INSPECT USERS MANUAL

Beta Release

TANDEM COMPUTERS INCORPORATED
19333 Vallco Parkway
Cupertino, California 95014

Part No. 82315 Preliminary

August 1982
Printed in U.S.A.

PREFACE

This book documents the features of INSPECT, one of Tandem's program development tools.

INSPECT is an optional program. It provides interactive debugging in both source level and machine level modes. It is used for debugging user processes running under the GUARDIAN operating system. INSPECT is also used for debugging terminal-programs running in the PATHWAY environment.

Intended users are system and application programmers.

The organization of this book is as follows:

- Section 1 Introduction provides a product overview.

- Section 2 Running INSPECT describes the methods of invoking INSPECT and briefly describes session features.

- Section 3 INSPECT Commands describes in alphabetic order the high-level commands and summarizes the low-level commands.

- Section 4 COBOL and SCREEN COBOL Dependencies gives language-specific information for COBOL and SCREEN COBOL users of INSPECT.

- Section 5 FORTRAN Dependencies gives language-specific information for FORTRAN users of INSPECT.

- Section 6 TAL Dependencies gives language-specific information for TAL users of INSPECT.

- Section 7 INSPECT Operations describes the system dependencies to provide INSPECT support. (Section to be added.)

- Appendix A Basic Commands describes the basic commands that are included in INSPECT support.

- Appendix B Language Operators lists the operators for each language. (Section to be added.)

- Appendix C INSPECT Error Messages lists the user error messages ar the messages related to system operation. (Section to be added.)

- Appendix D Sample Session. (Section to be added.)

- Appendix E Syntax Summary.

Prerequisites

This book assumes user knowledge of at least one of these languages: COBOL, FORTRAN, or TAL.

Familiarity with writing, compiling, and running programs in the Tandem environment is essential. The following manuals should be available.

- Introduction to Tandem NonStop Computer Systems
- COBOL Programming Manual
- FORTRAN 77 Reference Manual
- PATHWAY Programming Manual
- PATHWAY Operating Manual
- Transaction Application Language Reference Manual
- EDIT Manual
- GUARDIAN Operating System Command Language and Utilities Manual

In addition, these manuals can be helpful.

- Tandem NonStop II System Description Manual
- Tandem NonStop System Description Manual
- BINDER Users Manual
- CROSSREFERENCE Users Manual

<u>CONTENTS</u>

SYNTAX CONVENTIONS IN THIS MANUAL

The following is a summary of the characters and symbols used in the syntax notation in this manual. For distinctiveness, all syntactic elements appear in a typeface different from that of ordinary text.

Notation	Meaning
UPPER-CASE CHARACTERS	Upper-case characters represent keywords and reserved words. If a keyword is optional, it is enclosed in brackets. If a keyword can be abbreviated, the part that can be omitted is enclosed in brackets.
lower-case	Lower-case characters characters represent all variable entries supplied by the user.
Brackets []	Brackets enclose all optional syntactic elements. A group of items enclosed in brackets represents a list of selections from which to choose one or none. The list may be vertically- or horizontally-aligned.
Braces {}	Braces enclose a list of required items from which to choose only one. The list may be vertically- or horizontally-aligned.
Vertical | Bars	A vertical bar separates members of a horizontal list of selections. These lists usually contain a small number of simple elements.
Ellipses ...	When an ellipsis immediately follows an item, that item can be repeated any number of times. of brackets or braces, When an ellipsis immediately follows a list enclosed in brackets or braces, that list can be repeated any number of times.
Punctuation	All punctuation and symbols other than those described above must be entered precisely as shown. If any of the above punctuation appears enclosed in quotation marks, that character is not a syntax descriptor but a required character, and must actually be entered.

SECTION 1

INTRODUCTION

INSPECT is a symbolic debugger. It allows you to control running processes and SCREEN COBOL programs, to examine memory and modify data values -- all with commands that use your source language. Both the NonStop and NonStop II systems support INSPECT; INSPECT recognizes the same source language commands for either system.

Besides the source language commands, INSPECT supports machine level commands to give you maximum debugging flexibility.

SUPPORTED LANGUAGES

INSPECT can control programs written in COBOL, FORTRAN, TAL, and SCREEN COBOL, or in combinations of those languages. The command structure is the same for all supported languages, yet the command elements -- symbolic names and operators -- are expressed in the same syntax as the source language.

THE ENVIRONMENT

Once INSPECT has been installed on a system, INSPECT handles debugging through a system monitor and a cpu monitor. An INSPECT process is associated with a terminal and can control more than one process or program.

In the Command Interpreter environment, the $IMON process monitors cpu activity, creates the cpu monitor process $DMON as required, and creates INSPECT processes. $IMON creates an INSPECT process whenever a debugging event for a process or program in INSPECT-debugging mode occurs. Figure 1-1 shows the INSPECT environment.

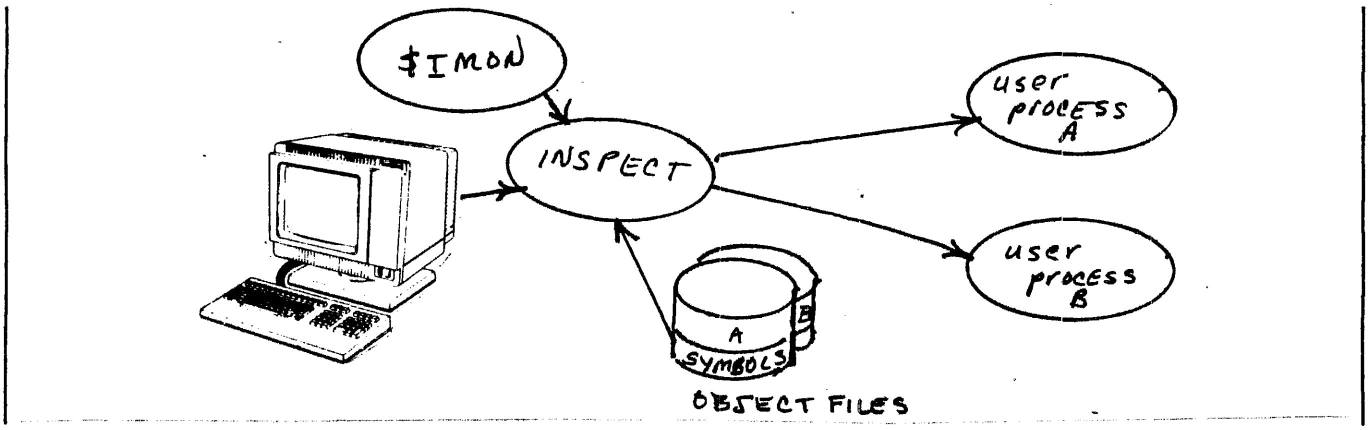

Figure 1-1. The INSPECT Environment Under the Command Interpreter

In the PATHWAY environment, the TCP process communicates with $IMON and the INSPECT process associated with a logical-terminal program. For convenience, it can be desirable to use separate terminals to enter commands for SCREEN COBOL programs and server processes. Figure 1-2 shows the PATHWAY-INSPECT environment.

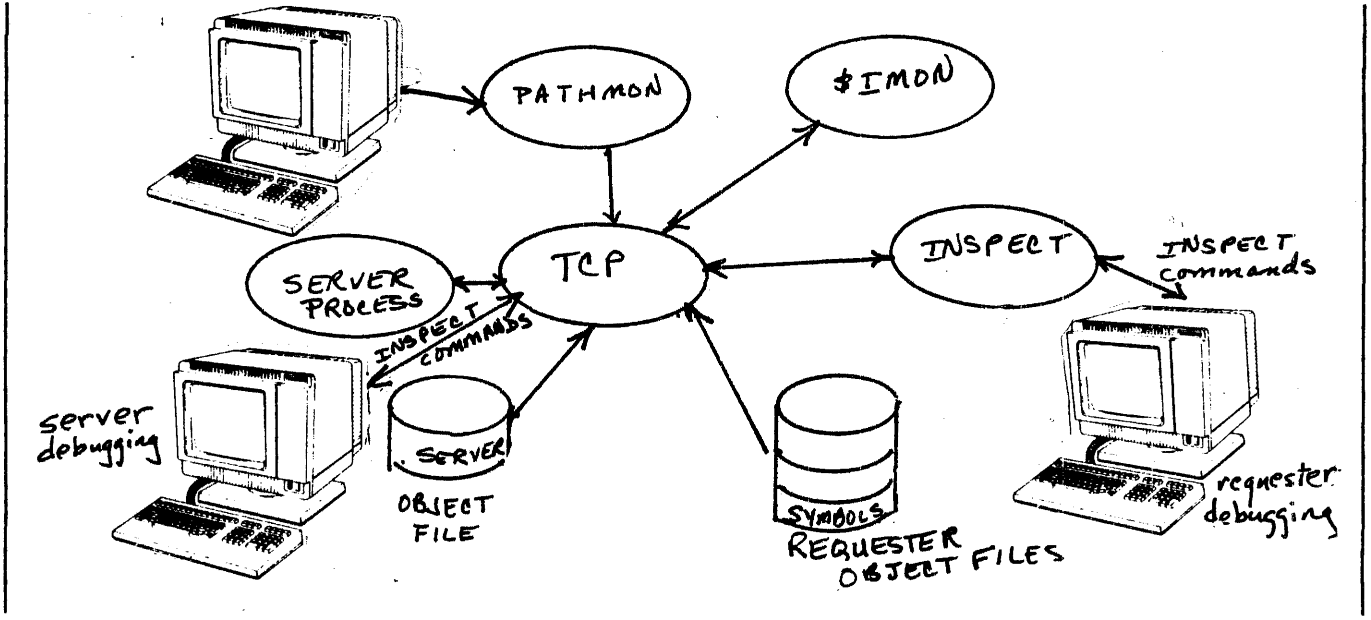

Figure 1-2. The INSPECT Environment Under PATHWAY

DISTRIBUTED DEBUGGING

A single INSPECT process can control or query all the processes involved in an application. The application can be distributed across an EXPAND network.

MODES AND FEATURES

INSPECT has two modes: high-level mode for source-language debugging, and low-level mode for machine-language debugging. You can use either one or both modes in a single session.

Besides display, modify, break, and trace functions, both modes provide basic utilities such as HELP, FC, LOG, /OUT listfile/ for all commands, and OBEY.

High-Level Mode

To use all the features of INSPECT's high-level mode, you must have symbol tables included in the output of compilation.

Special features of high level INSPECT are:

- identifying code and data locations using source language expressions
- stepping through execution
- automatic or user-controlled creation of process environment image in a disc save file
- examining the created save file
- defining names for command strings
- extended break conditions and actions
- formatted displays

Some features, such as program stepping, don't need program symbol information. These can be used by any program.

Low-Level Mode

Low-level mode allows operations that are not possible in high-level mode, like displaying and modifying registers. Low-level mode does not use symbol tables and is available at any time from high-level mode in COBOL, FORTRAN, and TAL. Low-level mode is not available in SCOBOL.

Its use is similar to the GUARDIAN debug facility DEBUG. Therefore, you must understand the system hardware registers and addressing modes to use low-level INSPECT. If symbol tables exist, you can use high-level commands instead of compiler listings for the information you need.

LOW LEVEL -- COMMANDS AND FEATURES. In general, the DEBUG commands are available in low-level INSPECT with some extensions. The commands and syntax are as for NonStop II DEBUG. The LOW command description in Section 3 includes a summary of low-level commands. If you need additional information, refer to the DEBUG Reference Manual for your system.

SECTION 2

RUNNING INSPECT

Requirements for running INSPECT are:

- INSPECT installed on the same system as the home terminal of the process to be debugged

- User read access to the object file of the process or SCREEN COBOL program to be debugged. The program can be running on another system as long as it is accessible across the network. The INSPECT process is created on the local node.

- INSPECT installed on the system where the process or SCREEN COBOL program to be debugged is executing.

Each process has an INSPECT attribute that specifies whether its debugging is to be handled by DEBUG or by INSPECT.

Assuming that INSPECT is available, the steps to start INSPECT debugging depend on the environment of the user's program. In general, when a process starts other processes, the descendants inherit the INSPECT attribute of the parent.

1. If the user specifies INSPECT as the debugger for a Command Interpreter, any processes started by that Command Interpreter automatically have INSPECT as the default debugger.

2. If PATHMON of a PATHWAY system has INSPECT specified as the debugger, server processes under PATHMON automatically have INSPECT as the debugger. INSPECT is the only debugger that can be specified for requester processes.

This section provides an overview of the options for those environments.

THE INSPECT MONITOR PROCESS

If INSPECT is installed on a system, one system process monitors INSPECT requests. A quick way to tell whether INSPECT is installed on your system is to query the status of the INSPECT monitor ($IMON):

```
    STATUS $IMON
```

The Command Interpreter displays the status line for the terminal and the program name if INSPECT is installed. Otherwise, the display informs you that $IMON doesn't exist on that system.

DEBUGGING PROGRAMS AS GUARDIAN PROCESSES

Since DEBUG is the GUARDIAN default debugger, INSPECT must be explicitly chosen for debugging. Three levels of choice are available; choosing INSPECT at any level is sufficient. (The decision is the logical OR.) Essentially, the INSPECT attribute is selected for the program file or for the run environment. The options are:

1. the default for the logon session

2. the RUN option for each process (which can override the session default)

3. the object-file default for the program.

Options 1 and 2 are available regardless of the compiler version used to compile the program. Option 3 is available only with compilers compatible with operating system releases starting with:

- o NonStop Release E05

- o NonStop II Release A04

Compiler Directives

COBOL, FORTRAN, and TAL provide these compiler directives to specify object file debugging characteristics:

1. **?[NO]INSPECT** chooses whether INSPECT or DEBUG is the default debugger for an object file.

2. **?[NO]SAVEABEND** chooses whether a save file for the process environment is created on abnormal termination; SAVEABEND

3. ?[NO]SYMBOLS chooses whether a symbol table is included in the object file for high-level symbolic debugging.

