFERENCE SECTION

OVERVIEW

Purpose

The purpose of TRACE is to provide execution statistics that are useful in the performance analysis and optimization of DASL programs.

Description

The TRACE program interprets loadable object code and provides different kinds of data which indicate where a program spends its time.

Depending on the options given by the user, TRACE

- produces data about the usage of each function in a particular overlay,
- tabulates the number of instructions executed in each 64 bytes of code, and
- traces specified instructions, including CALL, RET, SC, and JUMP.

Space required by TRACE

TRACE requires a logical address space of 64k. The TRACE program and its data reside in the top 8k. This is the region from 56k to 64k of the physical address space.

No program may map into the region from 56k to 64k. An attempt to do so via a $MEMKEY or $MEMMAP system call is trapped. TRACE makes the program being traced "think" the top 8k are not available.

No program may initially load into the top 8k. TRACE treats this as a terminal error.

Speed of TRACE

Since TRACE is actually interpreting the object code, it can run quite slowly. Depending on which options are used, TRACE can run 30 to 500 times slower than normal.

Coming up

The contents of this chapter include:

- Overview of Running TRACE,
- Using TRACE Option DASLMAP,
- Using TRACE Option MAP,
- Using TRACE Options CALL, RET, SC, and JUMP,
- Using TRACE Options SKIP and NOLOAD,
- Using On/Off Facility, and
- Restrictions.

OVERVIEW OF RUNNING TRACE

Introduction

The following pages provide general information on how to run TRACE.

Each of the specific TRACE options is discussed separately later in this section.

Invoking TRACE

To invoke TRACE, enter on the command line:

TRACE [<output file>][;<options>]

If you enter the option DASLMAP, then TRACE will request additional information.

TRACE always prompts for the command line of the program which you wish to trace.

Where TRACE outputs information

To specify where TRACE outputs information, you may

- specify a file name on the command line where TRACE directs all output, or
- use the DISP option to direct the output to the screen.

If you do not specify an output file or use the DISP option, the default output file is TRACEOUT/TEXT:W.

Kinds of TRACE options

TRACE has ten options which are briefly described in the following chart. Several options are incompatible, but many of them can be used in combination with others.

THIS option...	PERFORMS this function...
DASLMAP	produces data about the usage of each function in a particular overlay.
MAP	tabulates the number of instructions executed in each 64 bytes of code.
CALL RET SC JUMP	produces data about each execution of the specified machine level instruction.
DISP	directs output to the screen instead of to the output file.
SKIP = n	causes TRACE to begin tracing after the nth $LOAD system call.
NOLOAD	causes TRACE to stop at the first $LOAD encountered, after tracing has begun, and produce an output file. SKIP and NOLOAD are used together to isolate a single overlay to be traced from the rest of the program.
HELP	gives information about the TRACE options on the screen.

On/Off facility

The On/Off facility of TRACE allows the user to turn tracing on and off by commands embedded in the object code of the program. This facility is discussed later in this chapter on the pages titled USING THE ON/OFF FACILITY.

JSING OPTION DASLMAP

³urpose

The DASLMAP option causes TRACE to produce an output file with information about the use of each function in a particular overlay.

Jse of /SYM1 and /SYM2 files

To gain information about the functions in a program, TRACE needs to access the debugging information files, /SYM1 and /SYM2.

In order to create the /SYM1 and /SYM2 files, you must compile, assemble, and link your program using the DEBUG option. You can accomplish this by running the chain file, DASL/CHN, using either the TRACE or RASL option.

³estrictions on file names

/SYM1 files with names longer than eight characters will not be found. DASL programs take their /SYM1 file names from their source file names. Therefore, you must either

- name your source files with no more than eight characters or
- rename the /SYM1 files before you begin tracing.

/SYM2 file names are truncated to eight characters.

DASLMAP'S output file

The information in DASLMAP's output file is arranged in seven columns. The heading of each column and its meaning is described in the chart below.

