



DCLDBG

Benutzerhandbuch / User's Manual

Ferry Bolhar-Nordenkampf
Juni / June 2005

Einführung / Introduction

Diese Version des DCL-Debuggers basiert auf früheren Arbeiten von Jan Holzer und enthält einige zusätzliche Funktionen. / This version of the DCL debugger is based on prior work of Jan Holzer and contains some additional features.

```
DECUS (TELNET, T-ONLINE) - EXTRA!® Enterprise 2000
Datei Bearbeiten Ansicht Extras Session Optionen Hilfe
SOURCE: USER01:[UNGER]DK_STAT.COM ----- lines: 479 -----
23: $!           unique
24: $!
25: $!           p3           [optional, no wildcards allowed, default "SYS$PRINT"]
26: $!           print queue name, ignored if p2 is not equal to "PRINT"
27: $!
28: $! -----
29: $!
-> 30: $ save_verify = 'F$VERIFY(0)' ! Turn off verification
31: $!
32: $ ON CONTROL_Y THEN GOTO control_y
33: $!
34: $ GOSUB check_parameters
35: $!
36: $! -----
37: $!
38: $ yellow_alert = 75           ! Percent full for yellow alert
39: $ red_alert = 90             ! Percent full for red alert
-----
PROMPT -errors-program-input-output-----
DCLDBG> h
Enter/CR (step), skip it, up [<n>], down [<n>], $<DCL command>, <n> (n steps)
exit, quit, go, examine <symbol/logical>, break %<line>/<label>, clear break
watch [<symbol/logical>], find <string>, jump, reload, refresh, help or ?
DCLDBG>
1 (024,009)
```

Ferry Bolhar-Nordenkampf, bol@adv.magwien.gv.at
Michael Unger, unger@despammed.com

Editor's note: User's Manual layout created and converted to PDF by Michael Unger. The German part "Ein Debugger für DCL" is Ferry's original documentation taken from the plain text file and adapted to this layout.

This manual is to be submitted to the OpenVMS Freeware maintainer together with the DCL procedure itself (for a Freeware CD-ROM and/or publication on the web) and may be freely distributed. Neither the author nor the editor expect any remuneration for the use of this procedure and documentation from any party.

Ein Debugger für DCL

Aufruf

Aufgerufen wird der Debugger (selbst eine mehr als 650 Zeilen umfassende DCL-Prozedur) mit einem Parameter, der zu debuggenden Prozedur, und optional bis zu sieben weiteren Parametern, die der Prozedur als P1...P7 zur Verfügung gestellt werden. Beispiel:

```
$ @DCLDBG MY_PROC MY_PARAMETER
```

Hier wird die Prozedur MY_PROC.COM (im aktuellen Verzeichnis) debugged. Der Prozedur wird als Parameter P1 der String "MY_PARAMETER" mitgegeben.

Initialisierung und Startup

Nach dem Aufruf übersetzt der Debugger zunächst den logischen Namen "DCLDBG\$INIT". Existiert dieser Name und zeigt auf eine vorhandene DCL-Datei (Default: SYS\$LOGIN:.COM), so wird diese durchgeführt. Auf diese Art können Symbole oder logische Namen vordefiniert oder mit DEFINE/KEY Funktionstasten mit häufig verwendeten Befehlen belegt werden (diese Belegung ist aber nur für die Dauer der Debug-Session wirksam).

Danach "lädt" der Debugger zunächst die angegebene Prozedur, dh., er liest sie Zeile für Zeile ein und merkt sich u.a. Labels, die später angesprungen werden. Dieser Vorgang braucht je nach Länge und Komplexität der Prozedur Zeit; bei größeren Prozeduren empfehle ich das Kopieren der Prozedur und des Debuggers und das Laufenlassen auf einer Alpha. Auf VAXen muss man pro 60-80 Zeilen ca. mit einer Sekunde Verarbeitungszeit rechnen (Kommentarzeilen nicht eingerechnet), auf Alphas geht es natürlich schneller.

Alle Zeilen der Prozedur sowie diverse andere Daten werden in Symbolen abgelegt. Das heißt, der System Parameter CLISYMTBL sollte ausreichend groß gesetzt sein.

Man gelangt dann in eine interaktive semi-graphische Umgebung (dh., die verwendete Terminalemulation sollte VT-100 kompatible Escape-Sequenzen interpretieren können; ich empfehle generell die Verwendung des Debuggers nur von einem DECTerm-Fenster aus). Wer den VMS-Debugger kennt, wird sich hier sofort heimisch fühlen; das Ding ist ihm sehr ähnlich. Aber auch wer ihn nicht kennt, wird sich schnell zurechtfinden; die Bedienung ist sehr intuitiv.

Der Schirm ist unterteilt in einen List- und einen Kontrollbereich. Im Listbereich wird die zu debuggende Prozedur mit Zeilennummern angezeigt: die jeweils 7 vorhergehenden Zeilen, die **nächste** auszuführende Zeile (ist mit der Marke "→" gekennzeichnet) und die folgenden 9 Zeilen (so vorhanden). Bei Fortsetzungszeilen steht die Marke in der letzten Zeile. Im Kontrollbereich (5 Zeilen) befindet sich die Eingabeaufforderung "DCLDBG>". Hier wird die Ausgabe aller eingegebenen Befehle hingeschrieben und falls die Prozedur etwas auf SYS\$OUTPUT ausgibt, landet das ebenfalls hier.

Der Debugger kennt den Trace (Einzelschritt) Modus, in dem jeweils eine DCL-Zeile ausgeführt und danach wieder auf den nächsten Befehl gewartet wird, und den Run (Lauf) Modus, in dem mehrere Zeilen hintereinander ohne Unterbrechung abgearbeitet werden. Nach dem Starten befindet man sich im Trace-Modus; durch die Befehle "G" oder "J" wird in den Run-Modus gewechselt, aus dem man durch das Auflaufen auf einen Breakpoint oder durch die Eingabe von [CTRL][Y] wieder in den Trace-Modus gelangt.

Wird das Ende der Prozedur erreicht, beendet sich auch der Debugger automatisch.

Befehle

Die Befehle (in alphabetischer Reihenfolge, Groß-/Kleinschreibung spielt keine Rolle) sind:

[RETURN] (*RETURN-Taste*)

Führt den nächsten Befehl aus und wartet dann wieder auf eine Eingabe.

In durch CALL oder GOSUB aufgerufene Subroutinen wird hineingesprungen, dh., auch diese werden im Trace-Modus abgearbeitet. Soll die Subroutine im Run-Modus ablaufen (dh., innerhalb der Subroutine kein Tracing gemacht werden), kann dies mit dem "J"-Befehl geschehen.

<n>

Führt die nächsten <n> Befehle im Run-Modus durch und wartet danach wieder auf eine Eingabe. <n> ist eine Zahl im Bereich 1...65535.

\$ <cmd>

Führt den DCL-Befehl <cmd> aus. Dieser Befehl wird im Context der Prozedur ausgeführt, dh., es ist der Zugriff auf alle lokalen Symbole der Prozedur möglich und auch das Verzweigen (GOTO) oder das Aufrufen von Subroutinen (CALL, GOSUB) oder deren vorzeitiges Beenden mit RETURN oder EXIT. Kurz, alle Befehle, die normalerweise nur innerhalb von Prozeduren vorkommen dürfen, können hier interaktiv eingegeben werden! Natürlich können auch alle anderen DCL-Befehle hier verwendet werden; es können Images gestartet, Unterprozeduren aufgerufen (diese laufen aber im Run-Modus ab; es gibt kein Tracing), es kann das Defaultverzeichnis gewechselt werden u.v.a.m. All das passiert genauso, wie wenn der betreffende Befehl an dieser Stelle in der Prozedur vorgekommen wäre.

Neben den mit CALL, GOSUB und GOTO verwendeten Labelnamen können mit der Notation "%<line>" auch Zeilen direkt angesprungen werden (<line> ist die Nummer der Zeile, die angesprungen werden soll, zB. "\$ GOTO %65": springe zur Zeile 65 und warte dort auf weitere Eingaben).

? (Help)

Gibt im Kontrollbereich in Kurzform eine Übersicht über alle Befehle aus (dasselbe wie H).

B <pos> (Breakpoint)

Setzt einen Breakpoint an der angegebenen Position. <pos> ist entweder der Name eines Labels oder — mit vorangestelltem % — die Nummer einer Zeile. Beispiele:

```
B UPDATE
```

Setzt den Breakpoint auf die Zeile, die nach dem Label "UPDATE:" folgt.

```
B %23
```

Setzt den Breakpoint auf die Zeile 23.

Beachte, dass in dieser Version des Debuggers nur ein Breakpoint gesetzt werden kann. Wird ein neuer gesetzt, so wird der alte überschrieben.

Auf Kommentarzeilen und auf Zeilen, die mit DECK, ELSE, ENDIF, ENDSUBROUTINE, EOD, SUBROUTINE und THEN beginnen, kann kein Breakpoint gesetzt werden.

Bei Befehlen, die sich über zwei oder mehr Zeilen erstrecken, ist der Breakpoint auf die erste Zeile (in der der Befehl beginnt) zu setzen.

C (Cancel breakpoint)

Löscht den gesetzten Breakpoint wieder.

D [<n>] (Down)

Springt in der Prozedur eine Zeile (oder <n> Zeilen) nach unten. Die dazwischenliegenden Zeilen werden nicht ausgeführt. Auf diese Art kann man in der Prozedur "blättern" bzw. Codesegmente in der Ausführung überspringen.

E <name> (Examine)

Gibt den Wert von <name> aus. Ist <name> in einer der durch den logischen Namen LNM\$DCL_LOGICAL definierten logischen Nametables definiert, so wird der logische Name ausgegeben. Andernfalls wird <name> als Name eines DCL-Symbols interpretiert und der Wert dieses Symbols — so vorhanden — ausgegeben.