Refer to your language reference manual for details.

BINDER Options

After compilation, the BINDER process can be used in command-driven mode to specify INSPECT for an object file. These SET command options allow respecification of the compiler directives:

1. **INSPECT ON | OFF**

2. **SAVEABEND ON | OFF**

3. **SYMBOLS ON | OFF**

Note that the BINDER cannot include a symbol table after compile time. The SYMBOLS option can be useful to cause BINDER to delete symbol tables that are no longer needed. Using SYMBOLS, the tables still exist in the original object files since BINDER copies the input files to build a new target file. (To delete symbol tables without copying the original file, the BINDER STRIP command can be used.) Refer to the BINDER Users Manual for more information.

Command Interpreter Options

Regardless of an object file's characteristics, INSPECT can be selected at run time for the process. These Command Interpreter facilities are available:

1. **:SET INSPECT ON | OFF | SAVEABEND** specifies whether INSPECT is the default debugger for all programs started by the Command Interpreter. SAVEABEND implies that INSPECT ON is selected. SET INSPECT is effective until respecified or until logoff. At logoff, the default DEBUG is always in effect.

2. **:RUN[D]** program-name / **INSPECT ON | OFF | SAVEABEND** / selects INSPECT as the debugger for that process. SAVEABEND implies that INSPECT ON is selected.

3. **:DEBUG** places the identified process in debug mode. INSPECT must have been previously specified either in the object file or by another Command Interpreter command

Note that the INSPECT attribute is inherited by descendant processes. For example, if the SET INSPECT ON option was specified and a PATHMON is started under that Command Interpreter, INSPECT ON is in effect for programs started by PATHMON.

The GUARDIAN Operating System Command Language and Utilities Manual contains more information on Command Interpreter commands.

A process will inherit the INSPECT/SAVEABEND attribute only if the user has read access to the program file from which the process is created.

DEBUGGING PROGRAMS UNDER A PATHWAY TCP

To use INSPECT in the PATHWAY environment, a TCP must be identified for debugging the requester programs in addition to selecting INSPECT as the debugger for the server processes.

It is convenient to enter INSPECT commands at a a different terminal from the TCP's home terminal, so PATHCOM accepts an optional terminal name for INSPECT. The default terminal for command entry is the TCP's home terminal.

INSPECT can simultaneously control both requesters and servers. Servers can have the INSPECT attribute set for the object file either by the compiler or the BINDER. (See the preceding information in this section.)

SCREEN COBOL Compiler Directive

SCREEN COBOL includes one compiler directive for INSPECT.

?[NO]SYMBOLS specifies whether a symbol table for high-level INSPECT symbolic debugging should be included in the program's object files. Refer to the descriptions of the compiler directives in the PATHWAY Programming Manual. (Since neither save files or the DEBUG program can be used for SCREEN COBOL programs, the INSPECT and SAVEABEND compiler directives for other languages have no meaning.)

PATHCOM Commands

PATHCOM commands that identify TCPs, servers, and terminals for INSPECT communication are:

```
1.    = SET TCP INSPECT ON [ , inspect-terminal-name ] |  OFF

2.    = SET SERVER DEBUG ON |  OFF

3.    = SET TERM INSPECT ON [ , inspect-terminal-name ] | OFF

4.    = INSPECT logical-terminal-name [ , inspect-terminal-name ]
```

where inspect-terminal-name can be other than the TCP's home terminal. (The PATHMON terminal is one example.) The commands shown set the TCP characteristics when the TCP is first added. The characteristics can be respecified by ALTER commands. Refer to the PATHWAY Operating Manual for more information.

Inspecting Servers

PATHWAY servers have a combination of attributes because:

- they are started by PATHMON, and
- they are produced from COBOL, FORTRAN, and TAL source code.

Therefore, the means of specifying INSPECT debugging for servers includes all the following:

- compiler directives (INSPECT, SAVEABEND, and SYMBOLS)
- BINDER commands (SET INSPECT | SAVEABEND | SYMBOLS)
- inheriting the INSPECT attribute from PATHMON (Command Interpreter RUN / INSPECT ON | OFF | SAVEABEND /)
- PATHCOM command (SET SERVER DEBUG ON | OFF)

The SET SERVER DEBUG ON command causes INSPECT or DEBUG to receive control when the server is started. (Note that SET SERVER only results in INSPECT as the debugger if INSPECT was specified in the program file, or if PATHMON was run with INSPECT ON.)

The INSPECT Process

Unlike DEBUG, INSPECT runs as a separate process. The INSPECT process is created automatically when a debug event occurs in a process which has the INSPECT attribute on. The INSPECT process is created with the same home terminal as the process being debugged. A single INSPECT process is created for all processes being debugged with a given home terminal.

Alternatively, an INSPECT process can be run directly from the Command Interpreter. This is typically done to allow analysis of a "save file" created by SAVEABEND or the the INSPECT SAVE command. See the SAVE and PROGRAM command descriptions in Section 3 of this manual. An example of this use of INSPECT is:

```
:INSPECT PR ZZBI0317
```

CONCEPTS

INSPECT's interpretation of your commands relies on these concepts: the execution state of the process or SCREEN COBOL program being debugged and the name scope for command symbols.

The Hold State

Although this discussion concentrates on the hold state, there are three execution states:

1. run state - instructions are executing

2. hold state - execution is suspended

3. stop state - execution has ended, normally or abnormally.

Many INSPECT commands require the hold state. For example, INSPECT can only modify data if execution is suspended for a time.

If INSPECT mode is on for a process, INSPECT receives control and suspends execution for any of these conditions:

- :RUND command; INSPECT receives control before process execution begins

- :DEBUG entered while for a process started by the Command Interpreter. This is the "forced hold". INSPECT displays its header and the current execution status. Then, INSPECT prompts the terminal for commands. (The process identifier on the DEBUG command is required if the Command Interpreter should suspend a process other than the last one started.)

- =INSPECT entered while a TCP program thread (that is, a SCREEN COBOL program) is running. This results in a "forced hold" that suspends the program as described above.

- An illegal operation occurs during execution, and no ARMTRAP routine exists. This results in a "trap hold". INSPECT displays its header and a status line that describes the location and type of trap. Then, INSPECT prompts for commands.

- A CALL DEBUG statement in the source code causes entry to debugging mode. INSPECT displays a header that states the hold is for a call and gives the program location of the call. This facility does not exist for SCOBOL.

- A breakpoint was set for either normal termination (STOP) or abnormal termination (ABEND). The INSPECT header states whether the hold was for a STOP or ABEND and the program location at termination.

- A breakpoint was set for execution of a code location or for access of a data location. The INSPECT header states that the hold is for a BREAK command and displays the breakpoint description as it was entered.

For users who are familiar with DEBUG, INSPECT holds processes for all the conditions that caused the hold state under DEBUG.

Name Scope

When a command refers to a name, or symbol, the resulting action depends on the execution state of the procedure that declared the name. This is the name scope of the symbol. Scope equivalents in different languages are:

- program unit in COBOL or SCREEN COBOL

- program unit (program or subprogram) or common data block in FORTRAN

- procedure or common data block in TAL.

For simplicity, this discussion includes TAL procedures in references to "program unit". Relative to the program's execution, the scope of any program unit is one of:

- the current scope - the program unit that is currently executing

- an active scope - any program unit that was called but not yet exited. (It has a local entry on the data stack.) An active scope need not be the current scope.

- an inactive scope - any program unit that does not have a current call in effect.

INSPECT executes only display-type commands for inactive scopes. Note that you can set breakpoints for inactive scopes.

IDENTIFYING CODE LOCATIONS. The general form for code location expressions is:

```
[ #prog-unit . ] { name        [ OF qual-name ] }
                 {                              }
                 { qual-name [ .  name        ] }

[ { + | - } integer [ code-unit ] ] ...
```

where

#prog-unit indicates the name of a program unit

name is a language-dependent item that is a valid identifier such as an entry-point name, label, or statement number

qual-name is a language-dependent item. It either qualifies name (as in COBOL and SCREEN COBOL OF section-name) or it is qualified by name as in TAL, where name can be a suprocedure and qual-name can be a label. Refer to the language sections in this manual for additional information.

code-unit is a source level construct. It is one of:

STATEMENT	STATEMENTS	S
VERB	VERBS	V
INSTRUCTION	INSTRUCTIONS	I

Scope References. #prog-unit identifies a scope. If the code location is in the current scope, this expression element is optional. Otherwise, the scope reference must be explicit. Examples are:

#scopename - base of program unit scope-name.

#scopename.label - location of label in the program unit.

#scopename.entryname - location of entry point entryname.

#scopename.scopename - location of the primary entry point.

Code Units in Expressions. COBOL and SCREEN COBOL terms are the basis of the STATEMENT and VERB units. The FORTRAN and TAL equivalents of STATEMENT or VERB is the statement.

In COBOL, FORTRAN, and TAL, the INSTRUCTION is a machine-language instruction, which occupies one word. In SCOBOL, the INSTRUCTION unit is one byte.

INDENTIFYING DATA LOCATIONS. Data location expressions refer to any data object in memory. Read-only arrays have the same rules as other data objects.

The COBOL general form for data location is:

```
name [ { OF } ] qualifier-name ... [ index ]
```

Example:

```
FIELD OF SUBREC OF REC ( X, Y, Z )
```

The FORTRAN general form for data location is:

```
qualifier-name [ subscript ] [ ^ name [ subscript ] ] ...
```

Example:

```
REC ( X ) ^ SUBREC ( Y ) ^ FIELD ( Z )
```

The TAL general form for data location is:

```
qualifier-name [ subscript ] [ . name [ subscript ] ] ...
```

Example:

```
REC [ X ] . SUBREC [ Y ] . FIELD [ Z ]
```

THE INSPECT SESSION

When INSPECT gets control the first time, it displays this header.

```
INSPECT - SYMBOLIC DEBUGGER - T9623A00 - (19JUL82) SYSTEM \YOURS
INSPECT P=001174, E=000207
```

The P and E values are the current values for the P-register and the E-register, respectively. The reason for the hold is indicated on this line if the hold was not for initial hold. For example, a CALL DEBUG statement results in this display:

```
INSPECT P=001174, E=000207  - CALL -
```

The program status line is displayed next.

```
*099,07,043*     QUEENOBJ     #QUEENSCO + 6   QUEENSCS[56]
```

Status line information is:

1 *099,07,043* - asterisks indicate the current scope; "099,07,043" is the system and cpu,pin.

2 QUEENOBJ - the object filename; if the process was named, $pname is displayed instead (for example: $QN2).

3 #QUEENSCO + 6 - the current scope and offset; QUEENSCO is a program unit name in this example. "6" is the offset into the code block. In COBOL, FORTRAN, and TAL the offset is in words; in SCOBOL, it is in bytes.

4 QUEENSCS[56] - the name of the source file that yielded the scope and the source line equivalent to the scope offset; this field is only displayed if symbol tables exist for the scope.

The first prompt is the next line displayed. The prompt depends on whether high or low-level command mode is the default for the current scope. (On initial hold, the current scope is the MAIN code unit.) The default mode is high level if symbol tables exist; otherwise, it's low level.

The prompts are:

o hyphen "--" for high-level command mode

```
    -QUEENOBJ-      or      -$NAMEX-
```

o underscore "__" for low-level command mode

```
    _QUEENOBJ_      or      _$NAMEX_
```

The program id, if included in the prompt, identifies the current program. For COBOL, FORTRAN, and TAL named processes, the program id is the process name; otherwise, the program id is the disc file name.

Setting Breakpoints

Often, the first commands issued when a program enters the hold state are to set breakpoints at certain locations in the code area. On NonStop II systems, you can also set a single data area breakpoint.

Breakpoints can refer to active or inactive scopes.

EXAMPLES OF BREAKPOINTS. Using the code location scheme previously discussed, the following are examples of BREAK commands.

1) BREAK #attemptdisplay + 2 VERBS IF display-try > 10

   This BREAK command sets a breakpoint in COBOL program unit ATTEMPTDISPLAY at an offset equivalent to the second verb. The break occurs only if the named variable exceeds 10 in value.

2) BREAK #solve + 3 INSTRUCTIONS

   This shows a high-level command for a TAL procedure compiled without symbol tables. The breakpoint is three words beyond the beginning of the procedure code.

   STATEMENTS is the default unit for code location offsets in high-level mode.

BREAKPOINTS IN LOW-LEVEL. Low-level INSPECT is like DEBUG; however, INSPECT recognizes program-unit names in commands. Labels and secondary entry-point names are not recognized. No units can be specified; instruction words are the assumed units. For example, the following command sets is the low-level equivalent of example 2 above.

```
    B  #solve + 3
```

Break Occurrence

Whenever the conditions specified on BREAK commands occur, INSPECT holds the program's execution and notifies you that the break has occurred. The following example shows a display:

```
-QUEENOBJ-B #SOLVE + 3 I
-QUEENOBJ-R
INSPECT P=001224, E=000000 -BREAKPOINT- #SOLVE + 3 I
*099,07,043*  QUEENOBJ                 #SOLVE + 3
_QUEENOBJ_
```

Then, you can enter commands to analyze the program's status. A RESUME command lets program execution continue.

In the preceding example, INSPECT was in high-level mode when the breakpoint was set. (Note the prompt contains hyphen characters.) Since procedure SOLVE did not have a symbol table, the prompt at the break shows the low-level command mode as indicated by underscore characters.

Displaying Memory

Commonly, displays of code and data memory locations are useful. INSPECT includes extensive display options. For arrays and records, these options are especially helpful.

