

Toshiba 4690 Application Debugger
Version 4
User's Guide Product 5799-RXD

May, 2013

ABOUT THIS MANUAL

This manual describes the Toshiba 4690 Application Debugger Tool which helps to debug 16-bit applications on the 4690 system. The debugger can debug terminal applications or controller applications that run on the 4690 Operating System.

Tenth Edition (May, 2013)

Special Notices:

The following terms used in this publication, denoted by a double asterisk(**), are trademarks of other companies:

- Integrated Systems, Inc. (R)
- Meta Ware (R)
- Microsoft Windows 98, NT, 2000, XP (R)
- International Business Machines Corporation (R)

Copyright © 2012 Toshiba Global Commerce Solutions (TGCS), Inc. All rights reserved

Note to US Government Users: Documentation relates to restricted rights. Use, duplication, or disclosure is subject to restrictions that are set forth in GSA ADP Schedule Contract with Toshiba Global Commerce Solution Inc.

INTRODUCTION TO THE 4690 APPLICATION DEBUGGER	5
System Requirements	5
Hardware	5
Software	6
Installation	6
Compilation of 4690 BASIC programs for debugging	8
Compilation of C programs for debugging	9
USING DEBUG	10
Starting DEBUG	10
The Debugger Screen	11
Screen Control Keys	12
Screen Colors	12
Source Screens	14
Assembler Screen	15
Action Bar Options	16
General Keyboard Functions	18
Variables	20
Variables Known to the Debugger	20
4690 BASIC Chaining	26
Multiple DEBUG Sessions	26
Debugging Terminals in off-line mode	26
Actions That Affect Multiple Breakpoints	26
Files that are Used by the Debugger	27
Keystroke Recording and Playback	28
Using Data Breakpoints	30
Displaying the CBASIC Call History	34
Displaying the list of CBASIC Open Files	37
Displaying the CBASIC Error Trace	38
Displaying the CBASIC ON ERROR Routine	38
Interpreting CBASIC Error Codes	38
Appendix A. Source Window Keys	40
Appendix B. Assembler Window Keys	41
Appendix C. Storage Window Keys	42
Appendix D. Helpful Hints	42
Finding Your Source and other Files	43
Source Files	43
Executable Files	43
Finding and Displaying Source Files	43

Function Call Trace	43
Single Step Delay	44
Single Step Problems	44
Missing or Mismatched Source Files	44
Where is a Specific Variable Used?	45
Finding Non-Public Functions	45
Scoping Problems with Variables	45
Tips with Examples	46
Appendix E. Problem Determination	47
Assembler Screen Displayed	47
Unable to start Session error	47
Unable to Display Variable in the Storage Window	47
Unable to run Debugger in the Terminal	48
Unable to start Help Session error	48
Executable Lines incorrectly displayed	48
Unable to see the correct Application Screen	48
String Variable is displayed as a Long Integer	48
Application Abends after using Ctrl-F6 to set CS:IP	49
Known Problems and Solutions	50
Debugger hints	53
If you do have a problem	53
Application Debugger Support	53
Debugger Tools	54
Appendix F. Information for Microsoft Windows Users	55
Appendix G. Program Messages and Actions	56
DEBUGFRM Messages and Actions	56
DEBUGCDV Messages and Actions	57
DEBUG Messages and Actions	59
Appendix H. 4690 Application Debugger Reference Card - February, 2011	69

INTRODUCTION TO THE 4690 APPLICATION DEBUGGER

The 4690 APPLICATION DEBUGGER is a debugger for 4690 BASIC and C application programs running under 4690 OS Version 2 Release 1 CD 9001 Delta 3 or later. The debugger is designed to help speed up program development and 4690 debugging. The 4690 Application Debugger allows:

- Debug of 4690 BASIC or Meta Ware (R)** "C" programs (16-bit only)
- Display or altering of variables and arrays while debugging (Note 1.)
- Preparing debugger files on Microsoft Windows (R) or 4690 platforms (Note 2.)
- Debug of controller, terminal or Terminal Session Server (TSS) application programs
- Debug of foreground or background controller applications
- Debug of on-line or off-line terminal applications
- Setting of code and data break points, single stepping, changing variables by name or displaying the function call trace (Note 3)
- Display of the contents of 4690 pipes, application screens or a program's load segments
- Display of the CBASIC call history (Note 4), CBASIC open session numbers and file names, or CBASIC error trace (Note 5).
- Display of your application in the source language or in the resultant assembler code.

NOTE: 1. Enhanced Debugger Functions such as array support, local variables, and automatic type definitions are not available with the 4690 BASIC prior to Version 3.

NOTE: 2. Application development can be on Microsoft Windows or on 4690. 2000 Professional is the recommended Windows platform, although some testing has been performed on Windows 98, NT and XP platforms. DEBUGFRM.EXE and DEBUGCDV.EXE will run on Microsoft Windows. DEBUGFRM.286 and DEBUGCDV.286 will run on 4690. To compile and link BASIC programs on Microsoft Windows, see "Appendix F. Information for Microsoft Windows Users" on page 54 for more information.

NOTE: 3. 4690 OS V3R2 or later is required in order to use data breakpoints.

NOTE: 4. CBASIC Call History requires an updated SB286L.L86 or SB286LT.L86, released in February 2003 with defect 8000920., or a later release.

NOTE: 5. The format of the CBASIC error trace was changed in November, 2004. See "Displaying the CBASIC Error Trace" on page 37.

NOTE: 6. Some early versions of 4690 V6 Enhanced do not support this debugger. See NOTE 3 on the next page.

Notice to users of previous versions of this debugger: Support for DEBUGDMP has been withdrawn. Support for debugging of COBOL applications has been withdrawn. Support for DOS has been withdrawn.

System Requirements

Hardware

Computer

The Debugger supports any 4690 Controller (PC or PS/2**) that is running the 4690 Operating System. It supports point of sale terminals attached to this controller, such SurePOS 4800 series, 4693 and 4683 Model 1 terminals. Auxiliary console is not supported.

Memory

The Debugger module uses 800 Kb (or more) of store controller memory, and about 40 Kb of memory in the terminal. Symbols and source code adds to the amount of controller memory required. With a large application, this could be several hundred KB or more.

Software

The following software is required:

- 4690 OS. (Notes 1, 3)
- LINK86 "TM" Version 2.39 or later. (Note 2)
- 4690 Application Debugger Version 4 level 18 or later.
- 4690 BASIC Version 3 compiler for BASIC programs. (Note 2)
- Meta Ware (R)** HighC compiler Version 1.7 for C language programs

NOTE: 1. 4690 V3R2 or a later release is recommended, although most functions of the debugger will work at any level of 4690 OS. Certain functions (CTRL-BRK, Data Breakpoints) require 4690 V3R2 or a later release..

NOTE: 2. The 4690 Application Debugger will work with 4690 BASIC version 2 and LINK86 "TM" Version 2.29A. However, the debugger functions and features will be limited. There will be no local variable or array support.

NOTE: 3. Beginning with V6, 4690 supports an "Enhanced" mode that uses Embedded Linux for an operating system kernel. This debugger will not run on early versions of V6 Enhanced because the necessary debugging hooks into the Operating System are unavailable in those versions. If you attempt to start the debugger, you will see a popup box with one of these return codes: 80404015, 80004305, 80004005. The minimum level of 4690 OS maintenance to support the debugger on "Enhanced" is V6R2 with maintenance level 0BF1 or later. It was tested on level 0AH0 plus OS fix 21644. The debugger continues to run on V6R1 and V6R2 "Classic" as it did on prior OS versions and releases.