F <string> (*Find*)

Sucht in der Prozedur von der augenblicklichen Position abwärts nach dem angegebenen String. Wird der String gefunden, wird zur betreffenden Zeile gesprungen und dort auf eine weitere Eingabe gewartet. Die übersprungenen Zeilen werden nicht abgearbeitet.

Der Suchstring darf keine Leerzeichen enthalten. Die Suche kann in Abhängigkeit von der Größe der Prozedur einige Zeit in Anspruch nehmen.

G (*Go*)

Startet die Verarbeitung, es wird in den Run-Modes gewechselt, bis ein Breakpoint gefunden, [CTRL][Y] gedrückt oder die Prozedur beendet wird. Im Run-Modus wird die Ausgabe im List-Bereich nicht aktualisiert, sondern erst, wenn wieder in den Trace-Modus gewechselt wird. Dann wird wieder die als nächste auszuführende Zeile (mit “→” markiert), sowie die sieben vorigen und neun folgenden Zeilen ausgewiesen.

Ausgaben, die die Prozedur während des Laufes macht, werden in den Kontrollbereich geschrieben.

H (*Help*)

Gibt im Kontrollbereich in Kurzform eine Übersicht über alle Befehle aus (dasselbe wie “?”).

J (*Jump*)

Enthält die aktuelle Zeile eine CALL oder GOSUB Anweisung, so wird die angegebene Subroutine im Run-Modus ausgeführt und anschließend wieder in den Trace-Modus zurückgewechselt. Das kann hilfreich sein, wenn bereits bekannt ist, dass die Subroutine einwandfrei arbeitet oder die Subroutine innerhalb einer Schleife immer wieder aufgerufen wird und ein mehrfaches Tracen daher nicht sinnvoll ist. “J” entspricht somit dem Setzen eines Breakpoints auf die auf CALL oder GOSUB folgende Zeile, gefolgt von einem “G” (und ist intern auch tatsächlich so implementiert).

L (*reLoad*)

Startet die Prozedur wieder mit der ersten auszuführenden Zeile. Beachte, dass allfällige Symbole, die durch die Prozedur gesetzt wurden, durch diesen Befehl nicht gelöscht werden. Wenn also in der Prozedur in Abhängigkeit des Wertes eines oder mehrerer Symbole unterschiedliche Codeteile verarbeitet werden, kann das u.U. zu einem falschen Ablauf führen. In diesem Fall ist es sicherer, den Debugger zu beenden und neu zu starten.

Beim Reload wird auch — falls definiert — die Initialisierungsprozedur erneut durchgeführt.

P <pos> (*Print*)

Gibt die angegebene Zeile aus. Für <pos> gilt das unter dem “B”-Befehl Gesagte.

Q (*Quit*)

Beendet den Debugger (dasselbe wie "X").

R (*Refresh*)

Baut die Bildschirmausgabe neu auf. Das ist nützlich, wenn die Ausgabe des Debuggers durch Programme (z.B. TYPE/PAGE) überschrieben wurde.

S (*Skip*)

Überspringt die nächste Zeile (die Zeile wird nicht ausgeführt) und wartet dann wieder auf eine Eingabe.

U [*<n>*] (*Up*)

Springt in der Prozedur eine Zeile (oder <n> Zeilen) nach oben. Die dazwischenliegenden Zeilen werden nicht ausgeführt. Auf diese Art kann man in der Prozedur "blättern" bzw. Codesegmente in der Ausführung überspringen oder neuerlich ausführen.

V (*View*)

Gibt die Position des gesetzten Breakpoints aus. Ausgegeben wird immer die Zeilennummer der betreffenden Zeile, auch wenn zum Setzen des Breakpoints ein Labelname verwendet wurde.

W [*<symbol>*] (*Watch*)

Gibt im Trace-Modus nach der Ausführung jeder Zeile den Wert des angegebenen Symbols aus. Dazu wird der Kontrollbereich geteilt; das Symbol wird in einem eigenen Watch-Bereich in der untersten Zeile ausgegeben; der Kontrollbereich wird auf drei Zeilen reduziert. Die Eingabe von "W" ohne Symbolnamen schaltet das Watching wieder ab.

Im Run-Modus erfolgt die Aktualisierung des Symbolwertes erst nachdem wieder in den Trace-Modus gewechselt wurde.

Es kann immer nur ein Symbol überwacht werden. Ein neuerliches Watch überschreibt ein Vorheriges.

Achtung: um ein Symbol zu überwachen, muss es bereits definiert sein. Das kann in der Prozedur oder schon vorher in der Initialisierungsdatei geschehen.

X (*eXit*)

Beendet den Debugger (dasselbe wie "Q").

Syntaktische Vorgaben

Der Debugger besitzt einen recht schnellen, aber sehr einfachen Parser, der einige Befehle in einer ganz bestimmten Syntax bzw. Reihenfolge vorfinden muss, um sie erkennen zu können. Bevor eine Prozedur daher debugged werden kann, muss sie ggf. an die folgenden Regeln angepasst werden:

- 1) Ein Label muss das einzige Wort in einer Zeile (außer Kommentaren) sein, dh. statt

```
$ LABEL: SHOW TIME
```

nur

```
$ LABEL:  
$ SHOW TIME
```

(Ausnahme: Labels, denen der SUBROUTINE Befehl folgt.)

- 2) In einem strukturierten IF-Befehl darf nach THEN und ELSE kein Befehl folgen, sondern diese beiden Schlüsselworte müssen alleine in einer Zeile stehen. Daher statt

```
$ IF A .EQ. 3  
$ THEN B = 4  
$ ELSE B = 5  
$ ENDIF
```

nur

```
$ IF A .EQ. 3  
$ THEN  
$ B = 4  
$ ELSE  
$ B = 5  
$ ENDIF
```

- 3) Die Befehlswoorte CALL, GOTO und GOSUB dürfen nicht am Ende eines Symbolnamens, logischen Namens oder einer Zeichenkette vorkommen. Daher statt

```
$ TEST_CALL = 1
```

zB.

```
$ TEST_CALL_X = 1
```

- 4) Das Befehlswort RETURN darf generell nur am Anfang der Zeile als Befehl verwendet werden, es darf sonst nirgendwo (auch nicht als Teil eines Namens oder Strings) in einer

Zeile vorkommen. Sollte das notwendig werden (zB. als Teil eine Ausgabe), kann man es “zusammensetzen”. Daher statt:

```
$ WRITE SYS$OUTPUT "RETURN CODE: ",code
```

zB.

```
$ WRITE SYS$OUTPUT "RET", "URN CODE: ",code
```

- 5) Häufig werden lange Befehle abgekürzt, zB. RET (RETURN), SUB (SUBROUTINE), ENDSUB (ENDSUBROUTINE). Das ist mit dem Debugger nicht mehr erlaubt, weil der Parser diese Befehle dann nicht mehr erkennt. Obige Befehle sowie CALL, ELSE, ENDIF, GOSUB, GOTO und THEN müssen immer ganz ausgeschrieben werden.

Häufig legt man Befehle oder Befehlssteile in Symbolen ab, um später weniger tippen zu müssen, zB.:

```
$ SSY := SHOW SYMBOL  
$ SSY
```

Obwohl dies bei den o.a. Befehlen eher unüblich ist, sei dennoch darauf hingewiesen, dass das Ablegen dieser Befehle in Symbole und deren spätere Verwendung unter Angabe des Symbolnamens, wie hier gezeigt, nicht erlaubt ist.

- 6) Der Qualifier /END_OF_FILE des READ Befehls muss mit /END abgekürzt werden, zwischen ihm und dem folgenden “=” Zeichen darf kein Leerzeichen stehen. Daher statt:

```
$ READ/END_OF_FILE = THE_END
```

nur

```
$ READ/END=THE_END
```

- 7) Der Qualifier /ERROR der CLOSE, OPEN, READ und WRITE Befehle muss genauso geschrieben werden, er darf nicht abgekürzt werden. Auch hier muss das “=” unmittelbar danach folgen. Daher statt:

```
$READ/ERR = READ_ERROR
```

nur

```
$READ/ERROR=READ_ERROR
```

- 8) Bei einer durch CALL aufgerufenen Subroutine muss vor dem ENDSUBROUTINE Befehl ein EXIT Befehl (optional mit einem Exitstatus) stehen, daher statt:

```
$ MY_SUB: SUBROUTINE  
$ ...  
$ ...  
$ ENDSUBROUTINE
```

nur

```
$ MY_SUB: SUBROUTINE  
$ ...  
$ ...  
$ EXIT [<status>]  
$ ENDSUBROUTINE
```

- 9) Der Debugger verwendet intern Symbole, die alle mit “SS\$_” beginnen. Daher sollte dieser Präfix für Symbolnamen in der Prozedur nicht verwendet werden.

Funktionelle Einschränkungen

Der Debugger und die zu debuggende Prozedur teilen sich ein Environment. Das führt dazu, das bestimmte Befehle, die das Environment beeinflussen, nicht mehr oder anders funktionieren.

- 1) Zum Abfragen von Return Codes nur \$STATUS verwenden. \$SEVERITY wird nicht unterstützt. Man kann aber mit dem Ausdruck (\$STATUS .and. 7) den Wert von \$SEVERITY emulieren.
- 2) Da zwischen den einzelnen Zeilen der Prozedur mehrere Dutzend Zeilen Debugger-Code abgearbeitet werden, ist auch der ON <severity> THEN ... Befehl nicht unterstützt. Falls notwendig, muss dieser Code durch explizites Ausführen (GOTO <Label>) getestet werden.
- 3) Der Debugger fängt [CTRL][Y] ab, um vom Run-Modus in den Trace-Modus zu wechseln. Daher wird ein solcher Trap nicht an die Prozedur weitergegeben; falls gewünscht, muss der auszuführende Code explizit getestet werden.
- 4) Da der Debugger die Eingabe für seinen Prompt benötigt, können Tools, die eine interaktive Eingabe erwarten (dh., die von SYSS\$COMMAND einlesen) nicht ausgeführt werden.