Examples of DISPLAY commands

1) Assume a FORTRAN declaration "integer a (10,20)":

```
DISPLAY a(2) FORMAT 4I10
DISPLAY a(1,1) FOR 5 WORDS IN OCTAL
DISPLAY (a(1,1) + a (2,2))
```

2) Assume this COBOL fragment:

```
  01 B.
    05    C PIC X(10).
    05    D PIC 99 COMP.

DISPLAY c OF b
DISPLAY b WHOLE
DISPLAY d PIC "99v9"
```

INSPECT FUNCTIONS AND HIGH-LEVEL COMMANDS

INSPECT functions are primarily for program control and memory displays. An important means of program control is setting breakpoints. The following tables give brief descriptions of INSPECT high-level commands related to the major functions. Section 3 describes each command alphabetically.

The commands that control execution in high-level mode include those for breakpoints, program state, and for establishing the program unit for analysis; they are listed in Table 2-1. Tables 2-2 and 2-3 list the commands for memory display and miscellaneous commands for programmer convenience.

Table 2-1. Execution Control Commands

BREAK	B	set a breakpoint
CLEAR	C	cancel a breakpoint
HOLD	H	suspend a program
PAUSE		suspend INSPECT temporarily
PROGRAM	PR	set the current program in a multi-program debugging session
RESUME	R	activate a suspended program
SCOPE		set the current program unit
STEP	ST	run a small part of the program code; if followed by a carriage return, the step repeats
STOP		stop a program
TRACE	T	list the call history of program units

Table 2-2. Memory Display Commands

DISPLAY	D	look at data
MODIFY	M	change data
ATTRIB	AT	describe symbol characteristics
SAVE		save the state of a program
FILES	F	look at file status
TIME		display timestamps for source, object, or save files
RADIX		set a default base for numeric conversion

Table 2-3. Convenience Commands

COMMENT		inserts comments for listings
DEFINE	DEF	define a command or command list
ENV		shows environment settings
EXIT		stops INSPECT
FC		edits or reissues previous command
HELP		displays command information
IF		conditional command
HIGH		switches INSPECT from low to high-level mode
LOG		records session
LOW		switches INSPECT from high to low-level mode
OBEY		names command file
OUT		directs output to listfile

--->

TERM	switch INSPECT's home terminal

In a PATHWAY environment, the TERM command is particularly useful, since INSPECT operates in conversational mode. TERM switches the home terminal of the INSPECT process to a named terminal.

USING THE BREAK KEY

This discussion applies to processes that do not intercept the break event when the break key is pressed.

Ordinarily, INSPECT repeatedly prompts you for commands. INSPECT suspends prompting under one of three conditions:

1. if you press the break key, in which case INSPECT passes control to the Command Interpreter and regains control when you enter the Command Interpreter PAUSE command

2. if the process under INSPECT's control is in the run state

3. if you enter the INSPECT PAUSE command.

When INSPECT has suspended prompting, it monitors the break key. If you then press the break key, INSPECT reports the status line and prompts you for another command. If the process under INSPECT's control is in the run state, the status line will show either an execution location within the process or "SYSTEM CODE" followed by a number (meaning that your user code has invoked system code and not yet regained control). You can then use the HOLD command to put your process into the hold state.

While a process under INSPECT's control is in the run state, INSPECT responds to the break key without placing the process into the hold state. When the break key is pressed, INSPECT displays its header and a single prompt. You can enter CR or a command in response to the prompt. The process is not in the hold state, so command entry may not give the desired results.

INSPECT responds to the break signal only after it has displayed at least one prompt for the session. Entering

```
:DEBUG
```

is one way to manually cause the prompt to occur. (RUND causes the prompt before execution begins.) Then, after a RESUME command reactivates the process, INSPECT can respond continually to the break key (if the inspected process does not intercept the break event.)

Program Status

The status line displayed shows the code location that was executing at the time of the display. If the process is still running, a DISPLAY command entered may show values that do not correspond to the status line display.

Command Entry

If other commands are to be entered after pressing the break key, enter a HOLD command at the prompt. Remember, INSPECT won't suspend the program for break key entry alone. The PR command following the HOLD shows the process state.

STOPPING INSPECT

The INSPECT process stops when you enter either the EXIT command or CTL/Y. If any processes or SCREEN COBOL programs are still under INSPECT's control, they continue running. If any program was in the hold state, it remains suspended. You should use a PR command prior to exiting INSPECT to verify that all programs are in an acceptable state. If a process is left suspended, use either the Command Interpreter STOP or ACTIVATE command for the process. After an ACTIVATE, a subsequent Command Interpreter DEBUG command starts a new INSPECT process.

SET INSPECT COMMAND -- COMMAND INTERPRETER

The INSPECT debugging environment is enabled or disabled by the Command Interpreter SET INSPECT command.

SET INSPECT { ON | OFF | SAVEABEND }

where

OFF

disables the INSPECT environment and causes DEBUG to prompt for input when a program enters the hold state.

ON

enables the INSPECT environment and causes INSPECT to prompt for input when a program enters the hold state.

SAVEABEND

enables the INSPECT environment, causes INSPECT to prompt for input, and automatically creates a save file if the program terminates abnormally.

The Command Interpreter debugging environment determines the default debugging environment for all processes started by the Command Interpreter. You can override this default by using the INSPECT run option on the RUN command. The program file can override the default by means of its INSPECT attribute.

The debugging environment of the Command Interpreter remains in effect until a subsequent SET INSPECT command; that is, the effect of a SET INSPECT ON | SAVEABEND command lasts until entry of a SET INSPECT OFF or until entry of a LOGOFF command. Note that another LOGON without an intervening LOGOFF does not change the effect of the SET INSPECT command.

The INSPECT SAVEABEND command is the same as the INSPECT ON command except that a save file is automatically created if the program terminates abnormally. The save file can then be examined to determine the state of the program when it terminated.

"SET INSPECT" command is currently entered as "INSPECT param". This is to change prior to October Release of GUARDIAN.

RUN Command

The debugging environment for a process being started can be set by an optional parameter in the Command Interpreter RUN command. The debugging environment set by this command is in effect only for the process being started.

RUN[D] / [INSPECT { ON | OFF | SAVEABEND }] /

where

INSPECT OFF

disables the INSPECT environment and causes DEBUG to prompt for input when the program enters the hold state.

INSPECT ON

enables the INSPECT environment and causes INSPECT to prompt for input when the program enters the hold state.

INSPECT SAVEABEND

enables the INSPECT environment, causes INSPECT to prompt for input, and automatically creates a save file if the program terminates abnormally.

The INSPECT ON or INSPECT SAVEABEND option sets the INSPECT environment and the INSPECT OFF option sets the DEBUG environment. The selected option sets the debugging environment only for the process being started and overrides the environment in effect at the home terminal.

The INSPECT SAVEABEND option is the same as the INSPECT ON option except that a save file is automatically created if the program terminates abnormally. The save file can then be examined to determine the state of the program when it terminated.

SECTION 3

INSPECT COMMANDS

This section describes in detail the commands that can be used in high-level mode. Low-level commands are summarized in the description of the LOW command.

Following the command summary in Table 3-1 is the generalized syntax for symbolic references. Sections 4 through 6 describe specifics for your source language.

INSPECT provides basic utility commands in both high and low levels. Appendix A gives the syntax of the utility commands. The utility commands are listed here and summarized in Table 3-1.

ENV
EXIT
FC
HELP
LOG
OBEY
OUT
SYSTEM
VOLUME

Table 3-1. Summary of High-Level Commands

Command	Description
ATTRIB	displays the internal characteristics of code or data items
BREAK	specifies program locations for breaks in execution
CLEAR	cancels breakpoints set by BREAK
COMMENT	enters comments to appear in listings

--->

DEFINE	gives a name to a text string to be used as a macro or as a parameter
DISPLAY	specifies storage items to be displayed
ENV	displays the current settings of INSPECT environment commands
EXIT	stops the INSPECT process
FC	edits or repeats a command line
FILES	displays status of the debugged program's open files
HELP	displays INSPECT commands and syntax
HIGH	reenters high-level command mode from low-level mode
HOLD	suspends execution of one or more processes or programs
IF	sets conditions for command execution
LOG	records session activity
LOW	enters low-level command mode
MODIFY	specifies changes to data values
OBEY	directs INSPECT to read commands from a file
OUT	names the file for output listings
PAUSE	suppresses INSPECT prompts
PROGRAM	for multi-process debugging, specifies the program scope for following commands
RADIX	specifies a default numeric base for command input or displays
RESUME	continues execution of a held program
SAVE	dumps data and status information of a process to a disc file
SCOPE	sets the scope for following commands

--->

STEP	executes the program in increments
STOP	terminates process or program being debugged
SYSTEM	sets the default system for expansion of disc file names
TERM	names a different home terminal for INSPECT, usually different from the debugged program's home terminal
TIME	displays the timestamp of a source file, an object file, or a save file
TRACE	displays caller history (stack) for the current program location
VOLUME	sets the default volume and subvolume for expansion of disc file names

SYMBOLIC REFERENCES

Code and data references are entered as COBOL, FORTRAN, or TAL elements, depending on the source language of the scope containing the element. The command descriptions in this section do not give each possible syntax for these elements.

Instead, language dependent elements are referred to as such in the command descriptions. The sections for each language that follow this section describe language-specific information. The elements are:

- codeloc - expressions that specify a location in the code area
- dataloc - expressions that specify a location in the data area
- expression - numeric expressions
- condition - conditional expressions

ATTRIB Command

The ATTRIB command displays the internal characteristics of one or more code or data locations.

```
         [ # scopename ] [   { # scopename } ]
AT[TRIB] [ codeloc     ] [ , { codeloc     } ] ...
         [ dataloc     ] [   { dataloc     } ]
```

where

scopename

identifies a scope as specified under the SCOPE command.

codeloc

is a language-dependent identifier for a code element. codeloc can be the name of a program or procedure, a primary or secondary entry point, or a label.

dataloc

is a language-dependent identifier for a data element. Elementary and group items are valid.

The default display is the attributes for all code and data items for the current scope.

The ATTRIB command can be used for active and inactive scopes.

Examples. Examples 1 through 5 show displays from a COBOL program unit named ATTEMPTDISPLAY. Assume that it was not the current scope.

```
1)    COMMENT   Set the scope explicitly for following commands.
      SCOPE #ATTEMPTDISPLAY
      ATTRIB attempt
    ATTEMPT: VARIABLE
    TYPE=CHAR, ELEMENT LEN= 8 BITS, UNIT SIZE=16 ELEMENTS
    'L'+ 96S+64
    PARENT=DISPLAY-TRY, CHILD=ATTEMPT-ROW
```

```
2)    AT attemptdisplay
    ATTEMPTDISPLAY:  BLOCK NAME
```

```
3)    AT attempt-row
    ATTEMPT-ROW: VARIABLE
    TYPE=NUM UNSIGN, ELEMENT LEN= 8 BITS, UNIT SIZE = 2 ELEMENTS
    'L'+96S + 64
    [8]                ! dimension info
    PARENT=ATTEMPT

4)    ATTRIB assemble-display
    ASSEMBLE-DISPLAY: LABEL
    OFFSET= 82 BYTES

5)    COMMENT cobol-queens-solution is a scope name
      COMMENT for the main program unit
      ATTRIB #cobol-queens-solution
    COBOL-QUEENS-SOLUTION: PROC
    LOCALS = 5 WORDS
    MAIN, ENTRY OFFSET = 6

6)    AT guardian-err
    GUARDIAN-ERR: VARIABLE
    TYPE=BIN UNSIGN, ELEMENT LEN=16 BITS, UNIT SIZE = 1 ELEMENTS
    'G'+3I

7)    COMMENT #solutiondisplay is a TAL subprocedure
      ATTRIB #solutiondisplay
    solutiondisplay: PROC
    LOCALS = 5 WORDS
    MAIN, ENTRY OFFSET = 6
     PARAMETER 1: COUNTER(REF  )
     PARAMETER 2: SOLUTION(REF  )
```

The following examples show displays resulting from a FORTRAN program named PRICE. Assume it is a NonStop program and that it is the current scope.

```
8)    AT #price
    PRICE: PROC
    NonStop,LOCALS=8 WORDS
    MAIN,ENTRY OFFSET=43

9)    AT gross
    GROSS: VARIABLE
    TYPE+REAL, ELEMENT LEN=32 BITS, UNIT SIZE=1 ELEMENTS
    'L'+%11 +0

10)   AT term
    TERM: NAMED CONST
    TYPE+BIN, ELEMENT LEN=16 BITS, UNIT SIZE=1 ELEMENTS
    VALUE= 4

11)   AT 10
    10: LABEL
    OFFSET=92 BYTES

12)   AT #price.40
    40: LABEL
    OFFSET=530 BYTES
```

BREAK Command

The BREAK command specifies conditions for INSPECT to temporarily suspend execution of the process or SCREEN COBOL program.

On NonStop II Systems any executable instruction or data item in your segments is a valid breakpoint, and each user of a process can set breakpoints without affecting other users of that process.

On NonStop systems executable code instructions are the only valid breakpoints, and any breakpoint set by any user of a process is set for all users of that process.

Process-termination breaks are also available. Termination breaks are not available for SCREEN COBOL programs, however.

INSPECT displays the list of currently defined breakpoints when it receives a BREAK command without parameters.

The time at which you enter a BREAK command will be termed *BREAK definition time* and the time at which INSPECT responds to the process (or SCREEN COBOL program) reaching the executable instruction or selected data item and determines what is then to be done will be termed *BREAK time*.

BREAK OPTIONS. Conditional breaks, prompt suppression, and NonStop backup testing are breakpoint options. Although you can specify the options in any order, the options are not independent of one another. The information following the command syntax describes the option interdependencies.