Operational Differences and Limitations with 4690 OS V6 - Enhanced

1. Data breakpoints cannot be activated. There is no support in 4690 OS - Enhanced for that type of function, even with the maintenance level fixes listed above. The debugger will allow you to set data breakpoints or change them, but a break will not occur when the variables you specified are changed while your program runs.
2. Some debugger screens show up-arrow and down-arrow characters to indicate that you can use those keys to change the focus, or to point to a particular item. The tip of the arrow for those characters may not be as well defined on graphic screens as they were on character-based screens.
3. Time stamps default to 10 millisecond granularity. This is mainly evident on the CBASIC Call History screen. If you need better granularity, see the description for SETTRACE in the Knowledge Base article "CBASIC Application Tools". SETTRACE 4609 is one use that will give one-millisecond granularity.
4. Paging up or down in a Source or Assembler window is a little slower because of the additional overhead of managing graphic screens rather than character-based screens.
5. Some color combinations that the debugger uses to highlight certain lines may be difficult to read due to the narrower, graphical presentation of the bitmapped characters. See DEBUGGER.TIP on the installation diskettes or CD-Rom disc for some things you can try to make the text a little more readable.

Installation

The debugger install program creates debugger logical names. Therefore, the debugger will not operate correctly unless the install program is used. For installation instructions, refer to the README.DOC on the installation diskette or the CD-ROM disc. The complete debugger is available on two diskettes or one CD-ROM disc. To install using diskettes, insert the first debugger diskette in the A: drive, go to a

command prompt and type A:INSTALL. To install using the CD-ROM disc, insert the CD-ROM disc in the drive, go to a command prompt and type P:/INSTALL.

Note to prior users of this debugger : Refer to README.DOC on the first installation diskette or the root folder of the CD-ROM disc for a list of fixes and new function that is added since Version 3 of the debugger, which was released in 1995.

Compilation of 4690 BASIC programs for debugging

Two versions of the 4690 BASIC Compiler are available. BASIC Compiler versions prior to 3.0 limit debugger function. The steps shown below apply to BASIC version 3.0 or later.

Use the BAS.BAT file, included on the debugger diskette or CD-ROM disc, to prepare BASIC programs for debugging or use these five steps. A detailed explanation follows.

1. "BASIC filename [DF]"
2. "DEBUGFRM filename"
3. "LINK86 filename [S,MAP[A],DBI]"
4. "POSTLINK filename"
5. "DEBUGCDV filename"
6. "DEBUG filename"

A detailed explanation of the six steps follows:

1. "BASIC filename [DF...]" to compile a filename.BAS file. The "D" option generates debug information within the .OBJ file and the "F" option saves the listing file. The listing file is used in the next step.

NOTE: If you do not have executable code in include files, your compiler syntax becomes "BASIC filename [DG...]" and the DEBUGFRM step is not needed. If you do not run DEBUGFRM, then the debugger will use the .BAS file as a listing file instead of the .LCV file.

2. "DEBUGFRM filename nn" where nn is the read buffer size of 1 to 31 KB, the default is 16 KB. This operation adjusts the listing file for the debugger and creates a filename.LCV file. This LCV file will serve as the source file during debug. You must run DEBUGFRM for each listing file / object file.
3. "LINK86 filename [S,MAP[A],DBI]" to Link and create a filename.286 file. This operation creates the .MAP, .LIN and .SYM files for DEBUGCDV. The S option instructs LINK86 to search the library and only link externally referenced modules. The MAP[A] option creates the .MAP file and includes all symbols from the shared runtimes. The DBI option creates the necessary information for the debugger. The load module can be shared or not shared. Do not use the NOLIN option. If the NOLIN option is used with the DBI option, then line numbers will not be generated. The debugger needs line numbers to display source code.

NOTES: Do not use the DBI option with versions of 4690 BASIC prior to Version 3. If you are using .INP files to combine .OBJ and/or .L86 modules, select the DBI option on the first .OBJ file in the .INP file. Once the DBI option is set on in an .INP file, it will remain on for the rest of the file. Refer to the Toshiba 4690 Programming Guide for more information.

4. "POSTLINK filename" to create a postlinked filename.286 file required for debugging. All terminal programs must be post linked.
5. "DEBUGCDV filename /F" to create a filename.CDV file required for debugging. The /F option is used to make the .CDV file generation faster while losing additional global type information.

NOTES: Do not specify the same .OBJ name in a library and in the file list.

6. "DEBUG filename" to start the debugger. Filename refers to the .286 name created by the previous operations. The debugger will require the .286, .CDV, and source or .LCV files. Refer to "Starting DEBUG" on page 9 for information on options and arguments.

Compilation of C programs for debugging

The only C compiler supported by the Application Debugger is the MetaWare (R)** High C (R) Compiler 1.7 available through Integrated Systems, inc. (R)**.

Use the MW.BAT file, included on the 4690 Application Debugger diskette or CD-ROM disc, or use the following 4 steps to prepare C code for debugging:

1. "HC filename -mm big -DEBUG -ON CODEVIEW -OFF OPTIMIZE_XJMP -OFF OPTIMIZE_FP -OFF OPTIMIZE_XJMP_SPACE"
2. "LINK86 filename [S,MAP[A],DBI],HCBE.L86[S]"
3. "POSTLINK filename"
4. "DEBUGCDV filename"
5. "DEBUG filename"

A detailed explanation of the five steps follows:

1. "HC filename -mm big -DEBUG - ON CODEVIEW -OFF OPTIMIZE_XJMP -OFF OPTIMIZE_FP -OFF OPTIMIZE_XJMP_SPACE" to compile a filename.C file. The "-mm big" option uses the BIG memory model for compiling. The "-DEBUG" and "-ON CODEVIEW" options provide support for the debugger. The "-OFF OPTIMIZE_..." options turn off optimization features that are confusing while debugging. For your production level compile, using optimization options can significantly enhance the performance of your program. The DEBUG and ON CODEVIEW options can be turned on for your final compile.
2. "LINK86 filename [S,MAP[A],DBI],HCBE.L86[S]" to Link and create a filename.286 file. This operation creates the MAP, LIN and SYM files for the DEBUGCDV operation. The S option instructs LINK86 to search the library and only link externally referenced modules. The MAP[A] option creates the MAP file and includes all symbols from included libraries. The DBI option creates the necessary information for the debugger. Specify HCBE if you use the BIG memory model. If you are using an .INP file, make sure that these options are included.
3. "POSTLINK filename" to create a postlinked filename.286 file required for debugging.
4. "DEBUGCDV filename" to create a filename.CDV file required for debugging.
5. "DEBUG filename" to start the debugger. Filename refers to the 286 name created by the previous operations. The debugger will require the source, .CDV and .286 files. Refer to "Starting DEBUG" on page 9 for information on options and arguments.

NOTE: If you are using .INP files to combine .OBJ and /or .L86 modules, select the DBI option on the first .OBJ file in the .INP file. Refer to the Toshiba Store System Programming Guide for more information.

USING DEBUG

Starting DEBUG

The debugger must be able to find the .286, .CDV, and source files to function correctly. If you used BAS.BAT or the "F" BASIC compiler option and executed DEBUGFRM, the .LCV file serves as the BASIC source file (Note 2). The .286 and .CDV files must be in the same subdirectory.