COLUMN HEADING	MEANING
usage	A term to describe how TRACE saw the interaction between the functions in your program • element - function never called anything • normal - function called other functions • recurs - function was called recursively • self rec - function called only itself • untraced - for every section of memory with no function assigned to it, TRACE created an untraced "function" • unused - function was never invoked • error - count is incorrect, possibly due to a function which never exited or which was contained in an INCLUDE file

COLUMN HEADING	MEANING
this function's instruction	Total number of machine level instructions executed by the function itself
total calls to function	Total number of times that the function was called either by itself or by any other function
son's + this function's instruction	Sum of the total number of instructions executed by both the function and all of the functions which it called
non-re-ent (top level) calls	Total number of top level entries into the function
function's name	Name of the function
start	Starting address of the function

Example of the output file

The following example shows the TRACE output file for a program with four functions. The untraced functions represent I/O modules for which there were no /SYM1 files.

```
                  total     sons'   nonrecurs
            this  calls     +this       (top
          func.'s     to  func.'s     level)  function's
usage      instr.  func.   instr.      calls  name
element       150     10      150         10  random
normal        162      1      312          1  fillArray
element       121      1      121          1  average
normal         60      1     2412          1  MAIN
untraced        6                            no name given
untraced     1916                            no name given
```

Invoking TRACE with DASLMAP option

If you invoke TRACE with the DASLMAP option, then TRACE will prompt for the names of up to 16 /SYM2 files.

You may enter only the names of /SYM2 files, not the extensions or environments.

Entry terminates upon entry of a blank line or the 16th name.

Isolating a piece of code

If you try to TRACE overlapping overlays at the same time, you will get meaningless results.

In order to isolate a particular overlay for tracing with the DASLMAP option, you may use either

- the SKIP and NOLOAD options or
- the On/Off facility of TRACE.

The On/Off facility may also be used to isolate any particular piece of code.

See the pages in this chapter titled USING TRACE OPTIONS SKIP AND NOLOAD and USING THE ON/OFF FACILITY.

Limitations

The number of functions and untraced "functions" contained in your program cannot exceed 512. This limitation is imposed by the amount of memory which TRACE has available.

Aemory space requirements

With the DASLMAP option, TRACE uses 20k of physical memory, but only uses the top 8k of logical memory. Thus, the DASLMAP option will work on a 128k machine with all programs except those that require a great deal of memory.

USING OPTION MAP

Purpose

The MAP option causes TRACE to produce an output file which contains

- the number of machine level instructions executed in each 64 byte block of logical memory from 0k to 56k and
- a graph of the instruction count for quick comparison.

When to use

The MAP option can be useful in optimizing assembly language code. However, this option is not particularly useful in optimizing DASL code.

Isolating an overlay

If you try to TRACE overlapping overlays at the same time, you will get meaningless results.

You can use the SKIP and NOLOAD options to isolate a particular overlay for tracing with the MAP option. (The On/Off facility does not work with the MAP option.)

See the pages titled USING TRACE OPTIONS SKIP AND NOLOAD later in this chapter.

JSING OPTIONS CALL, RET, SC, & JUMP

³urpose

The options CALL, RET, SC and JUMP each cause TRACE to write a line of information in the output file whenever an instruction with the same name is executed.

Any combination of these four options may be given at the same time.

Vhen to use

The CALL, RET, and JUMP options can be useful in analyzing assembly language programs. However, their use is limited in the performance analysis of DASL programs.

The SC option is somewhat more useful in relation to DASL programs because it shows what files a program uses and how much I/O a program is using.

ʒpeed

As you might expect, CALL, RET, SC and JUMP are the slowest TRACE options.

Information in the output file

The following chart shows the kinds of information listed in the output file for each instruction.

COLUMN HEADING	DESCRIPTION
TOTAL	Running total of instructions executed since tracing was begun
SUBTOT	Number of instructions executed since the last line was printed
FROM	Absolute address of the first byte of the instruction
INST	Name of the instruction All conditional JUMPs, CALLs, and RETs are named simply JUMP, CALL, and RET. A conditional transfer which does not actually cause a transfer will not be printed.
TO	Absolute address to which the instruction passes control In the case of SC instructions, this is either the name of the system call or the number of the system call.
more info	Only used for the system call instruction Additional useful information is given here for $OPENENV and all the $SECR, etc. system calls.

Using options with DISP

The DISP option directs all output which normally goes to the output file to the screen.

If any of the four options (CALL, RET, SC, or JUMP) is given with the DISP option, TRACE will be competing with the traced program for the screen. This can create problems with programs which make considerable use of the screen.

TRACE does not print out instructions executed below 010000 (in the PCR) because of the possible conflict with logging. This conflict occurs when logging is active and the DISP option is given.

Detecting an infinite loop

An effective way to detect an infinite loop is to use these four options (especially JUMP) with the DISP option.

Tracing part of a program

You may use the On/Off facility with CALL, RET, SC, or JUMP to trace a particular part of a program. You may also use the SKIP and NOLOAD options to isolate an overlay.