INQUIRE und READ/PROMPT sind erlaubt.

- 5) Es wird empfohlen, kein Verify zu machen (SET NOVERIFY am Anfang der Prozedur), da sonst neben den Zeilen der Prozedur auch der gesamte Debugger-Code ausgegeben wird, was zu einem ziemlich unübersichtlichen Ablauf führt.

A Debugger for DCL

Invocation

The debugger (which is a DCL procedure itself containing more than 650 lines) is invoked with one parameter, the name of the procedure to be debugged, and optionally up to seven additional parameters which are passed to the procedure as P1...P7. Example:

```
$ @DCLDBG MY_PROC MY_PARAMETER
```

Here the procedure MY_PROC.COM (in the current directory) will be debugged. As a parameter P1 the string "MY_PARAMETER" is passed to the procedure.

Initialization and Startup

Having been invoked the debugger first translates the logical name "DCLDBG\$INIT". If this name is existing and pointing to a DCL file (default: SYSS\$LOGIN:.COM) that procedure will be executed. This way symbols and logical names can be predefined or by DEFINE/KEY function keys can be assigned frequently used commands (this assignment is effective for the duration of the debug session only).

After that the debugger "loads" the specified procedure, i.e., it reads line by line and among other things remembers labels at which is jumped later on. This will take some time depending on the procedure's length and complexity; for rather large procedures I recommend copying the procedure and the debugger to an Alpha and have it run there. On VAXes one second of runtime is to be reckoned with for about 60-80 lines of code (commentary lines not included), it is faster on Alphas of course.

All lines of the procedure as well as several other data are stored in symbols. Therefore the system parameter CLISYMTBL should be set sufficiently large.

Then an interactive semi-graphical environment is reached (i.e., the terminal emulation being used should be able to interpret VT-100 compatible escape sequences; I generally recommend using the debugger from a DECterm window only). Who already knows the VMS debugger will be immediately familiar with this tool because it is very similar. But also who doesn't know it will cope with it really soon; the operation is very intuitive.

The screen is divided into a "list" and a "control" area. Within the "list" area the procedure to be debugged is displayed with line numbers: the 7 preceding lines, the **next** line to be executed (marked with "→"), and the following 9 lines (if present). In the case of continuation lines the mark is set to the last line. The prompt "DCLDBG>" is located within the "control" area (5 lines). The output of all commands entered is written to this area as well as output (to SYSS\$OUTPUT) from the procedure itself.

The debugger recognizes the “trace” mode in which one DCL line gets executed at a time and the next input is waited for and the “run” mode in which several lines get executed at a stretch without interruption. Having just been started the “trace” mode is selected; entering the commands “G” or “J” switches to the “run” mode which is left again if a breakpoint is reached or [CTRL][Y] is pressed.

If the end of the procedure is reached the debugger will be terminated automatically.

Commands

The commands (in alphabetical order, upper or lower case doesn’t matter) are:

[RETURN] (*RETURN key*)

Executes the next command and then waits for input again.

Subroutines jumped into by CALL or GOSUB are processed in “trace” mode too. If the subroutine shall be processed in “run” mode (i.e., no tracing being done within the subroutine) the “J” command can be used.

<n>

Executes the following <n> commands in “run” mode and after that waits for input again. <n> in a number in the 1...65535 range.

\$ <cmd>

Executes the DCL command <cmd>. This command is executed in the procedure’s context, i.e., accessing all local symbols of the procedure is possible as well as branching (GOTO) and calling subroutines (CALL, GOSUB) or prematurely exiting from these by RETURN or EXIT. Briefly, all commands which under normal circumstances may only occur within procedures can be entered interactively here! Of course all other DCL commands can be used too; images can be run, subprocedures can be called (but these are executed in “run” mode; there is no tracing), the default directory can be changed and much more. All this happens the very same way as if the specified command had occurred at this position in the procedure.

Besides the label names used in conjunction with CALL, GOSUB and GOTO lines can be jumped at directly using the “%<line>” notation (<line> is the number of the line which is to be jumped at, e.g., “\$ GOTO %65”: jump at line number 65 and wait for further input there).

? (*Help*)

Short overview of all commands written to the “control” area (same as “H”).

B <pos> (*Breakpoint*)

Places a breakpoint at the position given. <pos> is either the name of a label or — with a preceding “%” sign — the number of a line. Examples:

```
B UPDATE
```

Places a breakpoint at the line following the label “UPDATE:”.

```
B %23
```

Places a breakpoint at line 23.

Note that in this version of the debugger only a single breakpoint can be set. If a new one is set the old one gets overwritten.

A breakpoint cannot be set to commentary lines or to lines beginning with DECK, ELSE, ENDIF, ENDSUBROUTINE, EOD, SUBROUTINE or THEN.

The breakpoint has to be set to the first line (where the command is beginning) for commands spanning two or more lines.

C (*Cancel breakpoint*)

Removes a breakpoint previously set.

D [<n>] (*Down*)

Jumps down one line (or <n> lines) in the procedure. The command lines in between are not executed. That way it can be “scrolled” through the procedure or code segments can be skipped during execution.

E <name> (*Examine*)

Displays the value of <name>. If <name> is defined in one of the logical name tables specified by the logical name LNM\$DCL_LOGICAL, that logical name is displayed. Otherwise <name> is interpreted as a name of a DCL symbol and the value of this symbol — if present — is displayed.

F <string> (*Find*)

Searches for the specified string starting downwards from the current position. If the string is found the corresponding line is jumped at and further input is waited for there. The skipped lines don't get executed.

The search string must not contain space characters. Searching may be time consuming depending on the procedure's size.

G (*Go*)

Processing is started, “run” mode is switched to until a breakpoint is found, [CTRL][Y] is pressed or the procedure is terminated. The “list” area isn’t refreshed in “run” mode until it is switched back to “trace” mode. Then again the next line to be executed (marked with “→”) is displayed as well as the seven previous and the nine following lines.

The procedure’s output during execution is written to the “control” area.

H (*Help*)

Short overview of all commands written to the “control” area (same as “?”).

J (*Jump*)

If the current line contains a CALL or GOSUB command the specified subroutine will be executed in “run” mode and it will be switched back to “trace” mode thereafter. That may be helpful if the subroutine is already known to work properly or the subroutine is called again and again within a loop and repeated tracing would be pointless therefore. So “J” is equivalent to setting a breakpoint at the line following the CALL or GOSUB followed by a “G” (and it really is internally implemented this way).

L (*reLoad*)

Restarts the procedure from the first line to be executed. Note that any symbols set by the procedure will not be deleted by this command. If different code segments within the procedure are processed depending on the value of one or more symbols this may result in an incorrect sequence. It is more reliable in this case to terminate and then restart the debugger.

The initialization procedure — if defined — will be executed again on “reload”.

P <pos> (*Print*)

Displays the line specified. For <pos> applies what has been said for the “B” command.

Q (*Quit*)

Terminates the debugger (same as “X”).

R (*Refresh*)

Rebuilds the screen display. That’s handy if the debugger’s output has been overwritten by programs (TYPE/PAGE, e.g.).

S (*Skip*)

Skips the next line (the command doesn’t get executed) and waits for input again.

U [<n>] (*Up*)

Jumps up one line (or <n> lines) in the procedure. The command lines in between are not executed. That way it can be “scrolled” through the procedure or code segments can be skipped or executed another time.

V (*View*)

Displays the position of a breakpoint having been set. Always the respective line number is displayed even if a label name has been used setting the breakpoint.

W [<symbol>] (*Watch*)

In “trace” mode the value of the symbol specified will be displayed having executed a line. Therefor the “control” area is splitted; the symbol gets displayed in a specific “watch” area in the bottom line; the “control” area is reduced to three lines. Entering “W” without a symbol name switches watching off again.

In “run” mode the symbol’s value isn’t updated before having switched back to “trace” mode.

Only one symbol can be watched at the same time. Another “watch” overwrites a previous one.

Note: To watch a symbol it has to be defined already. This can be done within the procedure itself or previously in the initialization file.

X (*eXit*)

Terminates the debugger (same as “Q”).

Syntactical Guidelines

The debugger has a rather fast but very simple parser which has to find some commands in a specific syntax or sequence to be able to recognize them. Thus before a procedure can be debugged it has to be adapted to the following rules if necessary:

- 1) A label has to be the only word in a line (except for comments), i.e., instead of

```
$ LABEL: SHOW TIME
```

just

```
$ LABEL:  
$ SHOW TIME
```

(Exception: labels followed by the SUBROUTINE command.)

- 2) In a structured IF command there must not be a command following THEN and ELSE, but these two keywords have to be in a line of their own. Therefore instead of

```
$ IF A .EQ. 3
$ THEN B = 4
$ ELSE B = 5
$ ENDIF
```

just

```
$ IF A .EQ. 3
$ THEN
$ B = 4
$ ELSE
$ B = 5
$ ENDIF
```

- 3) The command keywords CALL, GOTO and GOSUB must not occur at the end of a symbol name, logical name or character string. Therefore instead of

```
$ TEST_CALL = 1
```

for example

```
$ TEST_CALL_X = 1
```

- 4) The keyword RETURN may generally be used as a command only at the beginning of a line, it mustn't occur anywhere else (not even as part of a name or character string) in a line. Should that become necessary (as part of a display routine, e.g.) it may be "assembled". Therefore instead of

```
$ WRITE SYS$OUTPUT "RETURN CODE: ",code
```

for example

```
$ WRITE SYS$OUTPUT "RET", "URN CODE: ",code
```

- 5) Quite often long commands are abbreviated, e.g., RET (RETURN), SUB (SUBROUTINE), ENDSUB (ENDSUBROUTINE). That is no longer permissible with the debugger because the parser doesn't recognize these commands any longer. The commands mentioned above as well as CALL, ELSE, ENDIF, GOSUB, GOTO and THEN always have to be spelled out completely.