The syntax for starting the debugger is:

DEBUG [OPTIONS] FILENAME [ARGUMENTS]

OPTIONS: may be any of the following. Each option must be prefaced with a dash (-) or a slash (/) or a backslash (\) and separated by a space.

- /B Debug a 4690 Background Application. (Note 1)
- /C Forces case sensitivity for variable names.
- /I Debug initialization or startup code. (Note 3)
- /M Use the Medium Memory Model. The debugger default is large memory model.
- /Tnnn Debug a Terminal Application (nnn = terminal number).
- /Vnnn Debug a Terminal Session Server Application (nnn = terminal number). (Note 4)
- /wnn Wait for nn minutes. This option will allow time for an application to load. If the application loads in less time, this option has no affect. Use this option for terminal loads in a busy environment.
- /Df=n Assigns display colors. f is an integer from 1 to 24 and denotes the type of field you want to change. n is an integer from 0 to 127 and denotes the new background and foreground color mask. See DEBUGGER.TIP on the second installation diskette, or folder DISK2 on the CD-ROM disc, for examples of use, and why you might want to use this parameter.
- /on Program Options. n=1 buffers writes to DEBUGLOG, rather than flushing every write. n=2 bypasses validity checking of LCV files against object code. n=4 writes 4000 bytes of screen data to file DEBUG.SCR for each keystroke. The screens in this document were created by using this option. n=0 logs error messages only, n=8 logs messages and writes them to the screen. n=16 changes the Name Lengths default from 11 to 76 characters. n=32 turns off application memory caching. N=64 traces the storage window to DEBUGLOG. Options can be combined numerically. This flag is mainly for TGCS use.

FILENAME: the name of the 286 file you wish to debug.

The 286, source files and CDV file must be available to the debugger. For BASIC code, the LCV file serves as the source file. The debugger will always search for a file with .LCV to use as the source file. The debugger searches for your program's source files in this order:

1. Specified path

2. Current directory
3. Environment variable DEBUGSRC (Note 2)
4. Path

ARGUMENTS: the standard input character strings that are passed to the program. The total size of these strings, including spaces, must be less than 50 characters.

NOTE 1: If you are debugging a background or terminal application, you must enter the full path name. The Background or Terminal application will load its program from the file name specified. If you enter only the file name, the loader assumes that the file is in the current directory.

The filename plus the path is limited to 21 characters, due to 4690 OS limitations. If your terminal program files (i.e. terminal.286) are in a subdirectory called TESTFILE, you need to create a logical name for the subdirectory. For example:

```
DEBUG /T123 C:\TESTFILE\TERMINAL
```

This will not work because the system adds the .286 extension and exceeds the 21 character limit. To correct this, create a system logical name of TEST: to define C:\TESTFILE\, then your syntax becomes:

```
DEBUG /T123 TEST:TERMINAL
```

NOTE 2: Use the DEBUGSRC environment variable for source files that are not in the same directory as the .CDV and .286 files, or if the directory is not in the path. For example, if your source is contained in C:\$MYPROG\$SRC and your .CDV and .286 files are in C:\$MYPROG\$OBJ, and if \$MYPROG\$SRC is not in the path and is not the current directory, then you should set the DEBUGSRC environment variable to C:\$MYPROG\$SRC before invoking DEBUG. To set or reset this environment variable:

```
DEFINE DEBUGSRC = C:$MYPROG$SRC
```

Remember that the DEBUGFRM operation creates a filename.LCV file which the debugger uses as the source file for 4690 BASIC files.

NOTE 3: You must have the source code and .CDV file for the debug initialization or start-up code option to work.

NOTE 4: Refer to "Terminal Session Server" online help while running the debugger, or contact your TGCS support representative for details. The main difference is that you need to change your TSS configuration to start DEBUG:EWTTTERM.286 instead of the name of the application that you are trying to debug.

The Debugger Screen

The debugger screen consists of an action bar across the top, a body containing the source or assembler code, and a status line at the bottom. Context-sensitive help is available on all screens by pressing F1 for a help screen.

The action bar is a set of pull-down windows to assist you with the functions available in DEBUG. Pressing F10 accesses it. Then you can highlight the desired option by using the arrow keys. Press ENTER to activate the highlighted option. Press F1 to display a help screen for the highlighted option.

```
Display Window Search Commands Exit Help
```

The body or center of the screen displays the source code or assembler code that you are debugging. The reverse video line is the next line to be executed. Executable source lines are highlighted; comment lines are not. An underline or the color red (depending upon your display configuration) represents a breakpoint. An overlay memory window can be selected to display on the source screen near the top.

Status is at the bottom. On the source screen, a single line displays the name of the source file that you are debugging. The line below the source file name is the message line. Any system message or variable value is displayed on the message Line.

```
DEBUG:EAMTS10L.CDV <DEBUG:EAMEETSC.LCV> Line 7015 of 7947
Stopped at Breakpoint in <EETSEC29>
```

On the assembler screen, the file name line and the message line display the same data. Above these lines are the two lines that display the processor registers and flags. The registers display in hexadecimal and flag values display in binary.

```
AX 0000 BX 7F7C CX 02F7 DX 028B SI 0002 DI 0002 SP 7F86 IP 1738
Z=0 C=0 S=0 O=0 D=0 P=0 A=0 BP 7FF2 DS 0067 ES 05A7 SS 0577 CS 02F7
DEBUG:EAMTS10L.CDV <DEBUG:EAMTS11C.LCV>
```

Screen Control Keys

The screen control keys are used to help you debug applications: A short description of their function is available while running the debugger by pressing F1 for help followed by F9. (see "General Keyboard Functions" on page 17 for more information on keys.)

Screen Colors

The debugger displays fields in either monochrome or color, depending upon which type of monitor you are using. A color monitor is strongly recommended for higher productivity.

- Intense/white

This is an executable line of source-code or one machine-level instruction.

```
DIM EE.REG.E$(5,10)
```

- Blue on white

This is an executable line (at source-level), however it probably does not match the OBJ/CDV file.

The debugger flags all source lines in this way if the line information from the CDV file indicates that the line is an executable line of code, but the first non-blank character is a "!" or "\". This problem usually occurs because you did not copy the matching version of the LCV file into the DEBUG directory for the OBJ file being displayed. An error message is also written to DEBUGLOG for each mismatching line.

This type of checking is only appropriate for CBASIC programs. If you want to turn off checking for this type of error, use the /O2 (the letter Oh, not the number zero) flag when starting the debugger.

```
\ TS.GROSSNEG = 0
```

- Normal/yellow

This is a non-executable line in the source level (for example, a statement, a declaration, or a statement continued from a previous line).