DATA BREAKPOINT -- NONSTOP II. A single data breakpoint is available for each inspected process. Data breakpoints are not allowed from SCREEN COBOL programs. Subprogram and subprocedure locations (L+ and S-) are valid breakpoints; however, the break becomes invalid on the first exit from the scope. (INSPECT does not delete the breakpoint for you. You must use the CLEAR command.)

```
           [                           [ EVERY integer     ]     ]
           [ { codeloc               } [ TEMP [ integer ]  ]     ]
B[REAK]    [ { dataloc [ READ ]      } [ IF condition      ] ... ] , ...
           [ { ABEND                 } [ THEN action       ]     ]
           [ { STOP                  } [ PAUSE             ]     ]
           [                           [ BACKUP            ]     ]

                                                                  --->
```

where

codeloc

is a language-dependent code location expression; codeloc must be the location of an executable instruction in the user's code. Inactive scopes are valid.

dataloc

is a language-dependent data location expression on NonStop II Systems; dataloc must be the exact word to be monitored in the user's data segment; a single data breakpoint can be active at any time. Data breakpoints are invalid for SCREEN COBOL programs.

READ

specifies that a break should occur on both read access and write access to a data item. Code breakpoints cannot have write access. The default for data is write access only.

ABEND

causes a break when a call to ABEND occurs; not available for SCREEN COBOL programs.

STOP

causes a break when a call to STOP occurs (whether programmatically or not); not available for SCREEN COBOL programs.

EVERY integer

specifies an integer number of times the breakpoint must execute before INSPECT holds execution; the maximum value of integer is 2**31-1; see BREAK COMMAND OPTIONS below for order of processing.

IF condition

specifies that the break occurs only if condition is logically true; condition is a language dependent element; see BREAK COMMAND OPTIONS below for order of processing.

TEMP [integer]

specifies that INSPECT is to delete the breakpoint after it

--->

is reached integer times whether the break occurs or not; a CLEAR command overrides TEMP; the default for integer is 1; see BREAK COMMAND OPTIONS below for order of processing.

THEN action

specifies that action is to occur automatically whenever the break occurs; action is one of:

" command string " containing INSPECT or user commands separated by semicolons

text-name for a command string which was previously defined; see the DEFINE command for creating definitions.

If the last command of action is a RESUME command, no prompting occurs at the break. See BREAK COMMAND OPTIONS below for order of processing.

PAUSE

suppresses the breakpoint report and prompting if this break occurs. See Break Command Options below for order of processing.

BACKUP

specifies the breakpoint is for the backup process of a NonStop process pair.

EXAMPLES. These are simple examples for setting breakpoints.

1) Setting code breakpoints is illustrated by the following COBOL code fragment and the BREAK commands.

```
  SECT SECTION.
  PARA-1.
    MOVE A TO B  MOVE C TO D.
  PARA-2.
    DISPLAY B.
    ADD A TO B.
```

```
 BREAK #SECT.para-1 + 1 V
 SCOPE #SECT
 BREAK para-2 + 1 S
 BREAK para-2 IF a greater b
 B
#SECT PARA-1 + 1 V
#SECT PARA-2 + 1 S
```

```
#SECT para-2 IF A GREATER B
```

2) This example shows a way to set variable xyz to 15 every time a break at location A+4026 I occurs. (I in the expression specifies word instructions. If the expression contains no unit, INSPECT assumes statements.) Assume that the source code was TAL, so the expression in the MODIFY command is in TAL syntax.

```
BREAK a+4026 I THEN "M xyz:=15; RESUME"
```

3) The same effect is accomplished by the following sequence.

```
DEFINE action = "M xyz := 15; RESUME"
BREAK a+4026 I THEN action
```

PRIMARY ENTRY POINTS. You should specify the primary entry point rather than the base of a procedure as a breakpoint. It is possible to destroy initialization data if a breakpoint is placed at the base of a procedure.

```
COMMENT   this example specifies the primary entry point
B [ #proc. ] proc

COMMENT   the next example specifies a label or alternate ep
B [ #proc. ] procl

COMMENT   CAREFUL - this is the procedure base
B #proc
```

BREAK COMMAND OPTIONS.
The options, except BACKUP, control INSPECT's processing at each BREAK time (execution of the breakpoint). BACKUP applies only to the placement of the breakpoint.

To use the break options in combination, you can specify them in any order at BREAK definition time. They are always tested in the following order at BREAK time:

EVERY If EVERY is present, its counter is checked. If the proper count has not been reached, the remaining options are ignored.

IF If IF is present, its condition is evaluated. If the condition is false, the remaining options are ignored.

TEMP If TEMP is present, its counter is checked. If the proper count has been reached, the breakpoint is cleared but option processing continues.

THEN If THEN is present, its action is performed.

EVERY and IF.
The EVERY option can provide control of breakpoints within loops. IF further controls the breakpoint. (You can use either option alone.)

Suppose a loop malfunctioned on the 42nd iteration. You can set a breakpoint specifying EVERY 42. Then, the break does not occur until the 42nd iteration.

```
 SECT-1 SECTION.
 PARA-1.
   MOVE A TO B  MOVE C TO D.
   PERFORM LOOK-LOOP.

 COMMENT   set the breakpoint at the top of the loop
 BREAK para-1 + 3 V EVERY 42 IF c < 1000
```

Note that the IF option is evaluated only if the EVERY condition is satisfied. Whether the break occurs at iteration 42 depends on the value of C. If C is not less than 1000, the process continues to execute without issuing a prompt to the home terminal. INSPECT still checks the break conditions again at iteration 84.

IF Scope. Any data references in a BREAK command inherit the scope in effect at BREAK definition time unless the command explicitly specifies a different scope.

On the other hand, any data references in an expression within the IF option that are not scope-qualified inherit the scope in effect at BREAK time.

In other words if you use RUND to start a FORTRAN (main) program, then set a breakpoint within one of that program's subroutines, if you do not qualify the names mentioned in an IF option on the BREAK command they will implicitly have the scope of the main program. Even if the names exist within both scopes, they will not refer to the same storage locations in data space.

TEMP. Normally, a breakpoint remains in effect until you explicitly delete it via the CLEAR command.

TEMP is useful for setting a breakpoint for a single occurrence. Either TEMP 1 or TEMP gives this effect.

Consider also the breakpoint described above for EVERY and IF. If the breakpoint had both the EVERY 42 and the TEMP 2 options set, INSPECT can automatically delete the breakpoint for you.

```
1)  COMMENT   delete at iteration 84 (break 2)
    BREAK para-1 + 3 V TEMP 2 EVERY 42

2)  COMMENT   delete at iteration 84
    COMMENT   only if c was < 1000 both times
    BREAK para-1 + 3 V EVERY 42 TEMP 2 IF c < 1000.
```

THEN. This option sets the required action to be taken when the breakpoint occurs and the evaluation of the break options reaches the THEN.

One use of THEN is for predefined patching of data. The patch action can be given as a command string on the THEN option or as the text-name of a define string. Define strings CAN include other text-names to invoke other define strings.

NOTE

1. Numeric literals in a THEN option are evaluated when the breakpoint occurs and are interpreted using the radix current at BREAK time. All other numeric literals in a BREAK statement are evaluated at BREAK definition time, when the BREAK command is issued, using the radix current at that time.

2. You cannot patch your code area in an INSPECT session. This applies also to TAL 'P-relative' arrays. The BINDER MODIFY command allows code changes without recompilation. Refer to the BINDER manual for running BINDER via :BIND.

Since RESUME can be the last command in the command string, a known problem can be corrected without further intervention after entering the BREAK command. Note that if a defined text name invokes other text names, any RESUME that occurs within the nested invocations ends the break. Refer to the DEFINE command for more information.

PAUSE. This option is useful when debugging multiple processes. It allows completion of terminal i/o prior to INSPECT's prompting when a break occurs. INSPECT does not report the breakpoint even though the process is in the hold state.

In any case, after INSPECT writes all queued break messages, the process associated with the last message is the current process.

BACKUP. Use BACKUP to test NonStop process pairs. This option specifies that the breakpoint is only for the backup process. Enter a STOP command for the primary to force backup execution.

Changing BREAK Options. Change breakpoint options by reentering the BREAK command with new options. No prior CLEAR is needed.

CLEAR Command

CLEAR cancels one or more breakpoints that are in effect for the current program or process only.

CLEAR with no parameters cancels only the current breakpoint if more than one breakpoint is in the list of defined breakpoints.

```
         [ { codeloc [ , codeloc ] } ]
         [ { dataloc               } ]
C[LEAR]  [ { STOP                  } ] , ...
         [ { ABEND                 } ]
         [                           ]
         [ *                         ]
```

where

codeloc

is a language-dependent expression resulting in the same code location as on a BREAK command

dataloc

is a language-dependent expression resulting in the same data location as on a BREAK command

STOP

cancels the breakpoint at CALL STOP

ABEND

cancels the breakpoint at CALL ABEND

*

cancels all breakpoints for the current process or program

INSPECT does not notify you that the cancel occurred. (Using the BREAK command with no parameters to display breakpoints verifies that the cancel occurred.)

COMMENT Command

Use COMMENT to enter descriptive text to appear in the output listing.

```
COMMENT [ text ]

where

  text

    is a string of characters.
```

If COMMENT is entered on a multi-command line, the COMMENT must be the last command on the line.

DEFINE Command

Use the DEFINE command to:

- enter and name a text string for use in the current INSPECT session
- display all names defined for the session
- delete a single definition

Defined names are allowed wherever INSPECT expects a command name.

DEFINE strings can contain any INSPECT commands (except DEFINE and FC) separated by semcolons. Strings can contain other defined names.

```
DEF[INE] [ text-name = [ "string" ] ]
   [ , text-name = [ "string" ] ] ...
```

where

text-name

is the alphanumeric name of the *"string"*; the first character of the name must be *alphabetic*; the maximum name length is 31 characters; *text-name* cannot be the name of an INSPECT command (that is, *you cannot* redefine INSPECT commands)

Omitting *text-name* results in a display of all defines.

"string"

is the text string to associate with *text-name*; the string must be enclosed in quotation marks; *"string"* is a list of INSPECT commands separated by semicolons. DEFINE and FC are not allowed. *text-names* are allowed within string.

If *"string"* is omitted, *text-name* is deleted from the list of *defines*.

USING DEFINE STRINGS WITH THE BREAK COMMAND. *text-name* of a current DEFINE string is a valid response to an INSPECT prompt. It is also valid to supply *text-name* as the action for the THEN option of the BREAK command. *When the* associated break occurs, INSPECT interprets the commands defined by *"string"*. If *"string"* includes a RESUME command, INSPECT does not *prompt* for input. *Refer* to the description of the BREAK command's THEN option for additional information.

If an error occurs during processing of a DEFINE string, INSPECT displays an error message.

Examples.

```
1)  COMMENT  create multiple defines in a single entry
    COMMENT
    DEFINE qr="d q for 2", pfg="d p,f,g fmt (2i3, f5.3)"
    pfg;qr
     25 9 0.303
     Q= 37 44

2)  COMMENT  the IF command is useful in definitions
    DEFINE oldval="IF x < 99 THEN D('X hack '),X"
    DEF xset="D('X being set to 99');M X=99"
    DEF xhack="oldval;xset"
    COMMENT       entering xhack
    COMMENT       causes execution of oldval and xset
    COMMENT
    xhack
     X hack  X=50
     X being set to 99
    D x
     X=99

3)  COMMENT  display the defines in Example 2
    COMMENT  assume there are no other defines
    COMMENT
    DEFINE
    OLDVAL=  "D('X HACK '),X"
    XSET=  "D('X BEING SET TO 99');M X=99"
    XHACK=  "OLDVAL;XSET"
    COMMENT  delete xhack
    COMMENT
    DEF xhack =
    COMMENT  verify deletion by displaying defines
    DEF
    XSET=  "D('X BEING SET TO 99');M X=99"
    COMMENT  delete xset
    COMMENT
    DEF xset =
    -
    COMMENT   INSPECT displays only the prompt
```

DISPLAY Command

DISPLAY allows inspection of the data or code in user areas. In a single command, all display items must be in a single language.

On NonStop II Systems, you can display user library routines if you have read access to the code file.

DISPLAY command options provide flexibility in displaying records and arrays. For example, you can specify a group size for displaying a field. You can also request formatting of data (as allowed by the source language).

Code display is available if the source langauage is COBOL, FORTRAN, or TAL. (SCREEN COBOL pseudo code cannot be displayed.)

DISPLAY FORMATS. You can use any of these formats for a single item or a list of items:

- default data display, which includes (for each element in a record or group item) the variable name, an equal sign, and the value enclosed in quotation marks
- record or group item displayed without breaks for separate elements -- the WHOLE option
- data values displayed without element names, equal sign, or enclosing quotation marks -- the PLAIN option
- code displayed in ICODE
- items displayed in groups; groups can correspond to storage units, that is, bytes, words, doublewords, or quadwords
- items displayed in one or more of these modes -- ASCII character, based numerics, or icode mnemonics
- items formatted for display using PICTUREs or FORMATTER formats
- expressions evaluated and the results displayed.

D[ISPLAY] item [, item] ... [formatlist]

where

item

is a code or data specifier or an expression

```
{ { codeloc                                 }                  }
{ {                                         } [ spacelist ]    }
{ { dataloc [ WHOLE ] [ PLAIN ]             }                  }
{ "string"                                                     }
{ text-name                                                    }
{ ( expression )                                               }
```

where

codeloc

is a language-dependent expression specifying a code location in the user's area; in SCREEN COBOL codeloc is invalid.

dataloc [WHOLE] [PLAIN]

dataloc is a language-dependent expression specifying a data location in the user's area; the default display is the name of item, an equal sign, and the value; if item is an array or group item, its elements are displayed individually with their names; except for scalars, values are displayed enclosed in quotation marks.

WHOLE causes display of all elements of an array or group item as a single string of contiguous characters.

PLAIN suppresses the display of all names associated with item, the equal sign(s), and the enclosing quotation marks.

"string"

is a string of characters to be displayed.

text-name

is a text-name defined in a DEFINE command.

--->

(expression)

is a sequence of variables, operators, and constants. expression must be in the syntax of the source language for the current scope; limitations are noted in the language sections. expression cannot contain function calls.

spacelist

is the number of groups to display and the storage unit of each group if item is code or data. The default for spacelist depends on the declaration of dataloc; it is 1 word for code.