Frequently commands or parts of commands are deposited into symbols to save typing later on, e.g.,

```
$ SSY := SHOW SYMBOL  
$ SSY
```

Although this may be rather unusual with the commands mentioned above it has to be noted nevertheless that depositing these commands into symbols and using them later on by referring to the symbol name, as described here, is not permitted.

- 6) The qualifier `/END_OF_FILE` of the `READ` command has to be abbreviated using `/END`, there must not be any space character around the following “=” character. Therefore instead of

```
$ READ/END_OF_FILE = THE_END
```

just

```
$ READ/END=THE_END
```

- 7) The qualifier `/ERROR` of the `CLOSE`, `OPEN`, `READ` and `WRITE` commands has to be written exactly that way, it must not be abbreviated. Here the “=” has to be following immediately too. Therefore instead of

```
$READ/ERR = READ_ERROR
```

just

```
$READ/ERROR=READ_ERROR
```

- 8) In a subroutine invoked by `CALL` there has to be an `EXIT` command (optionally with an exit status) preceding the `ENDSUBROUTINE` command, therefore instead of

```
$ MY_SUB: SUBROUTINE  
$ ...  
$ ...  
$ ENDSUBROUTINE
```

just

```
$ MY_SUB: SUBROUTINE  
$ ...  
$ ...  
$ EXIT [<status>]  
$ ENDSUBROUTINE
```

- 9) The debugger internally uses symbols all of them beginning with “`SS$_`”. Therefore this prefix should not be used for symbol names within the procedure.

Functional Restrictions

The debugger itself and the procedure to be debugged share a common environment. That implies that specific commands which affect the environment won't work any longer or will work differently.

- 1) Only use `$_STATUS` to query return codes; `$_SEVERITY` isn't supported. But using the expression `($_STATUS .and. 7)` the value of `$_SEVERITY` can be emulated.
- 2) Umpteen lines of debugger code are executed between the lines of the procedure, so the command `ON <severity> THEN ...` isn't supported as well. If necessary this code has to be tested by explicit `(GOTO <label>)` execution.
- 3) The debugger intercepts `[CTRL][Y]` to switch from "run" mode to "trace" mode. Therefore such a trap isn't passed on to the procedure; if desired the code to be executed has to be tested explicitly.
- 4) Because the debugger requires the input for his own prompt tools expecting an interactive input (i.e., reading from `SY$_COMMAND`) can not be executed.

`INQUIRE` and `READ/PROMPT` are permitted.

- 5) It is recommended not to do procedure verification (`SET NOVERIFY` at the procedure's very beginning) because otherwise in addition to the lines of the procedure the entire debugger code will be displayed which results in a rather confusing processing.

Die Originalprozedur / The original Procedure (Jan Holzer)

Anmerkung des Herausgebers / Editor's Note

An verschiedenen Stellen der Prozedur steht eine Zeile ähnlich der folgenden, die als Beispiel genommen wurde: / At several points within the procedure there is a line similar to the following one which has been taken as an example:

```
$ ss$_d = "(0q"(B"                                ! horizontal line (graphical)
```

Darin ist ein nicht druckbares Zeichen enthalten. Wahrscheinlich sollte diese Zeile stattdessen lauten: / A non-printable character is contained therein. Probably this line should read instead:

```
$ ss$_d = ss$_ESC + "(0q" + ss$_ESC + "(B"          ! horizontal line (graphical)
```

oder /or

```
$ ss$_d = "'ss$_ESC'(0q'ss$_ESC'(B"                ! horizontal line (graphical)
```

Genau dasselbe gilt für die anderen Stellen. / The very same is true for the other occurrences.

DCL-Prozedur / DCL Procedure

```
$ set noverify
$ !
$ !=====
$ !
$ ! DCL_DEBUG.com
$ !
$ ! This procedure takes a .COM file and single-steps its execution.
$ !
$ ! Change Log:
$ !
$ ! ??-Feb-88  B. Engelhardt  initial coded SINGLE_STEP.GPX_COM.
$ ! 31-Aug-88  B. Engelhardt  created VT240 version by converting
$ !                               SINGLE_STEP.GPX_COM. Fixed label recognizer
$ !                               to recognize that "$[ ]<verb> [... ]<name>:".
$ !                               is not a label. Re-worked label storage
$ !                               for greater efficiency.
$ !
$ ! 27-Sep-88  B. Engelhardt  added recognition of GOSUB and RETURN.
$ ! 18-Oct-88  B. Engelhardt  added compound IF-THEN-ELSE.
$ ! 07-Nov-88  B. Engelhardt  added check for branch in IF structure
$ ! 23-Mar-89  B. Engelhardt  does terminal/nowrap to keep display clean
$ ! 21-Jul-90  J. Holzer      added e(xamine) for symbols and logicals,
$ !                               q(quit) and h(elp). changed (e)x(it)
$ !                               moved hardcoded values to symbols
$ !                               changed to VMS-DEBUG like screen
$ ! 27-Jul-90  J. Holzer      screen update only after GO, not on each step.
$ ! 28-Jul-90  J. Holzer      added check to avoid break-points on lines
$ !                               without command.
$ !
$ ! 31-Jul-90  J. Holzer      added .com file reload option.
$ ! 01-Aug-90  J. Holzer      fixed open/error=<label> trap bug.
$ !                               fixed line continuation bug.
$ !
$ !                               added recognition of CALL structure.
$ ! 07-Aug-90  J. Holzer      changed default ^Y behaviour to exit.
$ ! 14-Aug-90  J. Holzer      skip on-error and on-control per default.
$ ! 14-Sep-90  J. Holzer      restore p1 to p7 on reload.
```

```

$ ! 21-Sep-90 J. Holzer      added proper screen reset on coded EXIT.
$ !
$ ! 21-Sep-90 J. Holzer      added copyrighth on startup.
$ ! 26-Sep-90 J. Holzer      shortened symbol names.
$ ! 26-Sep-90 J. Holzer      changed back to hardcoded constants
$ !                          added WATCH feature
$ ! 01-Oct-90 J. Holzer      calculate WATCH value after "$<DCL command>"
$ !
$ ! =====
$ !
$ ! © 1990,1991 by Jan Holzer, Digital Equipment Munich.
$ !
$ ! =====
$ !
$ ! The terminal screen is divided into listing and control region. The
$ ! listing region displays 17 lines of the .COM file (the current line, 7
$ ! lines before and 9 after). The control region is 5 lines at the bottom.
$ !
$ ! All the lines of the .COM file are loaded/saved as symbol values (to
$ ! be able to "GOTO"). This takes a while for large files (approx 10 sec
$ ! for 200 lines).
$ !
$ ! For each command in the process flow of the file, the command is displayed
$ ! in the middle of the listing region and an "DCLDEB>" prompt is issued. The
$ ! possible responses to that prompt are:
$ !
$ ! - <Return> or <Enter> causes the command to be executed.
$ !
$ ! - "s[kip]" skips the command.
$ !
$ ! - "$<DCL command>" executes <DCL command>. This is a very powerful
$ ! feature of DCL_DEBUG. Using this, you can check (or assign)
$ ! symbol values, branch to another line of the COM file, change the
$ ! default directory, copy files, run executables, execute other COM
$ ! files (but not single step them), etc. And this happens in the
$ ! context of the COM file, as if these commands were part of it.
$ !
$ ! The <DCL command> may be a GOTO or GOSUB to a label in the .COM file,
$ ! or a RETURN. The GOTO/GOSUB parameter may be also be "%<line-no>".
$ !
$ ! - "u[p] [<n>]" moves up n lines in the COM file. If n is not specified,
$ ! the default is one.
$ !
$ ! - "d[own] [<n>]" moves down n lines in the COM file.
$ !
$ ! - "<n>" causes n commands to be executed.
$ !
$ ! - "g[o]" causes the file to be run without prompting. Encountering
$ ! a break point or CONTROL_Y will resume single-stepping. CONTROL_Y
$ ! will not break if the "On CONTROL_Y then (break)" response is
$ ! different (e.g., "exit"). Since a DCL_DEBUG com file command
$ ! may be executing when the CONTROL_Y occurs, it is possible that
$ ! the process will be disrupted.
$ ! During the "go"-run, no screen update occurs.
$ !
$ ! - "[e]x[it]" or "q[uit]" exits.
$ !
$ ! - "r[epaint]" refreshes the display. This is useful after "$type",
$ ! etc., which uses the entire screen and erases the listing.
$ !
$ ! - "b[reak] {<label> or %<line_no>}" sets a break point at a label or
$ ! line. There can be only one break point at a time (a new "break"
$ ! replaces the previous one).
$ !
$ ! - "c[lear break]" clears (resets) the break point. Encountering a
$ ! break point or CONTROL_Y does not reset the break point.
$ !
$ ! - "f[ind] <string>" will search down the COM file for <string>. This
$ ! command can be VERY slow.
$ !
$ ! - "?" or "h" lists the options.
$ !
$ ! - "e[xamine] <symbol/logical> displays the value of the entered
$ ! symbol or logical. If a name is found in one of the logical tables