NOTE: For multi-line statements, only the first line is shown as an executable line of code. All continuation lines display as comments. For example, if you use a backslash in 4690 BASIC to extend a statement to several lines, the debugger will only show the first line as executable code.

```
!AIR48940
END.TIME$ = RIGHT$("0" + STR$(SO.RESTEND(INDEX)),2) + \
            RIGHT$("0" + STR$(SO.RESTEND(MINCINDEX)),2)
IF END.TIME$ > "0000" THEN BEGIN
```

- Reverse video/white on black

The next line to execute in the program.

```
ON ERROR GOTO TS11END1
```

- Red on blue

A breakpoint was set for this line.

```
IF NOT TS.SIGNED_ON THEN BEGIN
```

- Underlined red on black

The current instruction is at a breakpoint.

```
IF NOT TS.SIGNED_ON THEN BEGIN
```

- White on blue, with dot patterns substituted for spaces

The return point after a function call. In the example below, the current instruction is in function TSRFECML.

```
CALL TSRFECML
TS.ET.RESET.TRX = 0
```

Source Screens

Following is a sample of a debugger screen that shows BASIC source code.

```
Source Code
Display Window Search Commands Exit Help
ON ERROR GOTO TS11EQ01 !
ON ASYNC ERROR CALL TS11EWAS !
DIM TS.ERR.ARRY(2,2) !
CALL EETSEC29 ! determine the eft type
COMPLETE.RESTART: !
IRRF.LOCAL.FILE$ = "R::$AMITEMR" !
TS.SIGNED.ON = 0 !
TS.GROSSPOS = 0 !
TS.GROSSNEG = 0 !
TS.IN.IPL = -1 !
TS.MSRSEP$ = CHR$(13)+CHR$(61) !
TS.MSR.ONKBD = -1 ! assume msr present
IF NOT TS.SIGNED.ON THEN BEGIN ! if not signed on, then
TS.IO.NEXTSTATE = 2 ! initial state
.\EAMIS10L.CDU <.\EAMIS11C.LCU> line 6746 of 11842
Toshiba 4690 Application Debugger
```

Figure 1. 4690 BASIC Source Code Screen

(see "Appendix A. Source Window Keys " on page 39 for more information on active keys.)

Selections are available from the action bar. Use F10 to access the action bar functions. Press the capital letter in the action bar item name to use that function. For example, to restart the application without exiting the debugger: press "C" for commands, "A" for restart application, and "Y" for yes to restart. Pressing the key sequence "CAY" will restart the application. Other examples are:

- "WA" to go to the assembler window
- "DF" to display function call trace
- "SS" to search for string
- "CC" to run to the cursor
- "CF" to single step into function
- "WP" to display the application screen

Assembler Screen

The following screen is a sample of an Assembler screen:

```
Assembler Code
Display Window Search Commands Exit Help
TS.SIGNED.ON
0 0x0000
02F7:178C INC SP
02F7:178D INC SP
02F7:178E CALL 0187H:567CH CALL EETSEC29 ! de 7055
02F7:1793 PUSH CS IRRF.LOCAL.FILE$ = 7059
02F7:1794 PUSH 73A3H
02F7:1797 PUSH DS
02F7:1798 PUSH 1B12H
02F7:179B CALL 006FH:009AH
02F7:17A0 MOV WORD [18F6H],0000H [0000] TS.SIGNED.ON = 0 ! 7060
02F7:17A6 MOV WORD [1944H],0000H [0000] TS.GROSSPOS = 0 ! 7061
02F7:17AC MOV WORD [1946H],0000H [0000]
02F7:17B2 MOV WORD [1948H],0000H [0000] TS.GROSSNEG = 0 ! 7062
02F7:17B8 MOV WORD [194AH],0000H [0000]
02F7:17BE MOV WORD [198EH],FFFFH [FFFF] TS.IN.IPL = -1 ! 7064
02F7:17C4 PUSH 003DH TS.MSRSEP$ = CHR$( 7065
02F7:17C6 CALL 00BFH:0003H
02F7:17CB PUSH 0000H
AX 0000 BX 0000 CX 0000 DX 000B SI 7380 DI 1465 SP 7F86 IP 17C6
Z=1 C=0 S=0 O=0 D=0 P=1 A=0 BP 7FF2 DS 020F ES 05A7 SS 0577 CS 02F7
DEBUG:EAMTS10L.CDV <DEBUG:EAMTS11C.LCV>
Instruction Step in <main>
```

Figure 2. Assembler Code Screen

See "Appendix B. Assembler Window Keys" on page 40 for more information on active keys.

Selections are available from the action bar. Refer to the Source Screen and the Action Bar Options section for more information.

Action Bar Options

The options pull-downs available on the ACTION BAR are Display, Window, Search, Commands, Exit, and Help.

Pull Down	Option	Explanation
Display	Variable	Find and display a variable name. You will be asked to enter the variable name. This will only display global variables or local variables of the function you are executing. To display other variables use the insert key with the cursor on the variable name.
	What var points to	Display the contents of the memory location that a given variable is pointing to. You will be asked to enter the name of the pointer.
	Memory	Display the contents of the memory location specified. You will be asked to enter the memory location in the hexadecimal format segment:offset.
	Function call trace	Trace the functions that were called since the test program started. This is a history of the functions previously called.
	Segment information	Display the segment descriptors of the program you are debugging.
	Pipes	Display the data contained in an active pipe. Display information about application pipes. Selecting this option will cause a window to appear with a list of the active pipes for the application. From this window you can choose the pipe that you want to view.
	cbasic Call history	Display the function call trace maintained by CBASIC. This trace shows the history of all functions called up to that point. The trace wraps after 25 entries.
	cbasic Open files	For all open CBASIC sessions, display the session number and the file name.
	cbasic Error trace	Display the ERR, ERRN and ERRF information from the last fifteen CBASIC runtime errors.
	cbasic ON error routine	Display the source code that was established with the last ON ERROR statement.
	Data break points	Display the names of the currently-set data breakpoint variables, and the conditions and masks that cause a stop to occur.
	code Break points	Not implemented (yet).
Window	Assembler	Switch to the assembler window from the source code window. (Source Window Only)
	sORce	Switch to the source code window from the assembler window. (Assembler Window Only)
	Storage	Overlay the current screen with the storage window. The storage window displays the data in variables and memory.
	Register	Change the contents of the registers displayed across the bottom of the screen. (Assembler Window Only)
	aPplication	Switch to the application screen. This is the screen that the user or operator would normally see.
	Next file	Switch to the next available window. This option does nothing if only one window is active.
Search	Search for string	Locate the first occurrence of a given string. You will be asked to enter the string to be found. This will only search the displayed module. (Source Window Only)
	find Next string	Locate the next occurrence of a given string. It searches only the currently displayed module. (Source Window Only) NOTE: from a source window, N is the same as S N.
	find Previous string	Locate the previous occurrence of a given string. It searches only the currently displayed module. (Source Window Only) NOTE: from a source window, P is the same as S P.
	search for	Locate a function. You will be asked to enter the name of the function.