The FOR keyword identifies a spacelist of the following syntax:

```
FOR [ integer * ] unit-list
```

integer * specifies the number of multiples of unit-list. The default is one.

unit-list has the form

```
{ unit                          }
{ ( unit [ , unit ] ... )       }
```

```
unit is [ integer ] { W[ORD][S]    }
                    { B[YTE][S]    }
                    { D[OUBLE][S]  }
                    { Q[UAD][S]    }
```

spacelist is invalid for SCREEN COBOL.

The default unit is determined by the declaration of the item in the current scope. For example, if an item is declared as a character, the unit default is bytes.

--->

formatlist

is a FORMATTER format, a PICTURE, or a character mode; INSPECT pairs the elements in formatlist with the item list elements; rules for reusing formats for list elements are as defined for the source language; the syntax of formatlist is

```
[ IN base ...                                    ]
[ { FORMAT } formatter-format                    ]
[ { FMT    }                                     ]
[ PIC "mask-format" [ , "mask-format" ] ... ]
```

where

base

is the display mode; if more than one base occurs, INSPECT displays each item in each specified base; bases are:

```
B[INARY]          A[SCII]
D[ECIMAL]         ICODE
O[CTAL]
H[EX]
```

ASCII character display has unprintable characters represented by question marks (?)

ICODE code display as assembler instruction mnemonics

"mask-format"

9, Z, or V FORMATTER M conversion descriptors; "mask-format" must be the last parameter in the command line

formatter-format

any FORMATTER conversion descriptor; it must be the last parameter in the command line

COBOL DISPLAY. Assume that a COBOL program unit contains the following partial structure within an FD clause. Display commands can be entered as shown.

```
    01  address-record.
        05 office-number      PIC X99.
        05 office-address     PIC X(45).
```

```
DISPLAY office-number of address-record
DISPLAY address-record PLAIN
DISPLAY office-number PIC "99999"
```

FORTRAN DISPLAY. Assume the array declaration

```
    INTEGER A (10,20)
```

The following examples display contents of A.

```
DISPLAY A (2) FMT 4I10
DISPLAY A (1,1) FOR 5 WORDS IN OCTAL
DISPLAY (A(1,1) + A (2,2))
```

TAL DISPLAY.

```
 D reply^length
 D #scope.reply^length
```

WHOLE AND PLAIN OPTIONS. These examples show displays of a record, a scalar, and a string. The only options in these examples are PLAIN and WHOLE. As shown, PLAIN and WHOLE can be used together.

```
  DISPLAY rec, num, str
     REC(DATE="082282", AMOUNT="000405"), NUM=1.5,
     STR="calculate income for date"

  D rec WHOLE
    REC="082282000405"

  D str PLAIN
    calculate income for date

  D rec WHOLE PLAIN
    082282000405
```

USING THE <spacelist> OPTION. The spacelist option allows display in the physical order of storage. The general forms of spacelist are:

o FOR [n *] m [unit] - displays n groups of m units

o FOR [n *] (m [unit] [, mn [unit]] ...) - displays n records, each record contains a group of m units, followed by a group of mn units.

The following examples, which show different spacelist forms, are based on these assumptions:

1) S is "ABCDEFGH"
2) ARR is an integer array of eight words
3) ARR's current contents are 0, 1, 0, 2, 0, 3, 0, 4.

Then, the following commands produce the indicated results.

```
D s
 S="ABCDEFGH"

D s FOR 4 * 2
 S="AB" "CD" "EF" "GH"

D s FOR 2 * 3
 S="ABC" "DEF"

D arr FOR 4 DOUBLES
 ARR=1 2 3 4

D arr FOR 8 BYTES IN ASCII
 ARR=?0?0?0?1?0?0?0?2
```

(INSPECT substitutes ?number for unprintable ASCII characters, where "number" is the byte value in decimal.)

```
D s FOR 2*(3b,1b),arr FOR(1d,1d,1d)
 S="ABC" "D" "EFG" "H", ARR=1 2 3
```

FORMATTED DISPLAYS. INSPECT formats data according to the rules established for COBOL PICs and FORTRAN FORMATs. The Tandem FORMATTER is used for FORTRAN FORMATs and can be called from TAL code. INSPECT follows the format rules of FORTRAN and the FORMATTER when formats must be repeated.

Formatted displays are limited to one screen (24 lines) and to 1024 bytes of data. Exceeding these limits results in error conditions.

The following examples require the same assumptions that were identified for "Examples - Grouping Data" above.

```
D arr FOR 12 BYTES IN OCTAL
 ARR=000 000 000 001 000 000 000 002 000 000 000 003

DISPLAY arr FOR 3 WORDS IN HEX
 ARR=0000 0001 0000

DISPLAY s IN OCTAL ASCII
 S=101 102 103 104 105 106 107 108 "ABCDEFGH"
```

```
D arr FMT (8I2)
 0 1 0 2 0 3 0 4

D arr PIC "zz9","zz99"
  0  01  0  02  0  03  0  04
```

Refer to the FORMATTER documentation in the *GUARDIAN Operating System Programming Manual*.

PIC conversion string special characters are supported by the FORMATTER M-type format, i.e. Z, 9, V, and ".".

If the DISPLAY command does not specify a format, INSPECT selects a format which is compatible with the item's declaration. Refer to the description of the RADIX command for INSPECT's selection of radix for output.

CODE DISPLAY. The ICODE keyword requests display of code words as if they were assembler instructions. INSPECT treats the code block as a word array. Depending on the format of your command, INSPECT does bounds checking for the request and returns an error if limits are outside the current code block.

The following examples show two ways to display code in procedure MAIN.

```
D #main + 30 I FOR 8 IN ICODE
D #main IN ICODE
```

INSPECT does not display SCREEN COBOL code locations.

STRING DISPLAY. The DISPLAY command is a valid THEN action on a BREAK command. This is one use of INSPECT's string display feature. The following is an example.

```
  BREAK ABEND THEN "DISPLAY 'program abend',error"
```

STRUCTURED DATA DISPLAY. If the item being displayed is not an element-level item, (i.e, a TAL struct or substruct, a FORTRAN record, or a COBOL group item), the item name and its components are displayed. Components follow the group name and are indented to denote inclusion. The following is an example.

Given a COBOL definition:

```
    01 PERSON-NAME.
       03 LAST-N PICTURE X(20).
       03 REST   PICTURE X(20).
```

```
  D person-name
    PERSON-NAME=
      LAST-N= "SATTERWHITE         "
      REST= "PHILLIP F.          "
```

ARRAY DISPLAY. Items declared as arrays will be displayed using the subscript range specified in the command or, if none is specified, then displayed in their entirety.

A singly-dimensioned item is displayed as a row of values, the name indicating the beginning subscript value. The following is an example.

Given a COBOL definition:

```
01 PERSON-NAME.
   03 LAST-N.
      05 LAST-CHAR PICTURE X OCCURS 20 TIMES.
   03 REST.
      05 FIRST-CHAR PICTURE X OCCURS 20 TIMES.
```

```
D person-name
  PERSON-NAME=
    LAST-N=
      LAST-CHAR[1]= "SATTERWHITE         "
    REST=
      FIRST-CHAR[1]= "PHILLIP F.          "
```

FILES Command

Use the FILES command to query status for files opened by the current process.

Use the syntax of only one language on a single FILES command. You can query files opened by scopes written in different source languages by entering:

- o FILES *
- o FILES with no parameters.

These forms of the command query all files opened by the process.

FILES is invalid for SCREEN COBOL programs.

```
F[ILES] [ { file-list } [ F ] [ DETAIL ] ]
        [ { *         }                  ]
```

where

file-list

identifies one or more open files; all names in the list must agree with the source language of the current scope; each element of file-list is one of:

COBOL	- FD name
FORTRAN	- expression that evaluates to a logical unit number
TAL	- expression that evaluates to a file number

*****

indicates all files

F

indicates that the file-list follows FORTRAN naming conventions; unless F is entered, INSPECT assumes a GUARDIAN file name; valid only with * or file-list

DETAIL

requests listing of the maximum available information for

--->

the queried files; valid only with * or file-list

The default file status listing includes the following information:

```
For all files  - physical file name
               - error number for last file error
```

If DETAIL is specified, the information displayed is similar to the FUP INFO, DETAIL display.

Examples.

1) Either of the following command forms queries all files.

```
FILES
F * DETAIL
```

2) COBOL code block, enter the command in this syntax

```
FILES name-file , address-file , receive-file DETAIL
```

Detail applies to all three files in file-list.

3) The following examples show file-list syntax for FORTRAN expressions. In the first example, nout is a file name.

```
FILES nout
FILES 1,3,5
```

HIGH Command

The HIGH command is entered while in low-level command mode to enter high-level commands.

When a break occurs, INSPECT selects its mode according to whether symbol tables exist for the current scope. If low-level mode is selected, you can use any high-level commands as long as no symbols other than procedure names are entered.

During a high-level break, you can enter low-level mode to do register manipulations, for example. Refer also to the description of the LOW command.

If INSPECT is already in high-level mode, the HIGH command is ignored.

```
HIGH
```

The HIGH command must be the last command on a line.

HOLD Command

The HOLD command suspends a process or a SCREEN COBOL program until a RESUME command is entered.

If any system routines are running on behalf of code to be held, the suspension occurs only after the system code completes and returns to the inspected code. Therefore, when suspending a list of processes (or programs), the order of suspensions can be different from the order specified.

Prompting does not begin again until all suspensions are accomplished.

```
H[OLD] [ process-id [ , process-id ] ... ]
       [ logical-term-name                ]
       [ *                                ]
```

where

process-id

specifies a named or unnamed process; it is one of:

```
{ object-filename                       }
{                                       }
{ [ \system . ] { cpu, pin }            }
{               { $pname   }            }
```

logical-term-name

specifies a SCREEN COBOL program

*****

suspends all processes and programs under INSPECT's control

HOLD with no parameter suspends the current process or program.

CURRENT PROGRAM CONSIDERATIONS. A hold results in a process or program becoming the "current program". In the case of multiple suspensions, the last one suspended becomes the current program. (Status displays show the current program name enclosed in asterisks.)

<u>Example.</u> The INSPECT prompt identifies the current program. Therefore, the prompt changes when the current program changes.

```
TESTB-PR
   PROGRAM ID   NAME     STATE OBJECT: SOURCE
   004,01,044   $P19      RUN  BLOCK+135I: SRCFILE[210]
  *004,03,078* TESTB      RUN  PUTA+104I: TSTBSRC[302]
TESTB-HOLD $p19
  *004,01,044* $P19       HOLD LOOKUP+4I: SRCFILE[1997]
COMMENT
COMMENT   hold causes $P19 to be the current process
COMMENT   it remains in the hold state until a resume command
COMMENT
-$P19-RESUME $p19 AT lookup+9I
  *004,01,044* $P19       RUN  LOOKUP+9I: SRCFILE[1998.3]
```

IF Command

The IF command specifies conditional command execution.

```
IF condition THEN command
```

where

condition

is an expression in language-dependent syntax; condition must be true for INSPECT to perform the given action.

command

is an INSPECT command; DEFINE and FC are not valid; command can be a user-defined text-name.

Example. A primary use of IF is defining a string for conditional execution.

```
DEFINE abcheck="IF a>100 THEN D'a exceeds 100',z;R"
BREAK tax-section THEN "abcheck"
RESUME
  *004,03,078* PROGB      RUN  PUTA+104I: TSTBSRC[302]
 "A EXCEEDS 100", Z=12
 "A EXCEEDS 100", Z=13
COMMENT  abcheck ends with a RESUME; so execution continues
```

LOW Command

The LOW command changes INSPECT's mode from high-level command entry to machine-level debugging. The HIGH command causes entry to high-level mode. (The LOW command is invalid for SCREEN COBOL programs.)

INSPECT's low-level support includes DEBUG commands and additional commands that are not currently available in DEBUG.

INSPECT selects its mode at each break depending on whether symbol tables exist for the current scope.

```
LOW
```

The LOW command must be the last command on a line.

EXAMPLE. This example assumes that a named process, SRVPROC, is the current process. During a break, low level mode is requested to display a stack trace. Note both the differences in the displays and the change in prompt from high to low level.

```
-SRVPROC-LOW
_SRVPROC_B #solve^proc
_SRVPROC_R
INSPECT P=001221, E=000017  -BREAKPOINT- #SOLVE^PROC    #SOLVE +0
*099,02,014* SRVPROC
_SRVPROC_T
000115: 001217 000007 000105             #SOLVE^PROC   + 31
COMMENT         trace output to listfile; no display
_SRVPROC_T / OUT listfile /
COMMENT         return to high level mode.
_SRVPROC_HIGH
-SRVPROC-TRACE
COMMENT
COMMENT  high level trace gives source file name and line #
LANG #PROCEDURE     + OFFSET:  SOURCE
     #SOLVE^PROC    + 0:
TAL  #SOLVOBJ       + 25: SOLVSRC[57]
```

The following list shows the correspondence of INSPECT low-level commands to high-level commands and to DEBUG commands.

High-level commands can be abbreviated to match the low-level commands (except for STOP).

Low-Level Command	Low Level Meaning	High Level Command	DEBUG Operation
A	ASCII display	--	same
B	set breakpoint	B[REAK]	similar
BM	set data breakpoint	B[REAK]	same (II)
C	clear breakpoint	C[LEAR]	same
CM	clear data breakpoint	C[LEAR]	same (II)
D	display	D[ISPLAY]	similar
ENV	display environ option	ENV	--
EXIT	stop INSPECT	EXIT	--
F	file status query	F[ILES]	similar
FC	edit or repeat command	FC	same (II)
HELP	display commands	HELP	--
HIGH	return to high-level	--	--
HOLD		HOLD	--
I	ICODE display	DISPLAY code	--
LOG	log	LOG	--
M	modify	M[ODIFY]	similar
O[BEY]	obey	O[BEY]	--
OUT	set output file	OUT	--
P	pause	P[AUSE]	same
R	resume	R[ESUME]	same
S	stop	STOP	same
SYSTEM	set default system	SYSTEM	--
T	trace	T[RACE]	similar
V	set view segment	--	same
VOLUME	set default volume	VOLUME	--
?	query segment	--	same (II)
=	display expression	DISPLAY	similar

SYNTAX OF LOW-LEVEL COMMANDS. Low-level INSPECT command syntax is based on the syntax of NonStop II DEBUG commands. However, INSPECT allows symbolic references to procedure names in break commands, as in

```
B #procname + nnnn
```

where nnnn is the offset in words from procname. No other symbolic references can be used.