```

```

$ !      of the system, no further check for a possible symbol is done.
$ !
$ !      - [re]l[oad] reloads the entire command file and starts processing
$ !      again on the first command line.
$ !
$ !      - w[atch] <symbol/logical> creates a third window on the bottom of the
$ !      screen and displays the value of the entered variable on each
$ !      step (not in 'go'-mode). To remove the watch-window, just type 'w'
$ !      without any variable.
$ !
$ ! ONLY THE FIRST CHARACTER OF THE "DCLDEB>" RESPONSE IS EXAMINED! This can
$ ! be a problem if the "$" prefix is inadvertently left off a DCL command.
$ ! Thus, if a "show symbol" command is entered, but without the required "$",
$ ! a skip will occur.
$ !
$ !=====
$ !
$ ! About the display: the "current" line will be marked with an "->". The
$ ! current line is never a comment or label (these will be stepped over,
$ ! whether they are encountered by sequential stepping or as the destination
$ ! of a branch command (GOTO, etc) or UP/DOWN). For multiple-line (continued)
$ ! commands, the marker will be positioned on the last line. However, when
$ ! repositioning to a continued command (by "branch" %<line> or UP/DOWN), the
$ ! first line of the command must be referenced. E.g., to "goto" the DIFF
$ ! command below, the DCL_DEBUG command would be "$goto %45" and after
$ ! repositioning, the marker would be on 46:
$ !
$ !      45: $ DIFF -
$ !      -> 46:      login.com
$ !
$ ! Likewise, to set break point on a multi-line command, the first line of
$ ! the command must be referenced.
$ !
$ !=====
$ !
$ !      - the structured IF statement is limited to the following syntax:
$ !
$ !              IF <condition>
$ !              THEN
$ !                  <command>
$ !                  .
$ !                  .
$ !              [ELSE
$ !                  <command>
$ !                  .
$ !                  .
$ !              ]
$ !              ENDIF
$ !
$ !      I.e., there cannot be a command on the THEN or ELSE line. Also,
$ !      <condition> may not include the string: " THEN " (blanks included).
$ !
$ !      - There is no checking of the IF structure syntax. There is a check
$ !      for branching from within an IF structure. A warning is given
$ !      and the assumption is made that the branch will be to outside the
$ !      IF structure.
$ !
$ !      - GOSUB's cannot be nested (only 1 level of return). RETURN does
$ !      not take a parameter.
$ !
$ !      - VMS V5.x's CALL structure is implemented the following way:
$ !
$ !              $ <Label_Name>: SUBROUTINE          ! must be one line !!
$ !              $   <command>
$ !              $-
$ !              $   .
$ !              $   .
$ !              $   <command>
$ !              $ EXIT
$ !              $ ENDSUBROUTINE
$ !
$ !              $-
$ !              $ CALL <Label_Name> [p1 p2 p3 p4 p5 p6 p7 p8]
$ !              $-
$ !

```

```

$ ! - GOSUB's within a subroutine are not allowed.
$ !
$ ! - labels must be the only DCL on their line (excluding comments),
$ ! with no spaces between the label and the ":". For example:
$ !
$ !           $ do_compile:      ! run PASCAL compiler
$ !
$ ! Also, since the line-number syntax for DCL_DEBUG commands is
$ ! %<line>, a label may not begin with a "%". Or, if it does, you
$ ! cannot branch to it by an SS $GOTO command.
$ !
$ ! - the strings "GOTO ", "GOSUB ", "RETURN" and "CALL" must not be
$ ! used except as verbs (e.g., they may not be used as the last part
$ ! of a name).
$ ! These strings are searched-for in commands to detect branches.
$ ! Note that "GOTO", "CALL" and "GOSUB" may be used in the beginning or
$ ! middle of names, since the restricted (search) string includes a
$ ! trailing blank/space.
$ !
$ ! - DCL_DEBUG symbol names are of the form SS$_<name>. Since the
$ ! DCL_DEBUG commands and the COM file commands are executing at the
$ ! same command level, COM file symbol names should not use this form.
$ !
$ ! - If a exit-code is supplied to a coded EXIT command, the exit-code
$ ! is ignored and a normal EXIT (1) is performed.
$ !
$ ! - you cannot "step into" another COM file. That is, single step'ing
$ ! only applies to the COM file specified when DCL_DEBUG is invoked,
$ ! not to any COM file executed by reference within that file or
$ ! by a "$@<COM-file>" command. Other COM files can be executed,
$ ! they just can't be single step'ed.
$ !
$ ! - since DCL_DEBUG DCL is executed around the object COM file,
$ ! the global system symbols $STATUS and $SEVERITY can change between
$ ! COM file commands. Thus, COM tests of $STATUS and $SEVERITY are
$ ! meaningless and the user must interject control.
$ !
$ ! - similarly, the command: "ON severity THEN $ GOTO label" does not work.
$ ! But, you will see all failure messages and can easily invoke the
$ ! desired action.
$ !
$ ! - also, any executable or DCL command which requires input (e.g.,
$ ! CREATE or Edit), won't work (as part of the COM file or as a $<DCL_
$ ! command> response). Do these kinds of things in a subprocess (enter
$ ! "$$SPAWN" at DCLDEB>).
$ !
$ ! - since DCL_DEBUG is a COM file, using DCL commands, if a user
$ ! has redefined commands with symbol definitions, there may be an
$ ! effect on DCL_DEBUG.
$ !
$ ! - auto-wrap is turned off at the beginning of DCL_DEBUG. This is
$ ! to prevent the corruption of the display by wrapped lines. If the
$ ! driver was set to "wrap" before DCL_DEBUG (by SET TERMINAL/WRAP),
$ ! it is turned back on at the end. If the terminal_device_ was on
$ ! "wrap" (by terminal Set-Up menu), it is left on no-wrap.
$ !
$ ! - there is a maximum of seven parameters for the COM file's use. (This
$ ! is because the COM file spec is a parameter to DCL_DEBUG.)
$ !
$ ! =====
$ ! Procedure syntax:
$ !
$ !   DCLDEB <com_file>[.COM] [<p1> [<p2> [...<p7>]]]
$ !
$ ! where:
$ !
$ !   DCLDEB ::= @[<disk>:] [<directory>]DCL_DEBUG.COM and
$ !   p1, p2, ... p7 are the com_file's parameters (seven max)
$ ! =====
$ !
$ ss$_CSI[0,8] = 155

```



```

$ !-----
$ ! have a potential label (but, could be "$ <command>...<logical_name>:")
$ !-----
$ !
$ ! if f$locate ("$", ss$cmd_ln) .ne. 0 then goto READ_LOOP ! continuation line
$ !
$ !-----
$ ! check for embedded blank other than 2nd char
$ !-----
$ !
$ ss$frst_blnk = f$locate (" ", ss$cmd_ln)
$ if (ss$frst_blnk .ne. f$length (ss$cmd_ln)) .and. -
  (ss$frst_blnk .ne. 1) then goto READ_LOOP
$ !
$ !-----
$ ! there's none or there is one and it's the 2nd char - check for another
$ !-----
$ !
$ if f$element (2, " ", ss$cmd_ln) .nes. " " then goto READ_LOOP
$ !
$ !-----
$ ! none - it's a label
$ !-----
$ !
$SUB_LABEL:
$ ss$cmd_ln = f$edit (ss$cmd_ln, "collapse")
$ if ss$sub_lbl
$ then ss$lbl = f$extract (1, f$locate (":", ss$cmd_ln) - 1, ss$cmd_ln)
$ else ss$lbl = f$extract (1, f$length (ss$cmd_ln) - 2, ss$cmd_ln )
$ endif
$ ss$lbl_is_'ss$lbl' = ss$ln_no
$ ss$ln_'ss$ln_no'_is_label = "T"
$ goto READ_LOOP
$ !
$RUN_IT:
$ close ss$com_file
$ ss$mx_lns = ss$ln_no
$ !
$ gosub GET_NOR_SCR
$ !
$ ss$last_ln = ss$ln_no
$ ss$ln_no = 0
$ ss$_$cmd = "F"
$ ss$ret_ln = 0
$ ss$prv_ln = -17
$ ss$brk_pt = "F"
$ ss$brk_line = 0
$ ss$if_lvl = 0
$ ss$skp_lvl = 0
$ ss$skp_to = "F"
$ ss$ctrl_y = "F"
$ ss$run = "F"
$ ss$cnt = 0
$ ss$srch = "F"
$ ss$in_cll = "F"
$ ss$srch_endsub = "F"
$ ss$wtc = "F"
$ !
$ if (.not. ss$rld) then ss$say ss$CSI, "2J", ss$CSI, "0;0H", ss$CSI, "?7l"
$ !
$ set terminal/nowrap ! set term software no-wrap
$ !
$ ss$rld = "F"
$ !
$NEXT_CMD:
$ gosub GET_NEXT_CMD
$ !
$ if (.not. ss$skp_to) then goto SEARCH_TEST
$ gosub TRAP_BRANCHES
$ goto NEXT_CMD
$ !
$SEARCH_TEST:
$ if (.not. ss$srch) then goto AROUND_IT