	Function	The search routine will only look for global/public functions. You can enter segment:offset as a function on the assembler screen.
	search for Variable	Locate where a variable is used.. You will be asked to enter the name of the variable, and then to select from the list of modules that use that variable. The cursor is then positioned on an occurrence of the variable in the selected source module. (Source window only.)
Commands	Single step	Single step the program in the current window. This performs the same function as the spacebar. If the current program calls a function, this function will free-run.
	set/clear Break point	Set or reset a program execution breakpoint on the cursor line. Performs the same function as F4. If debugging at source level, you may notice that not all lines may be selected for breakpoints. This is because your compiler generates a table of line numbers versus module addresses. The debugger can set breakpoints only on those lines that have an entry in this table. For example, you cannot set breakpoints on comments, some declarations, some terminating braces, or various other places. Resetting a breakpoint at the Source level resets the breakpoints on all assembler instructions covered by that source line.
	run to Cursor	Set a breakpoint at the cursor and run the target program until a breakpoint or event occurs. Same function as F5. The breakpoint at the cursor is removed after it is reached.
	Run	Run the target program until a breakpoint or event occurs. Same function as F6.
	run Without breakpoints	Temporarily run the target program without breakpoints. Press <CTRL>+BRK when you want to regain control of the target program.
	s step into Function	Single step the target program (trace) with code breakpoints enabled. If the current statement calls another function, it will single step into that function. This is the same as F2. NOTE: When you step into a function, parameters and the call stack cannot be displayed correctly until the function prologue code is executed. Press the spacebar or F2 to execute the function prologue code.
	find Executing line	Switch to the file that contains the statement or instruction that is about to be executed, and place the cursor on that line. Same function as F7.
	re-start Application	Restart the application from the beginning. All break points are cleared and all memory variables remain active in the storage screen.
	Terminal Status	Changes the terminal operating mode to Offline or Online.
	toggle Name lengths	In the storage window, two lines are used to display a variable if its name is 12 characters or longer. The default will always use just one line, truncating long names to 11 characters.
	toggle Zero/null elements	In the storage window, array elements that contain a zero scalar value or a null string can either be shown on their own line, or suppressed to reduce clutter. The default is to suppress.
	toggle func bPs this file	Set (or clear) a breakpoint at the beginning of each function in the currently-displayed file.
Exit	Exit	Terminate the current debug session.
	Resume	Resume the current debug session. This is the alternative to exiting.
Help	Help for help	Instruction on how to use the help facility.
	Extended help	A more in-depth look at the 4690 Application Debugger.
	Keys help	A description of the keys and their function.
	help Index	An index facility of all the available help screens. From this index you can go directly to any of the available help screens.
	Cbasicrc	Interpret the current value of CBASIC variables ERR and ERRN.
	About	Provides an on line level of the program. If you need to know what version of a program you are running, this is a good place to look.

General Keyboard Functions

The following definitions are in affect in the source or assembler windows. The first column is the name of an active key on the keyboard. The next column is the name of the debugger function that applies to that key. The last column is a description of that function. The notes in this column identify functions that only operate on certain screens.

Key	Function	Explanation
Ctrl+Break	Stop	Stop running program or application being debugged. If the program is waiting on an I/O operation or executing in the operating system, CTRL+BREAK will not stop the program until that condition completes. You cannot CTRL+BREAK out of a condition whereby the application is waiting for input from a background screen.
Spacebar	SStep	Single step the target program (no trace) with code breakpoints enabled. If the current statement calls another function, it will not be stepped into.
Plus(+)	ShowVarWindow	Show the variable window with the selected variable values.
Minus(-)	RemVarWindow	Remove the variable window.
Equal(=)	ShowVar	Display the variable at the cursor (provided the variable is one which is known to the debugger). See "Variables Known to the Debugger" on page 19 for details. (Source Window Only)
*	ShowVarPtsTo	Display the data pointed to by the variable at the cursor (provided the pointer variable is one which is known to the debugger). See "Variables Known to the Debugger" on page 19 for details. (Source Window Only)
&	ShowVarAddress	Show all addresses of the name of the variable at the cursor.
Ins	PutVarInStg	Place the variable at the cursor into the storage window (provided the variable is one which is known to the debugger). See "Variables Known to the Debugger" on page 19 for details.
Up Arrow	UpCursor	Move the cursor up one line. When the top of the current window is reached, the window scrolls back one line.
Down Arrow	DownCursor	Move the cursor down one line. When the bottom of the current window is reached, the window scrolls forward one line.
Right Arrow	RightCursor	Move the cursor right one word. (Source Window Only)
Tab	TabRight	Move the cursor right one word. (Source Window Only)
Left Arrow, Backspace	LeftCursor	Move the cursor left one word. (Source Window Only)
Shift+Tab	TabLeft	Move the cursor left one word. (Source Window Only)
PgUp	PrevWindow	Scroll to the previous window in the current view.
PgDn	NextWindow	Scroll to the next window in the current view.
Home	TopOfWindow	Move the cursor to the first line of the current window. The cursor column is unchanged.
Ctrl+Home	FirstWindow	Scroll to the first window in the current view. The cursor column is unchanged. (Source Window Only)
End	BotOfWindow	Move the cursor to the last line of the current window. The cursor column is unchanged.
Ctrl+End	LastWindow	Scroll to the last window in the current view. The cursor column is unchanged. (Source Window Only)
Ctrl+Ins	PutVarPtsToInStg	Place the data pointed to by the variable at the cursor into the storage window (provided the variable is one which is known to the debugger). See "Variables Known to the Debugger" on page 19 for details.
Ctrl+Alt+Tab	ToggleAppWindow	Show the target program's screen.
F1	Help	Displays context-sensitive help message.
F2	SStepIntoFunc	Single step the target program with code breakpoints enabled. If the current statement calls another function, it will be stepped

		into and debugged. NOTE: When you step into a function, parameters and the call stack will not be displayed correctly until the function prologue code is executed. This generally involves stepping one additional statement.
Ctrl+F2	RunToCbasicCall	Free-run until any CBASIC function is called, then stop at the first executable statement in that function. Other code or data breakpoints that you already have set may get hit first. If so, you can press Ctrl+F2 again, or can press Run. Either key will stop at the next CBASIC function call. Ctrl+Brk does not cancel this breakpoint.
F3	Exit	End the debug session and the program being debugged.
F4	SetClearBkPt	Set or reset a program execution breakpoint on the cursor line. Breakpoints cannot be set on every line. This is because your compiler generates a table of line numbers versus module addresses. The debugger can set breakpoints only on those lines that have an entry in this table. For example, you cannot set breakpoints on comments, some declarations, some terminating braces, or various other places. Resetting a breakpoint at the Source level resets the breakpoints on all machine instructions covered by that source line.
Alt+F4, Ctrl+F4	SetClearDataBkPt	Set or reset a breakpoint on the address of a variable. A breakpoint will occur right after an attempt is made to store any value at that address. Up to four addresses can be set, due to Intel limitations. To reset a data breakpoint, place the cursor on that variable name in the source window and press Alt+F4 or Ctrl+F4 (again). 4690 OS V3R2 is required for this to work.
Shift+F4	SetClearDataPtrBkPt	Same as SetClearDataBkPt, except that the breakpoint address is obtained from the current value contained in the variable you are specifying, which must be a long pointer. Note that if the value of the pointer changes, the breakpoint address does not. To reset a data pointer breakpoint, place the cursor on that variable name in the source window and press Shift+F4 (again). 4690 OS V3R2 is required for this to work.
F5	RunToCursor	Set a breakpoint at the cursor and run the target program until a breakpoint or event occurs. The breakpoint at the cursor is not kept once it is reached.
F6	Run	Run the target program until a breakpoint or event occurs. Ctrl+Break will stop a program that has been started with the RUN command.
Ctrl+F6	ChangeCSIP	Set the CS:IP so that it points to the source statement where the cursor is located (source window only).
F7	FindExecLine	Switch to the file containing the statement or instruction that is about to be executed and place the cursor on that line. It also shows you the name of the currently-executing public function.
Alt+F7	FindLastExecLine	Switch to the file and source window that was being displayed when you last pressed F2, F5, or F6, or the last Search Function. You can use this after you hit a breakpoint in another function or area (or you pressed Ctrl+Break), and want to redisplay the previous screen so that you can more quickly inspect certain variables or set additional breakpoints.
Ctrl+F7	GotoParent	Display the statement that calls the current function. Your application must use a standard calling stack frame for this to work correctly.
F8	FindFunction	Locate the first occurrence of the function name at the cursor. This is the same as SEARCH FOR FUNCTION option on the action bar.
Shift+F8	FindLocalFunction	Locate the first occurrence of the a function name that you enter. The function can be local or global, and in any file.