HELP in low-level mode lists the command names and the syntax.

EXPRESSION SYNTAX FOR LOW-LEVEL COMMANDS. INSPECT's low-level expression syntax is based on that of NonStop II DEBUG. The syntax is given here since it applies to all INSPECT users whether NonStop or NonStop II. (The NonStop DEBUG expression syntax is slightly diferent.)

```
value [ { arithmetic-operator value } ... ]

where

  arithmetic-operator

     is one of (in order of precedence):

       *    unsigned multiply
       /    unsigned divide
       <<   left shift
       >>   right shift
       +    unsigned add
       -    unsigned subtract

  value

     has one of these forms:

       ( expression )
       ' ASCII-char ASCII-char
       #code-block-name
       #data-block-name
       number [ . number ]
       register

        number default is positive octal integer; number
        must be:

          [ + | - ] [ # ] integer

             +    is unary plus (positive integer);
                   unary + is optional
             -    is unary minus (negative integer)
             #    indicates a decimal number; integer is octal
                   if # is not used

        register is one of:

           S    R0   R4    RA    RE
           P    R1   R5    RB    RF
           E    R2   R6    RC    RG
           L    R3   R7    RD    RH
```

USING LOW-LEVEL INSPECT. Following are notations of differences between DEBUG and low-level INSPECT. Also included are some facts about INSPECT that can cause surprises.

Default Radix. The high-level radix default is decimal. Low-level default is octal. Be aware of the change if switching between low and high levels.

Code Offset Units. The low-level default for code units is word instructions. However, the high-level default is STATEMENTS. In high-level mode, neglecting to specify INSTRUCTIONS on a command can result in a procedure-bounds error.

= Command. INSPECT supports additional bases as follows:

#	- decimal	C	- code space (proc + offset)
A	- ASCII	UC	- user code space (proc + offset)
B	- binary	UL	- user library (proc + offset)
I	- ICODE	E	- E-register (flags, RP setting)

Default Volume and Subvolume. Both in high and low levels, the default volume and subvolume are from the logon defaults. This is the case even if a Command Interpreter :VOLUME command is in effect.

NonStop II Use. INSPECT does not support the DEBUG ALL parameter when setting breakpoints.

INSPECT does not support display to a device or file.

```
D 0,200,$DEV              DEBUG command (invalid in INSPECT)
D /OUT out-file/ 0,200    INSPECT equivalent
```

MODIFY Command

The MODIFY command specifies changes to user data locations while the process is in the hold state. You can either specify the changes in the command line or as responses to INSPECT's prompts.

```
M[ODIFY] dataloc [ WHOLE ] { =  } [ change [ , change ] ... ]
                           { := }
```

where

dataloc

is a language-dependent expression identifying the data location(s) to be changed; dataloc must describe a contiguous area large enough to hold the change list. dataloc cannot be a read-only array or register.

WHOLE

causes INSPECT to treat the location as a string of contiguous characters, even if dataloc refers to a record or group item; not valid for SCREEN COBOL

change

is the replacement value for the contents of dataloc; change must fit into dataloc; change is language-dependent and has the following syntax:

```
[ integer C[OPIES] ] expression
```

where

integer C[OPIES]

is the number of times to repeat the value resulting from the following expression.

expression

is a language-dependent expression that fits into dataloc; INSPECT converts the data to the type required by the destination fields.

Prompting Sequence. INSPECT prompts for input if the command does not include replacement values or if the area specified by dataloc is larger than the replacement values.

INSPECT displays each element of the data item with its name and current value. Respond by entering the new value or a comma "," to retain the current value.

If the WHOLE option is in effect, INSPECT assumes the entire modification is received at once. No prompting occurs.

Prompting continues until a carriage return indicates no further modifications or until the last field of dataloc is displayed.

NOTE

INSPECT does not prompt for input if the MODIFY command is in:

- an OBEY file
- a DEFINE string
- a command line with more than one command.

Examples.

```
1) MODIFY #cobm.comp in-rec of infile (station, address) =
   "12 West 32 Street"
   DISPLAY modarea PLAIN
     814 12 West 32 Street

2) MODIFY a=5
   MODIFY s="Falcon"
   MODIFY k(1:5,3) := 4,7,9,15,22
   MODIFY k ( 1 : 5, 4 ) := 5 COPIES 0
   DISPLAY ( a, s, k [ 1 : 5, 3 : 4 ] )
     A=5, S="Falcon"
     K[1:3]=4 7 9 15 22
     K[1:4]=0 0 0 0 0
```

```
3) M arr(6:11)
   COMMENT   INSPECT prompts for replacement values
   COMMENT   the display is  "location=value :=    "
   COMMENT
     ARR(6)=49  := ,
     ARR(7)=50  := 37
     ARR(8)=51  := cr
   D arr(6:8)
     ARR(6:8)=49 37 51

4) COMMENT    Assume a TAL proc contains this declaration:
   COMMENT    int arr [0:2] = 'p' := [123, 124, 125];
   COMMENT    the following command produces an error
   COMMENT    because P-relative arrays can't be changed
   M arr[0:2] := 000, 000, 000
```

PAUSE Command

Like the PAUSE option of the BREAK command, the PAUSE command suppresses prompting.

```
PAUSE
```

INSPECT begins prompting after a PAUSE if the process is running and

- the break key is pressed. This assumes the inspected program does not take break.
- the process or program under INSPECT's control hits a breakpoint, trap, or other debug-event which awakens INSPECT.

PROGRAM Command

If debugging more than a single process or SCREEN COBOL program, use PR to establish the "current program" for the following commands. The PROGRAM command also initiates analysis of save files.

```
PR[OGRAM] [ process-id          ] [ , QUIET ]
          [ logical-term-name   ]
          [ save-file-name      ]
```

where

process-id

is one of:

```
{ [ \system. ] { cpu, pin } }
{              { $pname   } }
{ object-file-name          }
```

logical-term-name

is a PATHCOM identifier for a SCREEN COBOL program

save-file-name

specifies that INSPECT should retrieve the named disc file, which contains a save file created by an INSPECT SAVE command or a SAVEABEND option. save-file-name is:

```
save-file-name [ CODE code-file-name ]
               [ LIB lib-file-name   ]
```

where

CODE code-file-name can be different from the one that was used when the save file was created.

LIB lib-file-name overrides the user library file in use when the save file was created.

Omitting the process identifier displays the status of all processes under INSPECT's control.

QUIET

suppresses display of the status line.

If only one process or program is under control of INSPECT, that one is always the current program.

INSPECT displays the status for the named process or program, unless suppressed by the QUIET option.

EXAMPLES.

```
 -xxxx-PR
   PROGRAM ID    NAME     STATE  OBJECT: SOURCE
   004,01,044   $P19       RUN  BLOCK+135I: SRCFILE[210]
  *004,03,078* TESTB      HOLD PUTA+104I: TSTBSRC[302]
```

The asterisks enclose the process identifier of the current machine instruction.

SAVE FILE TIMESTAMPS. When INSPECT retrieves a save file for analysis, warning messages giving timestamp information can occur.

Each object file in the system contains a creation time stamp. When the INSPECT SAVE command creates a save file, it includes this timestamp. If you use the PROGRAM command to fetch a save file, and the recorded timestamp in the save file does not match the timestamp on the corresponding program file on disc, you get a warning message because the program file may have been modified.

Possible reasons for this discrepancy in timestamps are:

1. If operating across network nodes, the timestamps for each system can show a slight deviation.

2. If the object file corresponding to the save file has been recompiled since the save-file creation, the timestamp for the recompilation is shown.

3. On NonStop II Systems, the warning message can occur if a user-library file accessed by the save file has been modified since the save file was written.

For easier debugging, it can be useful to recompile abnormally terminating program code if symbol tables are not in the object file.

RADIX Command

The RADIX command changes the default base for integer displays from decimal to octal or hexadecimal. This default controls the output from DISPLAY commands unless a different option is selected for DISPLAY. The RADIX default also controls display of values from a TRACE command. (Low-level input or output is not affected by the RADIX default.)

Optionally, you can change the default for input.

RADIX { 8 | 10 | 16 } [INPUT]

where

8

sets the default base to octal for numerics

10

sets the default base to decimal for numerics

16

sets the default base to hexadecimal for numerics

INPUT

specifies that all unprefixed numeric input data is treated as the selected base until another RADIX INPUT command is entered.

INSPECT recognizes the following prefixes to specify numerics in a base other than the default for input:

%	octal	%2107
%B	binary	%b0101100
%D	decimal	%d98765
%H	hexadecimal	%h908adbf

No default is available for binary numbers.

Example.

```
1)  COMMENT  default is to be octal displays
    RADIX 8
    DISPLAY r^8
      A= %3252
    COMMENT
    COMMENT  display the values of two doublewords
    D r^a for 2d
      %65240152 %32440000
    COMMENT
    COMMENT enter binary only with prefix %B
    MODIFY data^block = %b0111
    D data^block
     data^block= %7
    COMMENT
    COMMENT  revert to decimal default
    RADIX 10
    COMMENT
    COMMENT  explicit base change leaves default unchanged
    MODIFY data^block := %ha4
    DISPLAY data^block
     data^block= 164
```

RESUME Command

The RESUME command continues the execution of a held process or SCREEN COBOL program. If more than one hold is in effect, you should specify a parameter on the RESUME command.

SCREEN COBOL programs cannot be resumed after a BREAK ABEND or BREAK STOP.

```
R[ESUME] [ process-id [ AT codeloc [ , RP integer ] ] ]
         [ logical-term-name                          ]
         [ *                                          ]
```

where

process-id

is one of:

```
{ [ \system. ] { cpu, pin } }
{              { $pname   } }
{ object-file-name          }
```

The cpu,pin is required if more than one unnamed copy of object-file-name is running on the home terminal.

logical-term-name

identifies a SCREEN COBOL program

AT codeloc

caution -- specifies that code execution is to resume at a location other than the next sequential one; codeloc is a language-dependendent expression that results in the address of an executable instruction within the current scope; using AT requires that you ensure the register pointer (RP) field of the E-register is correct when execution resumes

AT is invalid for SCREEN COBOL.

RP integer

caution -- specifies the value for the RP field of the E-register (to be used with AT codeloc); this subparameter requires machine-level understanding

--->

RP is invalid for SCREEN COBOL.

*

causes all programs currently in the hold state to continue execution from their current locations.

Omitting process identifiers continues execution of the current program if it is suspended.

SAVE Command

The SAVE command dumps the data and status information of the current process. SAVE can only be issued for a process that is already in the hold or stop state. INSPECT writes the environment to a disc file that can be examined during this or a subsequent INSPECT session.

NonStop II System process extended segments are included in the save file.

You initiate analysis of the save file by entering a PR command with the save file name. You can use any high or low-level commands that display code and data values (save file contents). However, commands that assume the running state are unavailable when working with a save file. For example, BREAK is not allowed. The commands you can use with save files are listed below in table 3-2.

If you are examining the save file during an INSPECT session, the STOP command closes the save file. A PR command CAN be used to select a different program for INSPECT control.

If you are examining the save file as the only program in the INSPECT session, STOP closes the file but does not end the session. In this case, you must enter the EXIT command or a <CTRL>-Y.

SAVE is not valid for SCREEN COBOL programs.

SAVE file-name

where

file-name

names the disc file to be created for the dump of the process environment

Table 3-2. INSPECT Commands Applied to Save Files

INSPECT allows these commands for analyzing save files. Also allowed are the basic commands in Appendix A.

ATTRIB
DEFINE
DISPLAY
FILES
HIGH
IF
LOW
PAUSE
PROGRAM
RADIX
SCOPE
STOP
TERM
TRACE
TIME

The following commands assume the process run state; INSPECT issues an error message if they are attempted.

BREAK
CLEAR
HOLD
MODIFY
RESUME
SAVE
STEP

EXAMPLE.

1) Assume process $NAMEX is already in the hold state

```
_$NAMEX_HIGH
-$NAMEX-SAVE savenmx
-$NAMEX-PR namestat
*099,01,029*  NAMEX        HOLD  #PROC^NAME + 80
-$NAMEX-COMMENT display the current programs
-$NAMEX-PR
 099,01,029   NAMEX        HOLD  #PROC^NAME + 80
*099,01,029*  NAMEX        HOLD  #PROC^NAME + 80
-$NAMEX-COMMENT the first status line is the process
-$NAMEX-COMMENT the second status line is the save file
  ...

-$NAMEX-COMMENT  switch back to inspect the process
-$NAMEX-PR $NAMEX
-$NAMEX-RESUME
```

2) This example assumes a previously created save file for $NAMEX.

```
:RUN INSPECT
INSPECT - SYMBOLIC DEBUGGER - (19JUL82)   SYSTEM \ANY
--COMMENT   zzsal630 is the name assigned by INSPECT
--COMMENT   when an save file was created in response
--COMMENT   to a SAVEABEND request on RUN or in object file
--PR zzsal630
-$NAMEX-
  ...
-$NAMEX-COMMENT  STOP closes the save file; INSPECT continues
-$NAMEX-STOP
--EXIT
```

SCOPE Command

Use the SCOPE command to change the scope for subsequent commands. If no name is given, the current scope is displayed.

```
SCOPE [ #procedure [ ( [ + | - ] integer ) ] ]
      [ ##G                                  ]
      [ ##GLOBAL                             ]
      [ data-block                           ]
```

where

#procedure

is one of:

COBOL or SCREEN COBOL program-name
FORTRAN PROGRAM name, SUBROUTINE name, or FUNCTION name
TAL PROC name or BLOCK name (except P-relative arrays)

([+ | -] integer)

specifies an activation of a recursive call. integer is a signed integer constant up to 2**15-1 in absolute value.