Another way to trace just the beginning of a long program is to turn on logging and use the DISP option. You can then abort the program and read the information in the log file. This strategy is effective because TRACE does not update the end of file pointer in its output file until it ends.

Caution

The JUMP option has the potential of producing a very large output file.

USING OPTIONS SKIP AND NOLOAD

Purpose

The SKIP and NOLOAD options are used together to isolate a single overlay to be traced from the rest of the program.

Description of SKIP option

The SKIP option must be followed by an integer in the range from 0 to 255.

The value given SKIP indicates the number of $LOAD system calls which will be executed before tracing begins. The initial load which TRACE performs when it loads the root segment is not counted.

Example: Suppose that you want tracing to begin with the second overlay.

The option SKIP=2 causes TRACE to interpret the first overlay, load the second overlay, and then begin tracing.

Description of NOLOAD option

The NOLOAD option causes TRACE to stop at the first $LOAD system call encountered after tracing has begun and produce an output file.

TRACE treats the first $LOAD system call as a $EXIT system call.

Alternative to SKIP and NOLOAD

The On/Off facility can be used instead of the SKIP and NOLOAD options. This facility is described on the following pages.

The advantages of using the SKIP and NOLOAD options are that

- they do not require the program to be remade and
- they allow TRACE to "lie" to the program about how much memory is available.

JSING THE ON/OFF FACILITY

Purpose

The On/Off facility of TRACE allows the user to turn tracing on and off by commands embedded in the object code of the traced program.

When to use

This facility makes it possible to trace just the portion of the program which is of interest and to accelerate the process of collecting information.

Restriction

The On/Off facility does not work with the MAP option or the SKIP option.

Preferred method of using facility

The preferred method of using the On/Off facility involves the following steps.

Turn tracing on just prior to calling the function(s) of interest and turn tracing off after leaving the function(s).

The normal entry mode of TRACE is On. Thus, if you want to trace a portion of code which does not start at the beginning of the program, you can turn TRACE off as soon as the program is entered.

Be sure to turn TRACE back on before the program ends or you will not get any output.

Use of facility with DASLMAP

When using the On/Off facility, some functions may be described as "error" in the output from the DASLMAP option. This reflects the fact that TRACE did not have complete information about the program that it was tracing.

If you turn TRACE on or off within a function, you will get inaccurate results in the following data produced by DASLMAP:

- total number of instructions executed by the function and
- total calls to the function.

Turning TRACE on and off

In order to use the On/Off facility, you must modify and recompile your source code.

TO turn TRACE...	INSERT in source code...
on	CALL 0
off	CALL 0614000

Using DEFINEs

The following three DEFINEs are designed to facilitate the turning of TRACE on and off.

```
DEFINE(D$TRCON,#[IFELSE(D$TRACE,ON,(<^()>0164000)^(),)#1)

DEFINE(D$TRCOFF,#[IFELSE(D$TRACE,ON,(<^()>0)^(),)#1)

DEFINE(D$TRACE,ON)
```

Cautions

Several serious problems are inherent in the implementation of the On/Off facility.

When TRACE is turned off, it is completely turned off. This means that it no longer

- "lies" to the program being traced about the memory above 56k or
- traps $RUN, $LOAD, $EXIT, and $ERROR system calls.

Be sure to turn TRACE on before a loop to determine the available memory space (via $MEMKEY or $MEMMAP system calls) or you will overstore the TRACE program.

RESTRICTIONS

Introduction

There are several restrictions to running TRACE.

Restriction for /SYM1 files

/SYM1 files with names longer than eight characters will not be found.

Restrictions for /SYM2 files

The following are restrictions on /SYM2 files:

- no more than 16 /SYM2 files are allowed,
- /SYM2 file names are truncated to eight characters, and
- extensions of /SYM2 files may not be given.

Limit on running time

All instruction counters and invocation counters are 32 bit numbers. TRACE will run for approximately two days before these numbers start to overflow.

Restrictions for system calls

TRACE is not capable of tracing a $RUN system call. All $RUN system calls are treated as $EXIT system calls.

TRACE does not "lie" to the system about the $SETMIN system call.

$CLOSEAL closes TRACE's output file also and TRACE dies because of it.

Incompatibility with RASL

TRACE and RASL are mutually incompatible.

Chapter 14.
CPUTIME: CPU TIMING PROGRAM

OVERVIEW

Purpose

CPUTIME is a CPU timing program. It is used to measure the amount of time (wall clock time) which a processor devotes to one or more tasks performed by a program.