F9	FormatVar	Change the current formatting of the variable/storage at the cursor. You can use this key to change a variable's display from an integer to a long, double, float, or other options. (Storage Window Only)
F10	ActionBar	Jump to the action bar. You can select action bar options with the left and right arrow keys followed by the Enter key or with the starting letter keys of the action bar options. Pressing Esc leaves the action bar without performing any action. The options are described in "Action Bar Options" on page 15.

Variables

Variables Known to the Debugger

The debugger can only display the variables it knows about. All variables, formal parameters, and labels declared within a module are local to that module. They are unknown or undefined outside of that module.

Refer to the Toshiba 4690 BASIC Language Reference under the heading "Scope of Data" or your favorite programming manual for more information on the scope of variables.

Formulas and expressions can have variable names, provided they are known to the debugger. The variables that are known to the debugger depends on the compiler used.

If you have BASIC Version 3 or later and cannot locate non-global variables and functions, check your compile options. Refer to "Compilation of 4690 BASIC programs for debugging" on page 7 for information.

NOTE: You can select variables to view or watch in the storage window by placing the cursor on their name and pressing the Insert, * or Ctrl+Ins keys. These functions work on non-executable and executable lines.

To view a variable, place the cursor on it and press the = key. This will display the variable on the bottom line of the display. If that name is a function, rather than a variable, then the debugger will show the source code for that function. After displaying that function, you can press <Alt>-F7 to return to the screen where you pressed =, or continue on with other debugging commands.

Partial Name Matching

When using Display Variable to insert a variable into the storage window, you do not have to type its entire name. You should specify a sufficient number of characters so that the variable you want can be uniquely selected. This feature is particularly useful if you have names that are very long, but have common prefixes or suffixes. It is also useful when you can't remember the exact, complete name.

If the debugger uniquely identifies just one variable with the string you typed, it processes that variable just as if you entered its entire name. If the debugger identifies two through 100 variables with that partial name, it presents you with a "Select a Symbol Name" window. You can press the PgDn, PgUp, DownArrow and UpArrow keys to find the desired variable, then press Enter to select it. An example of the Select a Symbol Name window is shown in the figure below. The string `refund` was typed as the variable to display.

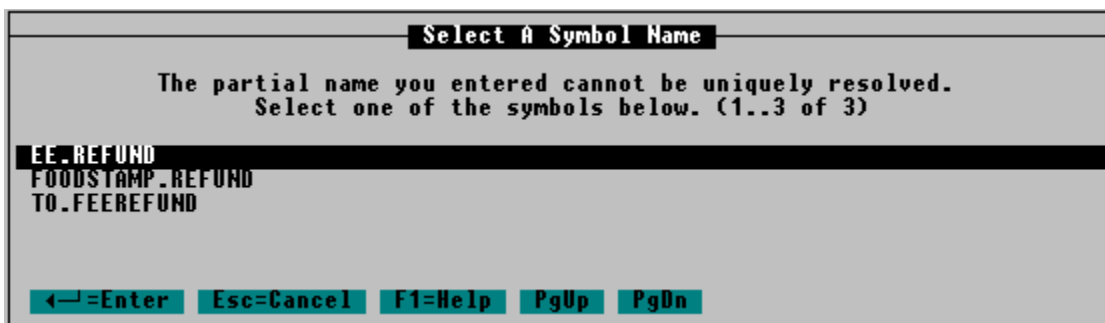


Figure 3. Select a Symbol Name

Recalling Previously Entered Names

In the Display Variable window, you can type the name of the variable that you want, or you can press the UpArrow and DownArrow keys to scroll through the list of variable names you last entered during the current debugger session. When you see the name you want, press Enter. If you see a similar name, you can use the LeftArrow or RightArrow keys to move to the character(s) you want to change and then type over the existing characters with the new name. The list of variable names you type is preserved across debugger sessions, using file DEBUG.KRB for intermediate storage between debugger sessions.

Formula Syntax

Formulas in the data window and expressions in other debugger contexts are composed of the following:

- Variable names known to the debugger
- The register names ax, bx, cx, dx, si, di, bp, sp, ds, es, ss, cs, ip
- Numeric constants, decimal or hexadecimal (see below)
- The plus (+) and minus (-) operators
- The multiply (*) operator
- The unary pointer de-reference (*) operator (only at the start of the formula)
- The address of (&) operator (only at the start of the formula)
- The pointer operator (->)
- The member operator (.) to display members of structures or unions
- The subscript operator [] or (). Subscripts must be constants; they may not be expressions or variables.

When the debugger checks a formula, variable or parameter name, it evaluates addresses. Register names evaluate to the contents of the named register. Constants evaluate to integers. A formula yields an address only if a segment is given (for example, ss:sp) or implied through use of a variable or parameter name. Formulas with no segment (for example, ax+1) yield scalar values.

Parentheses are not allowed in formulas and all operators have equal precedence. Expressions are evaluated from left to right except a pointer de-reference operator which is done last (for example, it applies to the address specified by the rest of the formula).

The previous formula interpretation differs from the way that the C language interprets expressions. For example, using the debugger, if array is an array of characters, array+2 refers to the second byte beyond array (as in C), and i+2. refers to the second byte beyond i (the C interpretation of i+2 is the value of i plus 2).

Although the area of the display devoted to the formula provides only a 10 character field, you may type expressions up to 80 characters long. When you press ENTER, the display field will be truncated to 10 characters, but the displayed value will still be correct. To see the full underlying formula, move the cursor to that line and press the left or right cursor key.

Numeric Constants in Debugger Expressions

Numbers are treated by the debugger as either decimal or hexadecimal in expressions parsed by the debugger. The rules are as follows:

1. A "0x" forces the number to be treated as hexadecimal. For example, the number "0x10" is hexadecimal 10 (decimal 16).
2. Any number appearing in an expression that contains a colon (":") is treated as hexadecimal. For example, in the following expression

47:5c+4

both 5c and 4 are treated as hex.

3. Any other number is treated as a decimal number. For example, the expression "alphabet[25]" treats the 25 as a decimal subscript.

How Variables are Displayed

The debugger displays several variable types in default formats. You can change the display format for each variable. See "[Data Display Format Options](#)" on page 24 for information. The default formats are:

<u>VARIABLE TYPE</u>	<u>DISPLAY FORMAT</u>
CHARACTER	Hex value followed by ASCII character in quotes.
INTEGER	Two-byte decimal.
LONG	Four-byte decimal. Four-
FLOAT	byte floating-point. Eight-
DOUBLE	byte floating-point.
POINTER	Selector:Offset => contents.
CHARACTER ARRAY	
INTEGER ARRAY	Hex value (16 bytes per line) followed by corresponding ASCII characters.
BASIC STRING	[Element number] followed by decimal contents - one element per line. Four byte pointer for early BASIC versions or Hex value(s) for the length of the string followed by the corresponding ASCII characters. For BASIC string arrays, if there are any characters in that element that are outside of the range of 0x20 .. 0x7F, the contents of that line will also be shown in hex if there is sufficient room on that line. About 18 characters can be shown in hex for single-dimension arrays.
BASIC REAL	A 4690 BASIC real number is represented by an 18 digit BCD floating point format.