```

```

$ if f$locate (ss$_srch_str, ss$_cmd) .eq. f$length (ss$_cmd) then -
    goto NEXT_CMD
$ ss$_srch = "F"
$ !
$AROUND_IT:
$ if (.not. ss$_run) then gosub DISPLAY_CMD
$ if (ss$_brk_pt .and. (ss$_cmd_ln_no .eq. ss$_brk_line) ) .or. -
    ss$_ctrl_y then goto BREAK
$ if f$type (ss$_ln_'ss$_ln_no'_is_label) .nes. "" then goto NEXT_CMD
$ if (ss$_run) .or. (ss$_cnt .gt. 0) then goto DO_IT
$ !
$INQUIRE:
$ read/prompt = "'ss$_prompt'" sys$output ss$_rsp
$ ss$_typ = f$edit (f$extract (0, 1, ss$_rsp), "upcase")
$ if ss$_typ .eqs. "" then goto DO_IT
$ if ss$_typ .eqs. "S" then goto NEXT_CMD
$ if ss$_typ .eqs. "$" then goto CMD_IN
$ if f$type (ss$_rsp) .eqs. "INTEGER" then goto DO_N
$ if ss$_typ .eqs. "G" then goto GO
$ if ss$_typ .eqs. "E" then goto EXAMINE
$ if ss$_typ .eqs. "U" then goto STEP_UP
$ if ss$_typ .eqs. "D" then goto STEP_DOWN
$ if ss$_typ .eqs. "R" then goto REPAINT
$ if ss$_typ .eqs. "F" then goto FIND_STRING
$ if ss$_typ .eqs. "B" then goto SET_BREAK_PT
$ if ss$_typ .eqs. "C" then goto CLEAR_BREAK_PT
$ if ss$_typ .eqs. "L" then goto RELOAD
$ if ss$_typ .eqs. "W" then goto WATCH
$ if ((ss$_typ .eqs. "Q") .or. (ss$_typ .eqs. "X")) then goto EXIT
$ if ((ss$_typ .eqs. "H") .or. (ss$_typ .eqs. "?")) then goto HELP
$ ss$_say ss$_err_unrent
$ goto INQUIRE
$ !
$DO_IT:
$ gosub TRAP_BRANCHES
$ 'ss$_cmd
$ if ss$_cnt .gt. 0 then ss$_cnt = ss$_cnt - 1
$ if (.not. ss$_run) .and. ss$_wtc then gosub DO_WATCH
$ goto NEXT_CMD
$ !
$DO_N:
$ ss$_cnt = f$integer (ss$_rsp)
$ if ss$_cnt .gt. 0 then goto DO_IT
$ ss$_say ss$_err_unrent
$ goto INQUIRE
$ !
$CMD_IN:
$ ss$_save_cmd = ss$_cmd
$ ss$_cmd = f$extract (1, f$length (ss$_rsp) - 1, ss$_rsp)
$ ss$_cmd = f$edit (ss$_cmd, "lowercase, uncomment, trim")
$ gosub TRAP_BRANCHES
$ ss$_$_cmd = "T"
$ ss$_brnch = "F"
$ 'ss$_cmd'
$ if ss$_wtc then gosub DO_WATCH
$ ss$_$_cmd = "F"
$ if ss$_brnch then goto NEXT_CMD
$ ss$_cmd = ss$_save_cmd
$ goto INQUIRE
$ !
$GO:
$ ss$_run = "T"
$ ss$_say ss$_go_output
$ goto DO_IT
$ !
$STEP_UP:
$ ss$_offset = f$element (1, " ", ss$_rsp)
$ if ss$_offset .eq. 0 then ss$_offset = 1
$ ss$_ln_no = ss$_ln_no - f$integer (ss$_offset) - 1      ! anticipate NEXT_CMD's
$ if ss$_ln_no .lt. 1 then ss$_ln_no = 0
$ goto NEXT_CMD
$ !
$STEP_DOWN:

```

```

$ ss$_offset = f$element (1, " ", ss$_rsp)
$ if ss$_offset .eq. 0 then ss$_offset = 1
$ ss$_ln_no = ss$_ln_no + f$integer (ss$_offset) - 1      ! anticipate NEXT_CMD's
$ if ss$ ln_no .gt. ss$_last_ln then ss$ ln_no = ss$_last_ln - 1
$ goto NEXT_CMD
$ !
$REPAINT:
$ gosub DO_REPAINT
$ goto INQUIRE
$ !
$FIND_STRING:
$ ss$_srch_str = f$edit (f$element (1, " ", ss$_rsp), "lowercase, trim")
$ ss$_srch = "T"
$ ss$_srch_line = ss$_cmd_ln_no
$ goto NEXT_CMD
$ !
$SET_BREAK_PT:
$ ss$_loc_prm = f$element (1, " ", ss$_rsp)
$ gosub GET_LINE_NO
$ IF ss$ lbl_ln_no .eq. 0 then goto INQUIRE
$ ss$_brk_pt = "T"
$ ss$_brk_line = ss$_lbl_ln_no
$ goto INQUIRE
$ !
$CLEAR_BREAK_PT:
$ ss$_brk_pt = "F"
$ goto INQUIRE
$ !
$BREAK:
$ ss$_run = "F"
$ ss$_ctrl_y = "F"
$ ss$_cnt = 0
$ if f$type (ss$ ln_'ss$ ln_no'_is_label) .nes. "" then goto NEXT_CMD
$ goto REPAINT
$ !
$END:
$ if .not. ss$_srch then goto EXIT
$ ss$_say ss$_err_notfnd
$ ss$ ln_no = ss$_srch_line - 1
$ ss$_srch = "F"
$ goto NEXT_CMD
$ !
$EXIT:
$ ss$_say ss$_c_off, ss$_CSI,"1;24r", ss$_CSI,"24;1H", ss$_c_on
$ set terminal/wrap
$ set control = (y,t)
$ exit 1
$ !
$HELP:
$ ss$_say " 'ss$_b'Enter/CR'ss$_n' (step), 'ss$_b's'ss$_n'kip it, " + -
    "'ss$_b'u'ss$_n'p [<n>], 'ss$_b'd'ss$_n'own [<n>], " + -
    "'ss$_b'$'ss$_n'<DCL command>, 'ss$_b'<n>'ss$_n' (n steps)"
$ ss$_say " e'ss$_b'x'ss$_n'it, 'ss$_b'q'ss$_n'uit, " + -
    "'ss$_b'g'ss$_n'o, 'ss$_b'e'ss$_n'xamine <symbol/logical>, " + -
    "'ss$_b'b'ss$_n'reak %<line>/<label>, 'ss$_b'c'ss$_n'lear break"
$ ss$_say " 'ss$_b'w'ss$_n'atch [<symbol/logical>], " + -
    "'ss$_b'f'ss$_n'ind <string>, re'ss$_b'l'ss$_n'oad, " + -
    "'ss$_b'h'ss$_n'elp or 'ss$_b'?'ss$_n'"
$ goto INQUIRE
$ !
$EXAMINE:
$ ss$_var = f$extract (1, f$length (ss$_rsp) - 1, ss$_rsp)
$ ss$_var = f$edit (ss$_var, "lowercase, uncomment, trim")
$ if (ss$_var .eqs. "") then goto INQUIRE
$ if f$trnlm ("'ss$_var'", "LNM$DCL_LOGICAL") .eqs. ""
$   then show symbol 'ss$_var'
$   else show logical 'ss$_var'
$ endif
$ goto INQUIRE
$ !
$WATCH:
$ ss$_wtc_nam = f$extract (1, f$length (ss$_rsp) - 1, ss$_rsp)
$ ss$_wtc_nam = f$edit (ss$_wtc_nam, "lowercase, uncomment, trim")

```

```

$ if (ss$_wtc_nam .eqs. "")
$   then gosub GET_NOR_SCR
$     ss$_wtc = "F"
$   else gosub GET_WTC_SCR
$     ss$_wtc = "T"
$ endif
$ gosub DO_REPAINT
$ goto INQUIRE
$ !
$DO_WATCH:
$ ss$_say ss$_wtc_area
$ if f$strnlm ("''ss$_wtc_nam'", "LNM$DCL_LOGICAL") .eqs. ""
$   then show symbol 'ss$_wtc_nam'
$   else show logical 'ss$_wtc_nam'
$ endif
$ ss$_say ss$_ctrl_area
$ return
$ !
$RELOAD:
$ ss$_rld = "T"
$ goto DO_RELOAD
$ !
$ !-----
$ ! subroutines
$ !-----
$ !
$GET_NOR_SCR:
$ ss$_ctrl_area = '''ss$_CSI'9;1H''ss$_b'->''ss$_n' " + -
$                 '''ss$_CSI'20;24r''ss$_CSI'24-1;1H"
$ ss$_go_output = '''ss$_CSI'20;24r''ss$_CSI'24;1H''ss$_CSI'1A"
$ ss$_prompt = '''ss$_CSI'24;1HDCLDEB> ''ss$_CSI'0K"
$ ss$_scr = '''ss$_CSI'1;1H''ss$_b''ss$_d' SOURCE: ''ss$_fil_nam' " + -
$           "(0'f$extract (1, 56 - f$len (ss$_fil_nam), ss$_frame)'(B" + -
$           " lines: " + f$fao ("!3SL", ss$_mx_lns) + " ''ss$_d''ss$_CSI'19;1H" + -
$           "'ss$_d' PROMPT ''ss$_d'errors''ss$_d'program''ss$_d'input''ss$_d'" + -
$           "output'(0'f$extract (1, 43, ss$_frame)'(B''ss$_n'"
$ return
$ !
$GET_WTC_SCR:
$ ss$_ctrl_area = '''ss$_CSI'9;1H''ss$_b'->''ss$_n' " + -
$                 '''ss$_CSI'20;22r''ss$_CSI'22-1;1H"
$ ss$_go_output = '''ss$_CSI'20;22r''ss$_CSI'22;1H''ss$_CSI'1A"
$ ss$_prompt = '''ss$_CSI'22;1HDCLDEB> ''ss$_CSI'0K"
$ ss$_scr = '''ss$_CSI'1;1H''ss$_b''ss$_d' SOURCE: ''ss$_fil_nam' " + -
$           "(0'f$extract (1, 56 - f$len (ss$_fil_nam), ss$_frame)'(B" + -
$           " lines: " + f$fao ("!3SL", ss$_mx_lns) + " ''ss$_d'" + -
$           "'ss$_CSI'19;1H''ss$_d' PROMPT ''ss$_d'errors''ss$_d'program''ss$_d'" + -
$           "input''ss$_d'output'(0'f$extract (1, 43, ss$_frame)'(B" + -
$           "'ss$_CSI'23;1H''ss$_d' WATCH ''ss$_d' ''ss$_wtc_nam' " + -
$           "(0'f$extract (1, 69 - f$len (ss$_wtc_nam), ss$_frame)'(B''ss$_n'"
$ return
$ !
$GET_NEXT_CMD:
$ ss$_cmd = ""
$ ss$_cmd_ln_no = ss$_ln_no + 1
$ !
$CMD_LOOP:
$ ss$_goto_cmd_loop = "F"
$ ss$_ln_no = ss$_ln_no + 1
$ if ss$_ln_no .gt. ss$_last_ln then goto END
$ ss$_cmd_ln = ss$_ln:ss$_ln_no
$ ss$_tmp_line = f$edit (ss$_cmd_ln, "lowercase, uncomment, compress, trim")
$ if ss$_srch_endsub
$   then ss$_endsub = -
$         f$lloc ("endsubroutine", ss$_tmp_line) .ne. f$len (ss$_tmp_line)
$         if ss$_endsub then ss$_srch_endsub = "F"
$         ss$_goto_cmd_loop = "T"
$   else ss$_sub = -
$         (f$lloc (" subroutine", ss$_tmp_line) .ne. f$length (ss$_tmp_line)) -
$         .and. (.not. ss$_in_cll)
$         if ss$_sub
$           then ss$_srch_endsub = "T"
$           ss$_goto_cmd_loop = "T"