Coming up

This chapter discusses

- how CPUTIME works and
- the procedure for using CPUTIME.

HOW CPUTIME WORKS

Introduction

CPUTIME is a simple but powerful program. These pages outline the process that CPUTIME uses to determine how much time a processor devotes to a specific task or tasks.

Process

CPUTIME's timing process involves the following three stages.

STAGE	DESCRIPTION
1	CPUTIME begins by calibrating itself so that it knows the speed of the machine on which it is being run. To perform the calibration, CPUTIME times a very specific loop without any other tasks active to find out how long the loop takes. The timing is performed through the wall clock function of $GETIME.
2	CPUTIME times a second (nearly identical) loop with other tasks active. These tasks are selected by the user.
3	CPUTIME calculates and displays the difference in • the time that it takes CPUTIME to perform the loop with other tasks active (Stage 2) and • the expected time it takes CPUTIME to perform the loop alone (known from the calibration in Stage 1). The difference in the two times represents the time it takes another task or tasks to run on the same processor with CPUTIME.

Before you begin

In order to use CPUTIME, you need to be signed on at two workstations which are running on the same processor.

Make sure that no other tasks are actively running on the processor on which you plan to run CPUTIME. If other tasks are running, CPUTIME will be calibrated incorrectly.

Be careful to eliminate "invisible" tasks such as background tasks and timekeeping tasks.

Caution

While you are running CPUTIME, you need to be careful that no one else initiates tasks on the same processor. This can be difficult on a machine which is a file processor or which supports many terminals.

Procedure

Follow this procedure to use CPUTIME.

<table>
<tr><th>STEP</th><th colspan="2">ACTION</th></tr>
<tr><td>1</td><td colspan="2">Invoke CPUTIME at workstation 1 by entering its name on the command line. CPUTIME has no options.</td></tr>
<tr><td colspan="3">Result: CPUTIME displays the following message:
Calibrating. Make sure other tasks are idle.

When CPUTIME is finished calibrating, it displays this message:
Type any key to start, then any key to stop.</td></tr>
<tr><td rowspan="3">2</td><td>IF you want to time a program...</td><td>THEN...</td></tr>
<tr><td>from the beginning</td><td>1) press any key at workstation 1 and

2) begin the program you wish to time at workstation 2.</td></tr>
<tr><td>from some point after the beginning</td><td>1) begin the program at workstation 2 and

2) press any key at workstation 1 when you wish to begin timing.</td></tr>
<tr><td colspan="3">Result: The cursor disappears when you press a key at workstation 1.</td></tr>
</table>

STEP	ACTION
3	To stop timing, press any key at workstation 1. You may stop timing at any point during the execution of the program.
Result: CPUTIME prints how many seconds the processor took to perform the program's tasks and returns to the start message.	
4	To exit from the CPUTIME program, use the ABORT key sequence (DSP-BACKSPACE-DSP).

Chapter 15.
THE INCLUDE FILE D$INC

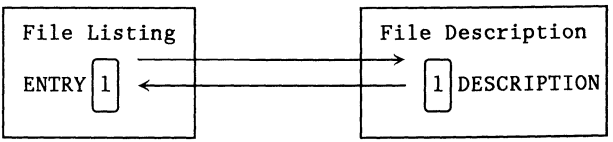
OVERVIEW

Purpose

This section of the DASL Programmer's Guide describes the contents and use of the DASL Standard INCLUDE file (D$INC).

Organization of this chapter

This chapter displays the listing of the D$INC file followed by a description of the file entries. Entries and descriptions are linked with call-out numbers. These numbers indicate which file entries are being described on that page.
Diagram