##G or ##GLOBAL

is the implicitly named TAL global-data block

data-block

is the name of a FORTRAN or TAL data block in the current process

A procedure name is sufficient to identify the scope if:

- o the procedure is not recursive; that is, the procedure does not directly or indirectly call itself
- o the procedure is recursive but the referenced element (such as a label or data) is not used in recursive calls.

However, to examine an element as it exists during a particular activation, specify the scope using the (integer) option. You can count from either direction using these conventions:

Activation 1 is the least recent activation, that is, the oldest chronologicaly. Positive values count from the base of the stack toward the top.

Activation -1 is the next most recent, that is, the youngest chronologically. Negative values count from the top of the stack toward the base.

The default is the most recent activation (the current scope). Its activation is number is 0.

STEP Command

Use the STEP command to execute a part of the program; the step can be as small as a single machine instruction. After the step, INSPECT accepts a carriage return as a request to execute another step of the same size.

If an interruption occurs while INSPECT is stepping, you must enter another STEP command to continue stepping through the program. For example, if a breakpoint within the step range causes a program hold, INSPECT does not remember the previous STEP command.

```
ST[EP] [ integer ] [ unit ]
```

where

integer

specifies the number of units to execute before the next hold; the default is one.

unit

is a unit of code for the step; it is one of:

```
{ STATEMENT[S]    | S }
{ VERB[S]         | V }
{ INSTRUCTION[S]  | I }
```

where

STATEMENT[S] or S

is the default unit; the source language equivalent is:

```
sentence  -  COBOL, SCREEN COBOL
statement -  FORTRAN, TAL
```

VERB[S] or V

is a COBOL or SCREEN COBOL statement.

INSTRUCTION[S] or I

is a machine instruction for compiled languages; it is one pseudo-operation for SCREEN COBOL.

These are INSPECT's rules for program stepping:

- a unit of code consists of the code from the beginning of one unit up to, but not including, the beginning of the next unit.

- within a single code block, units are counted in logical order of execution. Therefore, a branch to a label causes the branch target to be counted as the next unit.

- called code is stepped over; that is, the next unit after the call is next in physical sequence. For example in COBOL, starting at a PERFORM statement, stepping one statement will advance you through the entire perform range to the statement below the PERFORM statement.

COBOL EXAMPLE. The difference between sentence and statement for COBOL code is shown in this example.

```
  MOVE A-B TO REC-ANY
  DIVIDE DUCAT (REC-ANY) BY C GIVING ANSWER
  SET A-B UP BY 1.

COMMENT  step ends at the SET statement
STEP 2 VERBS
```

EXAMPLE.
Assume process $NAMEX is running; <break> key is pressed.
In this example, source code line numbers are not displayed because no symbol tables are available.

```
*003,09,051*    $NAMEX          RUN     #WRITE^MSG  + 262
_$NAMEX_HOLD
_$NAMEX_HIGH
-$NAMEX-RADIX 8
-$NAMEX-STEP 25 I
INSPECT  P=017617,  E=000313   (STEP)
*003,09,051*    $NAMEX                  #WRITE^MSG  + %367
-$NAMEX-STEP 25 I
INSPECT  P=017661,  E=000317   (STEP)
*003,09,051*    $NAMEX                  #WRITE^MSG  + %431
-$NAMEX-STEP 25 I
INSPECT  P=024043,  E=000221   (STEP)
*003,09,051*    $NAMEX                  #GET^VALUE  + %357
```

STOP Command

Use the STOP command to terminate a Command Interpreter process that is under INSPECT's control. If the INSPECT process was initiated by a Command Interpreter RUN command (:INSPECT...) then INSPECT continues running until an EXIT command or a <CTRL>-Y is entered. If INSPECT was initiated due to a debug event, then the INSPECT process runs until all processes under the control of INSPECT (including the one which incurred the debug event) terminate or until an EXIT command or a <CTRL>-Y is entered.

SCREEN COBOL programs are stopped from PATHCOM; STOP is invalid for SCOBOL programs.

For save files, the STOP command terminates the analysis of the save file. The save program is deleted from the set of programs currently under INSPECT's control.

```
STOP [ process-id ]
```

where

process-id

is the process identifier; usually, it is the same one displayed in the INSPECT prompt. process-id is one of:

```
{ [ \system. ] { cpu, pin } }
{              { $pname   } }
```

See also the HOLD, PROGRAM, and RESUME commands.

The default process is the current program.

EXAMPLES.

```
1)  _$UT_HIGH
    -$UT-COMMENT  hit break key or pause to start second process
    -$UT-<break>
    :RUND object /NAME $UT2/
    -$UT-PAUSE
      .
      .
    -$UT2-COMMENT  stop $UT2 gives prompt for remaining $UT
    -$UT2-STOP
    -$UT- COMMENT  pause or hit break key to start another process
```

TERM Command

The TERM command names the home terminal for INSPECT; INSPECT switches to the named terminal for prompting. The debugged process continues to have its original home terminal.

```
TERM terminal-name

where

  terminal-name

     identifies a terminal for INSPECT's home terminal; it must
     be on the same system as INSPECT
```

TIME Command

The TIME command displays the timestamp for a file. You must have read access to the file. File types and the type of timestamp are:

- o object file -- creation timestamp if created in a BINDER session (via :BIND Command Interpreter command); otherwise, it is the compilation timestamp
- o code block or data block -- compile time
- o save file -- the last modification or creation time of the program file from which the saved process was loaded.

```
TIME [ object-file-name  ]
     [ # code-block-name ]
     [ # data-block-name ]
     [ save-file-name    ]
```

where

object-file-name

is a disc file name of an object file

code-block-name

specifies a code block in the current program

```
COBOL   - program unit name
FORTRAN - program or subprogram name
TAL     - procedure name
```

data-block-name

specifies a data block in the current program

```
FORTRAN - COMMON
TAL     - BLOCK
```

save-file-name

is the disc file name for a dump created for an INSPECT SAVE command

The default is the object-file creation of the current program.

TRACE Command

The TRACE command displays the call history for the current program location. Calls are displayed sequentially from the most recent to the oldest.

Symbolic information for the caller is displayed only if the caller is in the user code or user library area. TAL subprocedure calls are not displayed.

```
T[RACE] [ integer ] [ REG[ISTERS] ] ...
                    [ ARG[UMENTS] ]
```

where

integer

specifies the maximum number of procedure calls to be listed beginning with the current location; by default, all outstanding calls are displayed

REGISTERS

requests listing of E and L registers and P values; ignored for SCREEN COBOL programs

ARGUMENTS

requests listing of formal parameter names and values for each call

Examples

```
1)  T REGISTERS
    LANG #PROCEDURE +OFFSET:   SOURCE (REGISTERS)
     TAL #TALPROC +238I: SRCTWO[20451.906] (L=%2047,RP=7,CCL,T,K)
     FTN #FTNMAIN +726I: SRCONE[66] (L=%603,RP=7,CCG,K,V)

2)  COMMENT no argments or registers are included
    T
    LANG #PROCEDURE +OFFSET:   SOURCE
         #SYSTEM CODE  28743
         #SYSTEM CODE  27316
         #SYSTEM CODE  27517
     CBL #ATTEMPTDISPLAY + 80:   QUEENSCS[1.4]
         #SOLVE + 73:
     CBL #QUEENSCO + 25:   QUEENSCS[57]
```

The E register values are listed in the following mnemonics:

CCE	condition code equal
CCG	condition code greater
CCL	condition code less
T	trap
K	carry
V	overflow

Numbers are displayed in decimal, unless octal or hexadecimal was requested by a RADIX command. Octal or hexadecimal numbers have the % or %H prefix, respectively.

If ARGUMENTS is specified, the formal parameter names and the value for each parameter are listed.

Register values, if requested, are listed first. Argument values, if requested, are then listed.

SECTION 4

COBOL and SCREEN COBOL Dependencies

This section describes the language-dependent elements of INSPECT commands for COBOL and SCREEN COBOL. These language dependencies include:

- code and data location references
- arithmetic and conditional expressions
- format specifications for displays

COBOL and SCREEN COBOL have common syntax rules, and this section treats them together. Differences are noted where they occur.

Forming location identifiers for code and data is similar to the COBOL qualification of names. These location identifiers are called codeloc and dataloc in the command syntax descriptions in Section 3. Forming expressions and formats is also similar to the rules defined for the COBOL or SCREEN COBOL language.

CODE REFERENCES

In high-level mode, INSPECT recognizes symbolic code references, if symbol tables exist. (In either mode, INSPECT interprets program-unit names.) Using the codeloc syntax, you can refer to all levels of code, from the name of the program down to to a particular machine intruction in a program unit. Common uses of codeloc are to:

- set and clear breakpoints
- display code or code attributes

An infrequent use of codeloc is to specify a point to resume execution after a break. (Caution - this use requires GUARDIAN operating system knowledge.)

These are the commands that usually contain code references:

ATTRIB BREAK CLEAR DISPLAY

Examples of codeloc in commands follow this discussion. Note that you cannot use code references on the MODIFY command. (The BINDER program's MODIFY command does accept code references.)

Scope

In COBOL and SCREEN COBOL, scope is equivalent to a program unit.

The scope for any identifier is the program unit that contains that name. A program can contain a single program unit or be composed of multiple code and data blocks. If a program unit has a LINKAGE-SECTION, identifiers common to a caller and a called program unit are included in both scopes. Refer to your compiler manual for more information.

Section 2 discussed name scope rules expected by INSPECT.

Breakpoint Locations

INSPECT allows you to set breaks in the Procedure Division of COBOL and SCREEN COBOL program units by specifying:

- the unqualified name of a program unit, a section, or a paragraph; this causes a break at the beginning of the unit, section, or paragraph

- the name and an offset within a program unit, section, or paragraph; offsets are given as a number of sentences, verbs (statements), or instruction words.

Inactive SCREEN COBOL programs can have only one break set; that break must be at the beginning of the program.

Code Location (CODELOC) Syntax

In both high- and low-level modes, INSPECT allows you to specify code locations by means of expressions. A codeloc must include scope-name qualification if outside the current scope. INSPECT assumes a name is in the current scope if the scope is not explicitly stated.

Name qualification rules are similar in COBOL and SCREEN COBOL.

```
[ #prog-unit . ] name [ OF qualifier-name ] ...

[ { + | - } integer [ code-unit ] ] ...
```

where

#prog-unit

explicitly identifies the scope of name (the program-unit in which name is defined). prog-unit is the source code name of the COBOL or SCREEN COBOL program unit; #prog-unit must be specified if the code location is not in the current scope.

name

is a COBOL or SCREEN COBOL program-unit name, section name, or paragraph name.

OF qualifier-name

is a qualifier that contains name or the previous qualifier-name; it can be a section name.

+ | - integer

is the count of code-units backward (-) or forward (+) from name that determines the offset of the code location.

code-unit

is the language element used to specify the offset of the code location from name; it is one of:

STATEMENT	STATEMENTS	S
VERB	VERBS	V
INSTRUCTION	INSTRUCTIONS	I

--->

where

STATEMENT[S] or S

indicates COBOL or SCREEN COBOL sentences

VERB[S] or V

indicates COBOL or SCREEN COBOL statements

INSTRUCTION[S] or I

COBOL - indicates word instructions (machine code)

SCREEN COBOL - indicates a byte

Default code-unit is STATEMENT.

For a COBOL program named COBMAIN with a section named CUST-DATA containing paragraphs named ADDRESSES and DATES, examples of code location expressions are:

Implicit Reference	Explicit Reference
COBMAIN	#COBMAIN.COBMAIN
CUST-DATA	#COBMAIN.CUST-DATA
DATES	#COBMAIN.DATES
ADDRESSES OF CUST-DATA	#COBMAIN.ADDRESSES OF CUST-DATA
ADDRESSES + 2 VERBS	#COBMAIN.ADDRESSES + 2 VERBS

DATA REFERENCES

Each data location referenced in a COBOL or SCREEN COBOL procedure can be any data item defined in the program unit. The locations can be anywhere in the Data Division of a currently active scope. SCREEN COBOL screen section items cannot be displayed.

Data can be displayed or modified. You can display the contents of the location or the characteristics of the contents. Characteristics include data type, number of elements, and length of element. For records, you can determine the family relationships of different elements.

COBOL-compiled processes running on a NonStop II System can also set and clear a single breakpoint in the data area. Since INSPECT only monitors the first word of the data location at a breakpoint, you must specify the exact word location desired for the breakpoint. (Data breakpoints are not available to either SCREEN COBOL programs or to COBOL programs in a NonStop system.)

Subscripts are used to refer to a specific item or range of items in a table.

Data Location (DATALOC) Syntax

The syntax of the data location expression in COBOL and SCREEN COBOL is shown here.

```
name [ OF qualifier ] ...

   [ {  (  } sub-item [ , sub-item ] ... {  )  } ]
   [ { "[" }                             { "]" } ]
```

where

name

is any COBOL or SCREEN COBOL identifier that is a data item defined in the source program.

qualifier

makes name unique.

sub-item

is a numeric expression that evaluates to a subscript, or two numeric expressions that are separated by a colon (:) and that evaluate to a subscript range.

A data location expression cannot include mnemonic names. A very complex condition might not be fully represented; an error message is displayed when this type of condition is referenced. An index item is interpreted as having its internal value.