```

```

$      endif
$ endif
$ if ss$_goto_cmd_loop then goto CMD_LOOP
$ if f$locate ("$", ss$_cmd_ln) .eq. 0 then -
    ss$_cmd_ln = f$extract (1, f$length (ss$_cmd_ln) - 1, ss$_cmd_ln)
$ ss$_cmd_ln = f$edit (ss$_cmd_ln, "lowercase, uncomment, compress, trim")
$ ss$_cmd = ss$_cmd + ss$_cmd_ln
$ if f$extract (f$length (ss$_cmd_ln) - 1, 1, ss$_cmd_ln) .nes. "-" then -
    goto GOT_CMD
$ ss$_cmd = f$extract (0, f$length (ss$_cmd) - 1, ss$_cmd)
$ goto CMD_LOOP
$ !
$GOT_CMD:
$ if ss$_cmd .eqs. "" then goto GET_NEXT_CMD
$ return
$ !
$TRAP_BRANCHES:                                ! "GOSUB" access
$ ss$_len = f$length (ss$_cmd)
$ if ((f$locate ("if ", ss$_cmd) .eq. 0) .or. -
    (f$locate ("if(", ss$_cmd) .eq. 0)) .and. -
    (f$locate (" then ", ss$_cmd) .eq. f$length (ss$_cmd))) then goto STRUCT_IF
$ if ss$_cmd .eqs. "then" then goto STRUCT_THEN
$ if ss$_cmd .eqs. "else" then goto STRUCT_ELSE
$ if ss$_cmd .eqs. "endif" then goto STRUCT_ENDIF
$ if ss$_skp_to then return
$ ss$_goto = f$locate ("goto ", ss$_cmd)
$ if ss$_goto .lt. ss$_len then goto REPLACE_GOTO
$ ss$_gsb = f$locate ("gosub ", ss$_cmd)
$ if ss$_gsb .lt. ss$_len then goto REPLACE_GOSUB
$ ss$_return = f$locate ("return", ss$_cmd)
$ if ss$_return .lt. ss$_len then goto REPLACE_RETURN
$ ss$_c11 = f$locate ("call ", ss$_cmd)
$ if ss$_c11 .lt. ss$_len then goto REPLACE_CALL
$ ss$_exit = f$locate ("exit", ss$_cmd)
$ if (ss$_exit .lt. ss$_len)
$   then if ss$_in_c11
$     then goto REPLACE_CALL_EXIT
$     else goto REPLACE_EXIT
$   endif
$ endif
$ endif
$ ss$_eof = f$locate ("/end=", ss$_cmd)
$ if ss$_eof .lt. ss$_len then goto REPLACE_EOF
$ !
$LOOKFOR_ERROR:
$ ss$_len = f$length (ss$_cmd)
$ ss$_error = f$locate ("/error=", ss$_cmd)
$ if ss$_error .lt. ss$_len then goto REPLACE_ERROR
$ return
$ !
$STRUCT_IF:                                    ! structured IF
$ if ss$_skp_to                                ! nested within skip-range
$   then ss$_skp_lvl = ss$_skp_lvl + 1
$   else ss$_if_lvl = ss$_if_lvl + 1
$   ss$_'ss$_if_lvl'_test = "F"
$   ss$_cmd = ss$_cmd + " then ss$_'ss$_if_lvl'_test = "T" " "
$ endif
$ return
$ !
$STRUCT_THEN:
$ if .NOT. ss$_skp_to                            ! not nested within skip-range
$   then if .NOT. ss$_'ss$_if_lvl'_test
$     then ss$_skp_to = "T"                      ! skip to ELSE or ENDIF
$     endif
$   ss$_cmd = ""                                ! skip THEN
$ endif
$ return
$ !
$STRUCT_ELSE:
$ if ss$_skp_lvl .eq. 0 ! not nested in skip-rng (could be skip_to, if level=0)
$   then if .NOT. ss$_'ss$_if_lvl'_test
$     then ss$_skp_to = "F"                      ! skipped-to here, do these
$     else ss$_skp_to = "T"                      ! executed to here, skip to ENDIF
$   endif
$

```

```

$          ss$_cmd = ""                                ! skip ELSE
$ endif
$ return
$ !
$STRUCT_ENDIF:
$ if ss$_skp_lvl .gt. 0                                ! within nested skip
$   then ss$_skp_lvl = ss$_skp_lvl - 1
$   else ss$_skp_to = "F"                              ! end of current level
$     ss$_if_lvl = ss$_if_lvl - 1
$     ss$_cmd = ""                                     ! skip ENDIF
$ endif
$ return
$ !
$REPLACE_GOTO:      ! change GOTO <label> to GOSUB GOTO_SUB with loc_prm set up
$ ss$_beg = f$extract (0, ss$_goto, ss$_cmd)
$ ss$_loc_prm = f$extract (ss$_goto + 5, (ss$_len - ss$_goto - 5), ss$_cmd)
$ ss$_cmd = ss$_beg + " GOSUB GOTO_SUB "
$ return
$ !
$REPLACE_GOSUB:    ! change GOSUB <label> to GOSUB GOSUB_SUB with loc_prm set up
$ ss$_beg = f$extract (0, ss$_gsb, ss$_cmd)
$ ss$_loc_prm = f$extract (ss$_gsb + 6, (ss$_len - ss$_gsb - 6), ss$_cmd)
$ ss$_cmd = ss$_beg + " GOSUB GOSUB_SUB "
$ return
$ !
$REPLACE_RETURN:   ! change RETURN to GOSUB RETURN_SUB
$ ss$_beg = f$extract (0, ss$_return, ss$_cmd)
$ ss$_cmd = ss$_beg + " GOSUB RETURN_SUB "
$ return
$ !
$REPLACE_CALL:     ! change CALL <label> to GOSUB CALL_SUB with loc_prm set up
$ ss$_cmd_sav = ss$_cmd
$ ss$_beg = f$extract (0, ss$_c11, ss$_cmd)
$ ss$_loc_prm = f$element (0, " ", -
  f$extract (ss$_c11 + 5, (ss$_len - ss$_c11 - 5), ss$_cmd))
$ ss$_cmd = ss$_beg + " GOSUB CALL_SUB "
$ !
$ ss$_c = 1
$PARAM_LOOP_1:     ! save p1-p7
$ ss$_tmp = p'ss$_c'
$ ss$_p'ss$_c'sav = "'ss$_tmp'"
$ ss$_tmp = f$element ('ss$_c', " ", -
  f$extract (ss$_c11 + 5, f$length (ss$_cmd_sav), ss$_cmd_sav))
$ ss$_prm_cnt = ss$_c
$ if ss$_tmp .eqs. " " then return
$ p'ss$_c' = 'ss$_tmp'
$ ss$_c = ss$_c + 1
$ if ss$_c .lt. 8 then goto PARAM_LOOP_1
$ return
$ !
$REPLACE_CALL_EXIT: ! change EXIT to GOSUB CALL_EXIT_SUB
$ ss$_beg = f$extract (0, ss$_exit, ss$_cmd)
$ ss$_cmd = ss$_beg + " GOSUB CALL_EXIT_SUB "
$ return
$ !
$REPLACE_EXIT:     ! change EXIT to GOTO EXIT
$ ss$_beg = f$extract (0, ss$_exit, ss$_cmd)
$ ss$_cmd = ss$_beg + " GOTO EXIT "
$ return
$ !
$REPLACE_EOF:      ! change /END = <label> to /END = END_TRAP (the SS label)
$ ss$_beg = f$extract (0, ss$_eof + 5, ss$_cmd)
$ ss$_end = f$extract (ss$_eof + 5, (ss$_len - ss$_eof - 5), ss$_cmd)
$ ss$_rst = f$locate ("/", ss$_end)
$ if ss$_rst .eq. f$length (ss$_end) then ss$_rst = f$locate (" ", ss$_end)
$ ss$_end_prm = f$extract (0, ss$_rst, ss$_end)
$ ss$_end = f$extract (ss$_rst, (f$length (ss$_end) - ss$_rst), ss$_end)
$ ss$_cmd = ss$_beg + "END_TRAP" + ss$_end
$ goto LOOKFOR_ERROR                                ! for possible /ERROR =
$ !
$REPLACE_ERROR:    ! change /ERROR = <label> to /ERROR = ERR_TRAP (the SS label)
$ ss$_beg = f$extract (0, ss$_error + 7, ss$_cmd)
$ ss$_end = f$extract (ss$_error + 7, (ss$_len - ss$_error - 7), ss$_cmd)