Coming up

The listing of the D$INC file begins on the next page.

```
DEFINE(MAXINT,077777)
DEFINE(MAXUNSIGNED,0177777)
DEFINE(MAXLONG,017777777777)
DEFINE(NIL,0)

DEFINE(ENUMV,#[IFELSE(#2,,,#[DEFINE(#2,#1)
   ENUMV(INCR(#1),#3,#4,#5,#6,#7,#8,#9)#1)#1)

DEFINE(ENUM,#[DEFINE(#1,0)
   ENUMV(1,#2,#3,#4,#5,#6,#7, #8,#9)#1BYTE)

DEFINE(SETV,#[IFELSE(#2,,,#[DEFINE(#2,(#1))
   SETV(#1*2,#3,#4,#5,#6,#7,#8,#9)#1)#1)

DEFINE(SET,#[SETV(1,#1,#2,#3,#4,#5,#6,#7,#8)#1BYTE)

DEFINE(SETW,#[DEFINE(#1,1)
   SETV(2,#2,#3,#4,#5,#6,#7, #8,#9)#1UNSIGNED)

ENUMV(0,FALSE,TRUE)

TYPDEF ULONG, ILONG STRUCT {
   LSW UNSIGNED;
   MSB BYTE;
   };

EXTERN D$GET24 (pul ^ ULONG) LONG;
EXTERN D$PUT24 (l LONG, pul ^ ULONG);

EXTERN D$MOVE, D$MOVER (S, D ^ BYTE, N UNSIGNED);

EXTERN D$COMP (S, D ^ BYTE, N UNSIGNED) INT;

EXTERN D$INFO () BYTE;

/* SYSTEM INTERFACE */

TYPDEF D$CCODE SET(D$CFLAG,D$ZFLAG,D$SFLAG,D$PFLAG);

TYPDEF D$CALLF ();
```

```
/* Important: reference to the following register variables,
   D$SC, D$CALL, and D$CC should be avoided outside of
   macro definitions in INCLUDE files to preserve
   machine independence.  Reference to the above
   D$xFLAG names is okay. */

EXTERN D$X, D$A, D$B, D$C, D$D, D$E, D$H, D$L BYTE;
EXTERN D$XA, D$BC, D$DE, D$HL UNSIGNED;
EXTERN D$SC (SCNUM BYTE) D$CCODE;
EXTERN D$CALL (F ^ D$CALLF) D$CCODE;
EXTERN D$CC D$CCODE;

/* DEBUGGER */

EXTERN RASLRES$, RASLEND$ () D$CCODE;  10

/* PAGE ALIGNED BUFFERS */  11

EXTERN D$BUF1, D$BUF2, D$BUF3, D$BUF4, D$BUF5
   [256] BYTE;
```

DESCRIPTION OF D$INC FILE ENTRIES

Description

The D$INC INCLUDE file provides standard definitions for DASL programs. The following table contains a description of the contents of the file. In some cases the description covers a group of definitions where it is logical to do so.

SECTION	DESCRIPTION
1	These are the definitions for standard symbolic constants.
2	The recursive macros ENUM and ENUMV are similar except that ENUM defines up to nine arguments as ascending from 0 and has a result of BYTE; ENUMV defines its arguments to be ascending from the first argument and has no result.
3	SET and SETV macros are similar to ENUM and ENUMV macros except that they define arguments in successive powers of two. SETW is similar to SET except it is defined as UNSIGNED.
4	The standard definition for FALSE (0) and TRUE (1).
5	ULONG is a definition for 24 bit values. The D$GET24 and D$PUT24 routines convert between 24 bit ULONG and 32 bit LONG values.

SECTION	DESCRIPTION
6	D$MOVE is an external function that performs a block move of from 0 to 65,535 bytes. It performs a block move from the BYTE address given by the first parameter to the BYTE address given by the second parameter for the number of bytes given by the third parameter. D$MOVER is similar but takes the ending addresses of the blocks.
7	D$COMP performs a block comparison between two strings given by the first two parameters (which are BYTE pointers) for the number of bytes given by the third parameter. If the two strings are equal, the function returns 0, otherwise it returns the difference of the first two differing bytes (byte in first string minus byte in second string).
8	D$INFO is a byte which indicates the type of processor... a 5 indicates an 8600, a 1 indicates a 6600 etc.
9	D$CCODE and D$CALLF are the system interface types. The D$CCODE type value is the condition code returned by the system. The D$CALLF function type is assigned the register values which are passed into the system call from the external variables corresponding to the registers.
9a	These are external variables which correspond to the registers. These variables may be assigned values by the program before the D$SC call.

SECTION	DESCRIPTION
9b	The external function D$SC performs an RMS system call; its argument is the system call number. The function gets the register values to passed into the system call from external variables (see 9a above). It returns the register values to the same variables after the system call.
9c	D$CALL works like D$SC but the argument is the subroutine address of type pointer to D$CALLF.
9d	D$CC receives the condition code value with a bit for each of the possible flags. The masks for these bits are D$CFLAG, D$ZFLAG, D$SFLAG, and D$PFLAG.
10	RASLRES$ and RASLEND$ are the external functions for starting and ending the DASL debugger, RASL.
11	D$BUF1 through D$BUF5 are external 256 byte arrays. These arrays are aligned on memory page boundaries so they may be used as RMS I/O buffers.