Array Support

The 4690 Application Debugger supports the display and editing of arrays, provided you have all of the prerequisites.

To display the first few elements of an array, place your cursor on the array name (in the source window) and press INSert or the = key. The INSert key will place the array name in the storage window. The = key will show the first member of the array at the bottom of the screen.

To reduce clutter, array elements that contain a zero or null value are not shown in the storage window. You can switch to a mode whereby all values are shown, by pressing C Z. The first and last elements are always shown, regardless of the setting of the Zero/null toggle. Scalar variables are always shown, regardless of value. Figure 3B, below, illustrates suppression of zero-value array elements. TS.IO.DATA\$

has a value only in element (3). EMSS.DELAY.DATA has non-zero values in only four elements; the other 77 elements are zero, and 75 of those are not shown. EMSS.TG.CPN has non-zero values in five elements; the other 37 elements are zero, and 35 of those are not shown. All of the elements in TS.IO.KEYS are zero, therefore only the first and last are shown. If you display an array with a large number of zero-value elements, it can take some time to refresh the storage window. This is because of the overhead required to read each element in order to determine if it should be shown. An array with 1,000 elements could take several seconds to refresh, especially if you are running a Terminal application on a physically-different PC as compared to the controller that is running DEBUG.

```

Display Window Search Command
TS.IO.DATA$ (00) Invalid Address
              (03) "1"
              (10) Invalid Address
EMSS.DELAY.DATA
              (00, 00) 0
              (01, 00) 2001
              (02, 00) 2003
              (01, 01) 2
              (02, 01) -2
              (26, 02) 0
EMSS.TG.CPN (00, 00) 0
              (01, 00) 4203905
              (02, 00) 4203906
              (03, 00) 4203907
              (04, 00) 4203908
              (05, 00) 4203909
              (20, 01) 0
TS.IO.KEYS (00) 0 0x00
           (10) 0 0x00

```

Figure 3B. Suppression of zero-value array elements

To change a specific element of an array use the storage window. For example, assume that you have a BASIC array named XARRAY, which is a two dimensional array of strings (XARRAY (5,5)). It is 6 elements of 6 strings each, starting with (0,0). To modify the second string in the third array, go to the storage window and enter XARRAY(2,1) to display the string. To change the contents, tab to the data field, type over the data displayed, and press Enter. You can change the hex or the ASCII interpretation.

Displaying the contents of an array will always show the element of the array as it was type-defined in the program. If the program was 4690 BASIC and the array was not type-defined, the result is a BASIC real array. A BASIC Real number is binary coded decimal (BCD) floating point form. Each Real number occupies 10 byte of memory. The first byte is the sign and the exponent, the remaining nine bytes are the mantissa. Refer to the Toshiba 4690 BASIC Language Reference for more information on BASIC Real representation.

For the debugger to display a C language array, use square brackets. For example, the previously mentioned XARRAY element in C is displayed by the command XARRAY[3][2].

Data breakpoints are not directly supported for array elements.

Storage Window Keyboard Functions

The following general keyboard definitions are in effect in the storage window: (see "General Keyboard Functions" on page 17)

F1 (Help)
F3 (Quit)
UpCursor
TopOfWindow
DownCursor
BotOfWindow
PrevWindow
ShrinkStorageWindow
NextWindow
ExpandStorageWindow
FirstWindow
LastWindow

The following definitions are also in effect in the storage window.

Debugger Function	Key	Explanation
PrintGlobalVars	F8	For each Global variable, interpret it as if it were inserted into the storage window, and then print the interpretation to file DEBUGLOG.
FormatVar	F9	Show a menu of choices on the format of the data display. To use this function, move the cursor to the line containing the variable you wish to format and press F9. The choice of format affects only the current formula (for example, the one the cursor is within when you press the F9 key).
	ESC	Return to the source window.
	<CTRL> Enter	Insert a blank line at the location of the cursor.
	<CTRL>u	Move the variable at the cursor up by one line in the storage window, effectively swapping its position with the variable above.
	<CTRL>d	Move the variable at the cursor down by one line in the storage window, effectively swapping its position with the variable below..
	<CTRL>t	Move the variable at the cursor to the top line in the storage window. All variables that were above it are shifted down by one line.
	<CTRL>b	Move the variable at the cursor to the bottom line in the storage window. All variables that were below it are shifted up by one line.

Storage Window Display Screen

You can select variables to view or watch in the storage window by pressing the * (ShowVarPtsTo), Ins (PutVarInStg), or Ctrl+Ins (PutVarPtsToInStg) keys.

If you have a variable in the storage window and you use the Ins key to place a second variable in the window, the first variable will be truncated to one line. If you want to see more than one line of a variable displayed, it must be the last variable in the window. Depending on format, the last variable can be viewed on multiple pages.

Arrays can be displayed in the storage window. If you specify the array name without an index, and the storage window is empty, the debugger will display the entire array. By using an index, the element of the array can be modified. For more information on arrays refer to "[Array Support](#)" on page 21.

Printing Global Variables

In the storage window, you can press F8 at any time to interpret and dump all global variables to file DEBUGLOG. If the state of your application is adequately described by the content of Global variables, this feature can be very useful for comparing the state of your application at two different points in time.

All variables that are declared so as to be externally visible across modules will be interpreted. Each variable will be interpreted in the same fashion as if it were individually inserted into the storage window. There are three debugger settings that control the interpretation and printing of global variables.

- C N (toggle name lengths). Variables will be printed using one or two lines depending upon the current setting and the length of that variable's name.
- C Z (toggle zero/null array elements). If zero/null elements are currently being suppressed, only those array elements that are non-zero or non-null are interpreted and written to DEBUGLOG. Scalar variables and simple strings are not affected by this setting.
- the size of the storage window. <CTRL>-UpArrow or <CTRL>-DownArrow can be used to increase or decrease the size of the storage window to any number of lines between 1 and 20. Arrays that are interpreted and written to DEBUGLOG are limited to this number of lines (per array).

Because the interpretation of each variable or array element involves at least one storage access (three or more accesses for strings), this function can take several minutes to run if your application is being run on a physically different terminal as compared to the controller on which you are running the debugger, or if your application has many thousands of variables or array elements. You should set your expectations accordingly when invoking PrintGlobalVars under these conditions.

Data Display Format Options

Some variables cannot be displayed because of formatting problems (i.e. bad structures, pointers, memory has been freed,...).

The options for formatting the display of a variable are:

HEX	Displays storage as a sequence of hexadecimal bytes.
CHAR	Displays a variable as a single ASCII character (also in hex).
INT	Displays a variable as a decimal (2-byte) integer (also in hex).
LONG	Displays a variable as a decimal (4-byte) integer (also in hex).
FLOAT	Displays a variable as a floating point (4-byte) number.
DOUBLE	Displays a variable as a double-precision (8-byte) floating point number.
BASIC STRING	Displays a string for 4690 BASIC (for example, A\$).
BASIC REAL	A 4690 BASIC real number is represented by an 18 digit BCD floating point format.
(.NATIVE)	This option's wording depends on the type of formula the cursor is on. It shows the current type of the expression. If you select this option, the variable is reformatted according to the indicated type. An example is a C language structure or type.
?	Any type. Selecting this option will result in a prompt requesting a type name. The variable will be formatted as indicated by the type, which may be one of the following: <ul style="list-style-type: none">• char• int• long• uchar• uint• ulong• float

- double
- BASIC string
- BASIC real

An asterisk (*) may also be added after any of the above types to indicate a pointer to the type.