Subscripts

Subscripting in a data location expression is the same as subscripting in COBOL and SCREEN COBOL. For example, the following DDL structure requires three subscripts when referencing an occurrence of the data item SUBFIELD:

```
RECORD REC.
  03 SUBREC OCCURS ...
     05 FIELD OCCURS ...
        07 SUBFIELD OCCURS ...
```

If the name is fully qualified, an implicit reference would be entered as follows:

```
SUBFIELD OF FIELD OF SUBREC OF REC (X, Y, Z)
```

The equivalent explicit reference is entered as follows:

```
#COBPROC.SUBFIELD OF FIELD OF SUBREC OF REC (X, Y, Z)
```

In each of these examples, the subscripts X, Y, and Z refer to SUBREC, FIELD, and SUBFIELD respectively.

EXPRESSIONS

INSPECT interprets arithmetic and conditional expressions according to COBOL and SCREEN COBOL syntax rules. Refer to the appropriate language manual for details of the rules. For your convenience, a summary of operators for all languages is given in Appendix B.

Numeric Expressions

Where expression or condition is part of the syntax, a numeric expression or a conditional expression, respectively, is required. Expressions are formed according to COBOL and SCREEN COBOL rules for syntax. In SCREEN COBOL, exponentiation is not valid.

INDEX items can only be used as indexes.

DISPLAY FORMAT SPECIFIERS

INSPECT allows PIC mask-formats to specify the display of data. The mask-formats are a subset of those defined for the COBOL language.

SECTION 5

FORTRAN Language Dependencies

Included in this section are rules for forming code and data location expressions for INSPECT commands.

Numeric and conditional expressions for INSPECT commands follow the established FORTRAN syntax. FORMATTER formats are also as expected (for the DISPLAY command).

CODE LOCATIONS

Symbolic code locations in FORTRAN can be names of program units or statement numbers. The location expressions can evaluate to an offset of statements or machine instructions from the procedure name or statement number. The syntax of the code location expression in FORTRAN is:

```
[ #prog-unit . ] name [ { + | - } integer [ code-unit ] ] ...

where

  #prog-unit

     is a program or subprogram

  name

     is a subprogram, or statement number

  + | - integer

     is the count of code-units backward (-) or forward (+) from
     name that determines the offset of the code location
```

--->

code-unit

is the language element used to specify the offset of the code location from <u>name</u>; it is one of:

STATEMENT	**STATEMENTS**	**S**
VERB	**VERBS**	**V**
INSTRUCTION	**INSTRUCTIONS**	**I**

where

STATEMENT[S] or **S**

indicates a number of FORTRAN statements

VERB[S] or **V**

indicates a number of FORTRAN statements (same meaning as STATEMENT)

INSTRUCTION[S] or **I**

indicates a number of machine instructions in the FORTRAN procedure.

The default <u>code-unit</u> is S (FORTRAN statements).

For a FORTRAN subroutine named FTNSUB with a statement number of 23, examples of code location expressions are as follows:

```
#FTNSUB.FTNSUB
#FTNSUB.23
#FTNSUB.23 - 1 I                    <--- minus 1 instruction
#FTNSUB.23 - I                      <--- minus 1 instruction
#FTNSUB.23 + 4 STATEMENTS + 2 INSTRUCTIONS
```

DATA LOCATIONS

Expressions for data locations are based on the source language declarations. The syntax is:

```
[ #prog-unit . ] name [ subscript ] [ ^ name [ subscript ] ] ...

where

  prog-unit

     is a FORTRAN program or subprogram

   name

     is any data object (including code location names);
     qualifying names precede the qualified names;

   subscript

     is a numeric expression that evaluates to a subscript, or
     two numeric expressions that are separated by a colon (:)
     and that evaluate to a subscript range.
```

Subscripting in a data location expression is the same as subscripting in FORTRAN. For example, the following DDL structure requires three subscripts when referencing an occurrence of the data item SUBFIELD:

```
RECORD REC.
  03 SUBREC OCCURS ...
     05 FIELD OCCURS ...
        07 SUBFIELD OCCURS ...
```

If the name is fully qualified, an expression with implicitly stated scope is as follows:

```
REC^SUBREC(X)^FIELD(Y)^SUBFIELD(Z)
```

The equivalent expression with explicit scope is:

```
#FTNPROC.REC^SUBREC(X)^FIELD(Y)^SUBFIELD(Z)
```

FORTRAN multi-dimensional arrays are specified according to FORTRAN syntax, with the most significant subscript appearing first.

```
DIMENSION A(10,20)
COMMENT display first and last elements of A
D A(1,1), A(10,20)
```

EXPRESSIONS

INSPECT allows the TAL-type REAL(64) constants in the syntax for FORTRAN scopes. That is, the L instead of D precedes the exponent.

SECTION 6

TAL Language Dependencies

Code locations in a TAL procedure can be the procedure name, a subprocedure name, or a label name. Offsets from the base of the procedure can be given in macine instructions or in statements. The syntax of the code location expression is:

```
[ #proc . ] name [ . name1 ] [ { + | - } int [ code-unit ] ]
```

where

#proc

is a procedure name

name

is a procedure name, subprocedure name, or label

name1

is a subprocedure label

+ | - integer

specifies an activation of the scope; refer to the SCOPE command for INSPECT's interpretation

code-unit

is the language element used to specify the offset of the code location from the procedure base; it is one of:

S	STATEMENT	STATEMENTS
I	INSTRUCTION	INSTRUCTIONS
V	VERB	VERBS

--->

where

STATEMENT[S] or S

is determined by statement beginners. This is the default unit.

INSTRUCTION[S] or I

indicates machine instructions (words)

VERB[S] or V

the same as STATEMENT

Use caution in specifying the procedure base as a breakpoint; this can result in destruction of data.

For a TAL procedure named TALPROC that contains label TAL-LABEL and subprocedure TALSUB, which in turn contains label TAL-SUBLABEL, examples of code location expressions are:

	Interpretation
#TALPROC	#TALPROC.TALPROC
TALSUB	#TALPROC.TALSUB
#TALPROC.TALSUB	
#TALPROC.TALSUB + 5 INSTRUCTIONS	
TAL^SUBLABEL	#TALPROC.TAL^SUBLABEL
TALSUB.TAL^SUBLABEL	#TALPROC.TALSUB.TAL^SUBLABEL
#TALPROC.TALSUB.TAL^SUBLABEL	
TAL^LABEL	#TALPROC.TAL^LABEL
#TAL^PROC.TAL^LABEL	

DATA LOCATION EXPRESSION

A data location expression references a data object in memory. This type of expression is identified as the dataloc parameter in INSPECT commands.

The data location referenced in a TAL procedure is any data object that is defined in the procedure. Subscripts are used to reference a specific item or range of items in an array. The syntax of the data location expression in TAL format is:

```
name [ {  (  } sub-item {  )  } ]
     [ { "[" }          { "]" } ]

   [ . name [ {  (  } sub-item {  )  } ] ...
   [        [ { "[" }          { "]" } ]
```

where

name

is any TAL identifier that is a data object defined in the source program.

sub-item

is a numeric expression that evaluates to a subscript, or two numeric expressions that are separated by a colon (:) and that evaluate to a range of subscripts.

An index register defined by the USE statement cannot be specified as a data object. An identifier described by a DEFINE declaration is not automatically expanded; however, the defined text is displayed when the identifier is specified in the ATTRIB command. Variable and argument values in a subprocedure can be accessed only if the subprocedure is active and if the stack pointer has not been modified by the subprocedure or by any subprocedure called by that subprocedure.

Subscripting in a data location expression is the same as subscripting in TAL. For example, the following DDL structure requires three subscripts when referencing an occurrence of the data item FIELD:

```
RECORD REC.
  03 SUBREC OCCURS ...
     05 FIELD OCCURS ...
        07 SUBFIELD OCCURS ...
```

If the name is fully qualified, an implicit reference would be entered as follows:

```
REC.SUBREC[X].FIELD[Y].SUBFIELD[Z]
```

The equivalent explicit reference is entered as follows:

```
#TALPROC.REC.SUBREC[X].FIELD[Y].SUBFIELD[Z]
```

EXPRESSIONS

INSPECT allows FORTRAN REAL(64) constants in commands referring to TAL scopes; that is, the D precedes the exponent. (It also allows TAL's REAL(64) syntax in FORTRAN scopes.)

Expressions of the IF ... THEN ... ELSE form are not allowed; that is, you cannot use "M X := IF Y>5 THEN 12 ELSE 24".

Hexadecimal doubleword and fixed constants must end with %D and %F, respectively. (This avoids confusion in INSPECT's interpretation since D and F are also hexadecimal digits.) INSPECT does not allow all delimiters possible in the TAL language.

Integer constants do not need the trailing D or F.

USAGE

The STEP command requires caution if CASE and FOR loops are in the path. The CASE illustrates the need for caution. Following the step for the CASE statement beginner, INSPECT steps through the OTHERWISE, then one of the cases. Steps through the cases continue until exit from the CASE code.

NEWPROCESS PROCEDURE

A process can be started by a call to the NEWPROCESS procedure. The debugging environment for a process started in this manner is the debugging environment of the caller of NEWPROCESS.

APPENDIX A

BASIC COMMANDS

INSPECT supports basic commands and automatic file name expansion in both high- and low-level modes. This Appendix describes the commands in alphabetic order.

ENV	FC	LOG	OUT	VOLUME
EXIT	HELP	OBEY	SYSTEM	

File Name Expansion

INSPECT assumes that file names supplied for input and output follow GUARDIAN naming conventions. Defaults are supplied by the Command Interpreter when INSPECT is started.

File names are assigned to all disc files and devices. Running processes can be named at your discretion. Refer to the GUARDIAN Operating System Programming Manual for details.

EXPANDED DISC FILE NAMES. Disc files of any type are identified, and located, via the expanded file name. File name expansion assumes the following:

\system name	identifies a system within a network
$volume name	identifies a physical disc pack mounted on a disc drive
subvolume name	identifies a group of related files defined by the user
disc file name	identifies a single file in the subvolume

A fully expanded disc file name has the form:

```
\system-name.$volume-name.subvolume-name.disc-file-name
```

If only a partial file name is supplied as a command parameter, the file name is expanded into the full four-part file name for internal representation.

To guarantee correct file name expansion, at least the disc-file-name must be supplied.

PROCESS AND DEVICE NAMES. Each process and each device, such as a tape drive or printer, is identified in a similar manner. For example:

```
   \YOUR.$TAPE1
```

might specify a particular tape drive on system \YOUR; when operations are already on that system, only $TAPE1 is required.

ENV Command

The ENV command displays the current settings of program environment parameters. In addition to the GUARDIAN ENV options, INSPECT displays the RADIX defaults for input and output. The command with no parameters displays the settings for all ENV options.

```
      [ LOG      ]
  ENV [ RADIX    ]
      [ SYSTEM   ]
      [ VOLUME   ]
```

EXIT Command

The EXIT command stops the INSPECT process for the home terminal. Debugged processes are not stopped.

```
  EXIT
```

Entering <CTL>-Y also stops INSPECT immediately.

Since debugged programs are not stopped, you should ensure that no program is left suspended when EXIT is issued. (Use either :ACTIVATE or :STOP for a suspended process.)

FC Command

The FC command operates the same as the Command Interpreter FC. It allows editing and repetition of the last command line.

```
FC
```

When this command executes, it displays the previous command line up to 132 characters and prompts for editing input with a period (.).

HELP Command

The HELP command displays INSPECT commands and syntax depending on whether the current mode is high- or low-level. For low-level INSPECT, this is useful to determine differences between INSPECT and DEBUG commands.

```
HELP [ command-name  ]
     [ "<" param ">" ]

where

  command-name

     is a valid command name for the current INSPECT mode (high
     or low); if no command-name occurs, INSPECT displays the
     names of all commands valid for the current mode

  param

     is a command parameter
```

LOG Command

The LOG command records the session input and output on a permanent file.

```
LOG { TO file-name }
    { STOP         }
```

where

file-name

identifies a file to receive the copy of commands and output; if the file does not exist, a disc file is created with file-name

Logging is initiated when the command specifies a file name. If logging is already in progress, the previous log file is closed and logging begins on the new file. If file-name is the same as the previous log file, the LOG command is ignored and logging continues on the same file.

If file-name has the form of a disc file and the file does not exist, an EDIT file is created. If the named file is an existing disc file, the output is appended to the file.

The current log file is closed and all logging is stopped when the LOG STOP command is entered.

OBEY Command

The OBEY command causes commands to be read from a specified file.

```
OBEY file-name

where

  disc-file-name

     is the file name of an OBEY file
```

Commands are read from the named file and processed until an end-of-file is encountered. At end-of-file the OBEY file is closed and command input reverts to the previous input file, normally the home terminal.

Additional OBEY commands can appear within an OBEY file; OBEY files can be nested to a depth of four.

If the default setting of SYSTEM or VOLUME is changed in an OBEY file, the setting is not automatically returned to the previous state when the OBEY terminates.

If any part of the specification is invalid, if the file does not exist, or if the file cannot be opened, an error occurs. INSPECT displays an error message and prompts for input if the input file is a terminal. If the input file was not a terminal, INSPECT terminates.

OUT Command

The OUT command directs the output listing to a specified file. The syntax of the OUT command is:

```
OUT { file-name          }
    { / OUT file-name /  }

where

  file-name

      is a file name.
```

The first form of the OUT command causes permanent redirection of the output.

The second form of the OUT command causes temporary redirection of the output. This form is specified as part of another command and must be positioned immediately after the other command name and before any other part of that command. For example:

```
    HELP/OUT filename/command-name
```

If the file name has the form of a disc file and the file does not exist, an EDIT file is created. If the named file is an existing disc file, the output is appended to the file.

If the file name is invalid or if the file cannot be opened, an error occurs. An error message is displayed and the listing is not executed.

SYSTEM Command

The SYSTEM command sets the default system for expansion of any file names.

```
SYSTEM [ \system ]

where

  system

     is a system name of the form \system
```

VOLUME Command

The VOLUME command sets the default volume and subvolume for expansion of any file names.

```
VOLUME { $volume                }
       { [ $volume. ] subvol }

where

  volume

     is a volume name of the form $volume

  subvol

     specifies a subvolume on volume.
```