```

```

$ ss$_rst = f$locate ("/", ss$_end)
$ if ss$_rst .eq. f$length (ss$_end) then ss$_rst = f$locate (" ", ss$_end)
$ ss$_error_prm = f$extract (0, ss$_rst, ss$_end)
$ ss$_end = f$extract (ss$_rst, (f$length (ss$_end) - ss$_rst), ss$_end)
$ ss$_cmd = ss$_beg + "ERR_TRAP" + ss$_end
$ return
$ !
$ !-----
$ ! traps
$ !-----
$ !
$GOTO_SUB:                                ! "goto" by setting the line_no to branch location
$ gosub GET_LINE_NO
$ if ss$_lbl_ln_no .eq. 0 then return
$ if ss$_if_lvl .gt. 0
$   then ss$_say ss$_err_brniif
$     ss$_if_lvl = 0
$ endif
$ ss$_ln_no = ss$_lbl_ln_no - 1             ! anticipate get_cmd adding one
$ ss$_brnch = "T"
$ return
$ !
$GOSUB_SUB:                               ! "gosub" by setting the line_no to branch location
$ gosub GET_LINE_NO
$ if ss$_lbl_ln_no .eq. 0 then return
$ ss$_ret_ln = ss$_ln_no + 1
$ if ss$_$_cmd then ss$_ret_ln = ss$_ln_no ! $gosub returns to same line
$ ss$_ln_no = ss$_lbl_ln_no - 1           ! anticipate get_cmd adding one
$ ss$_brnch = "T"
$ return
$ !
$RETURN_SUB:                              ! subroutine "return"
$ if ss$_ret_ln .eq. 0 then goto NESTING_ERROR
$ ss$_ln_no = ss$_ret_ln - 1             ! anticipate get_cmd adding one
$ ss$_ret_ln = 0
$ ss$_brnch = "T"
$ return
$ !
$CALL_SUB:
$ gosub GET_LINE_NO
$ if ss$_lbl_ln_no .eq. 0 then return
$ ss$_ret_ln = ss$_ln_no + 1
$ if ss$_$_cmd then ss$_ret_ln = ss$_ln_no ! $gosub returns to same line
$ ss$_ln_no = ss$_lbl_ln_no - 1           ! anticipate get_cmd adding one
$ ss$_brnch = "T"
$ ss$_in_cll = "T"
$ return
$ !
$CALL_EXIT_SUB:
$ if ss$_ret_ln .eq. 0 then goto NESTING_ERROR
$ ss$_ln_no = ss$_ret_ln - 1             ! anticipate get_cmd adding one
$ ss$_ret_ln = 0
$ ss$_brnch = "T"
$ ss$_in_cll = "F"
$ !
$ ss$_c = 1
$PARAM_LOOP_2:                             ! restore p1-p7
$ ss$_tmp = ss$_p'ss$_c'sav
$ p'ss$_c' = "'ss$_tmp'"
$ ss$_c = ss$_c + 1
$ if ss$_c .lt. ss$_prm_cnt then goto PARAM_LOOP_2
$ ss$_prm_cnt = 0
$ return
$ !
$NESTING_ERROR:
$ ss$_say ss$_err_nogoseb
$ return
$ !
$END_TRAP:                                  ! for /END =
$ ss$_loc_prm = ss$_end_prm
$ gosub GOTO_SUB
$ goto NEXT_CMD
$ !

```



```

$ERR_TRAP:                                     ! for /ERROR =
$ ss$_loc_prm = ss$_error_prm
$ gosub GOTO_SUB
$ goto NEXT_CMD
$ !
$GET_LINE_NO:                                  ! ss$_loc_prm set by TRAP_BRANCHES
$ if f$locate ("% ", ss$_loc_prm) .eq. 0 then goto LINE_NUM
$ !
$ !-----
$ ! must be a label
$ !-----
$ !
$ ss$_lbl = f$edit (ss$_loc_prm, "lowercase")
$ gosub FIND_LABEL
$ return
$ !
$LINE_NUM:
$ ss$_lbl_ln_no = f$extract (1, f$length (ss$_loc_prm) - 1, ss$_loc_prm)
$ if f$extract (1, 1, f$edit (ss$_ln'ss$_lbl_ln_no, "trim, collapse")) -
    .eqs. "!" then goto COMMENT_LINE
$ if ss$_lbl_ln_no .le. ss$_last_ln then return
$ ss$_say ss$_err_lnbeof
$ ss$_lbl_ln_no = 0
$ return
$ !
$COMMENT_LINE:
$ ss$_say ss$_err_nocmdl
$ ss$_lbl_ln_no = 0
$ return
$ !
$FIND_LABEL:                                   ! ss$_lbl set by caller
$ if f$type (ss$_lbl_is 'ss$_lbl') .eqs. "" then goto LABEL_NOT_FOUND
$ ss$_lbl_ln_no = ss$_lbl_is 'ss$_lbl'
$ return
$ !
$LABEL_NOT_FOUND:
$ ss$_say ss$_err_lbnfnd
$ ss$_lbl_ln_no = 0
$ return
$ !
$ !-----
$ ! display code
$ !-----
$ !
$DISPLAY_CMD:                                  ! display cmd at ss$_ln_no in center of listing window
$ if (ss$_ln_no .gt. ss$_prv_ln) .and. -
    (ss$_ln_no .lt. ss$_prv_ln + 17) then goto SCROLL_UP
$ if (ss$_ln_no .lt. ss$_prv_ln) .and. -
    (ss$_ln_no .gt. ss$_prv_ln - 17) then goto SCROLL_DOWN
$ if (ss$_ln_no .eq. ss$_prv_ln) then return
$ !
$ !-----
$ ! else no common lines -> repaint
$ !-----
$ !
$DO_REPAINT:
$ ss$_say ss$_CSI, "2J"
$ ss$_say/symbol ss$_scr
$ ss$_say ss$_c_off, ss$_lst_area
$ ss$_dsp_ln = ss$_ln_no - 8
$ ss$_curs_ctl = ss$_CSI + "2;1H"
$ !
$REPAINT_LOOP:
$ ss$_dsp_ln = ss$_dsp_ln + 1
$ ss$_in_rng = (ss$_dsp_ln .gt. 0) .and. (ss$_dsp_ln .le. ss$_last_ln)
$ if ss$_in_rng then ss$_txt = -
    " " + f$fao ("!3SL:", ss$_dsp_ln) + " " + ss$_ln'ss$_dsp_ln
$ if .not. ss$_in_rng then ss$_txt = " "
$ if ss$_dsp_ln .gt. ss$_ln_no + 8 then goto REPAINT_DONE
$ ss$_say ss$_curs_ctl, ss$_EL, ss$_txt           ! erase old & write new
$ ss$_curs_ctl = ""
$ goto REPAINT_LOOP
$ !

```

```

$REPAINT_DONE:
$ ss$_say ss$_EL, ss$_txt, ss$_ctrl_area, ss$_c_on
$ ss$_prv_ln = ss$_ln_no
$ if ss$_wtc then gosub DO_WATCH
$ return
$ !
$SCROLL_UP:
$ ss$_say ss$_c_off, ss$_lst_area, ss$_CSI,"18;1H"
$ ss$_dsp_ln = ss$_prv_ln + 9
$ !
$UP_LOOP:
$ ss$_dsp_ln = ss$_dsp_ln + 1
$ ss$_in_rng = (ss$_dsp_ln .le. ss$_last_ln)
$ if ss$_in_rng then ss$_txt = -
  " " + f$fao ("!3SL:", ss$_dsp_ln) + " " + ss$_ln'ss$_dsp_ln
$ if .not. ss$_in_rng then ss$_txt = " "
$ if ss$_dsp_ln .eq. ss$_ln_no + 9 then goto UP_DONE
$ ss$_say ss$_EL, ss$_txt
$ goto UP_LOOP
$ !
$UP_DONE:
$ ss$_say ss$_EL, ss$_txt, ss$_ctrl_area, ss$_c_on
$ ss$_prv_ln = ss$_ln_no
$ return
$ !
$SCROLL_DOWN:
$ ss$_say ss$_c_off, ss$_lst_area, ss$_CSI,"2;1H"
$ ss$_dsp_ln = ss$_prv_ln - 7
$ !
$DOWN_LOOP:
$ ss$_dsp_ln = ss$_dsp_ln - 1
$ ss$_in_rng = (ss$_dsp_ln .gt. 0)
$ if ss$_in_rng then ss$_txt = -
  " " + f$fao ("!3SL:", ss$_dsp_ln) + " " + ss$_ln'ss$_dsp_ln
$ if .not. ss$_in_rng then ss$_txt = " "
$ if ss$_dsp_ln .eq. ss$_ln_no - 7 then goto DOWN_DONE
$ ss$_say ss$_CSI, "1A", ss$_CSI,"1L", ss$_txt
$ goto DOWN_LOOP
$ !
$DOWN_DONE:
$ ss$_say ss$_CSI, "1A", ss$_CSI, "1L", ss$_txt, ss$_ctrl_area, ss$_c_on
$ ss$_prv_ln = ss$_ln_no
$ return
$ !
$ !-----
$ ! errors
$ !-----
$ !
$FILE_NG:
$ ss$_say ss$_err_notope, ss$_fil_nam
$ exit

```