4690 BASIC Chaining

The debugger supports chaining from one executable program to another. While debugging a chain command, a window will display asking if you want to continue debugging the current application or switch to the new application that was just started. To perform a successful chain operation the files in both applications must be correctly compiled and linked.

For more information on chaining programs refer to the Toshiba 4690 BASIC Language Reference.

NOTE: The debugger can only maintain control of one application. If you select to debug the new (chained to) application, you lose debugger control of the old (chained from) application. All code and data breakpoints are cleared.

NOTE: The debugger can continue to chain to another application if specified by any of your chained applications, but loses capability to exit normally back to 4690 OS after the second chain has occurred. If you attempt to exit after two or more chains, you will be presented with popups that instruct you to kill the debugger window with <AltSys>-W, then <F8>.

Multiple DEBUG Sessions

The debugger can be run simultaneously in multiple sessions, but can only run one session in the terminal. Multiple controller applications or a controller and a terminal application may be run simultaneously. Each debugger session requires a separate command window.

It is recommended that each session be run from a separate directory. Use of files DEBUG.KBR, DEBUG.KBP and DEBUG.KRB is intended to be for only one debugger session. Separate directories for each session is compatible with this requirement.

Debugging Terminals in off-line mode

A simulated terminal off-line mode is available for debugging terminal programs. You must be debugging a terminal application to use this mode.

Use the fast path letters CTF or the debugger (C)ommand pull-down menu to set the off-line condition on the terminal, select the (T)erminal status option, and select the terminal o(F)f-line option on the terminal status menu. While in the off-line mode, the terminal cannot communicate with the controller except through debugger control.

Remember that debugger terminal off-line is a simulated condition and may cause unpredictable errors when a terminal returns to the on-line mode after an off-line debug session. This causes the terminal and controller to be out of sync. The files opened by the terminal may have been closed by the controller. No terminal or controller error recovery is available for these conditions.

Actions That Affect Multiple Breakpoints

Each of these features is capable of setting or clearing more than one breakpoint with a single action. You can use any of them to quickly affect multiple breakpoints by using that single key combination or sequence.

- Restarting your application (C A Y) will clear all existing code and data breakpoints.
- Chaining to another application will clear all existing code and data breakpoints.
- Command - run Without breakpoints (C W). This sequence temporarily deletes all breakpoints and issues a Run for your application. To regain control, press <CTRL>+BRK. After you regain control, all of your existing breakpoints will again be in effect. It is useful when you want your application to quiesce, but you do not want to deal with all of the breakpoint interrupts that you expect to occur before quiescence.
- Run to CBASIC function call (<CTRL>+F2). If you are debugging a CBASIC application, this keystroke will run your application until any CBASIC function is called. For more information, see [Monitoring CBASIC or C Function Calls](#) on page 35.
- Command - toggle func bPs this file (C P). This sequence sets (or clears) a code breakpoint at the beginning of each function in the currently-displayed file. It applies to both C and CBASIC programs. It is useful for watching the flow of control when you believe that a problem exists in the current source file, but you are not certain which function in the file is causing the problem. A message indicates how many breakpoints were set or cleared. You can use this sequence independently on any number of source files, and can independently clear or set any of the breakpoints that were set or cleared in this fashion. Breakpoints that are set with C P in a file remain in effect until they are cleared with <F4> or C P in that file.

Files that are Used by the Debugger

The debugger uses certain files for its own purposes. They are created or opened using the current directory where the debugger is started. Do not use these files for any other purpose.

DEBUG.KBR	This file is created. It contains one line for each keystroke you enter from the keyboard. You can copy this file to DEBUG.KBP in order to replay a session.
DEBUG.KBP	This file is read during the startup of the debugger. It allows you to establish a unique startup script. You can edit this file before restarting the debugger to alter the replay script. Only the first four characters of each line are significant.
DEBUG.KRB	This file is created when you exit the debugger, and is read when you start the debugger. It contains the contents of the keyboard recall buffer. You may safely delete this file but do not edit it.
DEBUG.SCR	When invocation parameter /o4 is used, this file contains the entire screen contents, including color and highlighting attributes, captured after every keystroke.
DEBUGLOG	Error and trace messages.
DEBUG\$\$\$CDV	When the debugger starts, your application.CDV file is copied to this file. This minimizes file access and data integrity problems if you update your CDV file while the debugger is running.

Keystroke Recording and Playback

Every character you type on the keyboard is unconditionally written to file DEBUG.KBR in the current directory of your 4690 system. This serves four purposes:

1. Inspection of a previous session.

You can edit, type or print DEBUG.KBR to review the commands and variable names you entered during a session. If the session was complex, you might have forgotten what steps you took to get to a particular point in your session, or might forget what function names you typed to locate, or what variable names you typed to display.

DEBUG.KBR contains one line of information for each keystroke of the DEBUG session. It consists of six fields: the four-digit hex code for the key; the symbolic name of the key (for most of the more common keys); the selector:offset of the current instruction when that key was pressed; the name of the current LCV file; the line number within the current LCV file; and the current source statement, if known, when that key was pressed. To save space, the current source statement is usually omitted when it is a duplicate of the previous line.

2. Replaying a session.

To replay a session, copy file DEBUG.KBR to file DEBUG.KBP and restart the debugger. All keystrokes you entered during the previous session will be played back just as if you typed all of those same keystrokes again in that new session. Playback is not artificially throttled, but occurs as fast as your computer, your application and the debugger will allow. If your replay script involves any interaction with an external entity, such as a terminal, then any keystrokes, scans, etc. on the terminal must occur exactly in sequence during the replay as they occurred during the initial session.

You can edit DEBUG.KBP before replaying a session, using XE, DREDIX or any ASCII text editor. Only columns 1 through 4 on each line are significant; additional information on each line is ignored during playback. An F3 keystroke (3D00 in columns 1 through 4) is not replayed, and signifies that the playback script should cease. When an F3 is read, or an EOF occurs on the playback file, processing of keystrokes resumes from the keyboard so that you can perform any additional debugging actions desired for that particular session.

An excellent use of Replay is to quickly set the same breakpoints in a new debugger session that you set in a previous session. This was the primary reason for implementation of keystroke recording and playback in the debugger. When debugging an application, a user often runs several sessions of the debugger, making minor code changes between sessions. Establishing a base session with useful breakpoints and then replaying that session provides a customized starting point for each follow-on session. Adding lines near breakpoints may cause replayed sessions to behave in undesirable ways.

Avoid use of the four arrow keys and the Insert key when establishing a base session that you intend to replay. Use Search Function, Search String and Display Variable, in order to minimize replay problems caused by changes in source code between sessions. The use of the uparrow and downarrow keys immediately prior to setting breakpoints or inserting variables into the storage window will cause the replay session to behave differently if you add or remove lines of code in those areas.

There is no limit to the size of the DEBUG.KBP file.

3. Documenting an application problem.

When handing off a problem with your application to another person, you can start a DEBUG session, insert appropriate variable(s) into the storage window and set appropriate breakpoint(s), then run to the point where the problem occurs and exit the debugger. Furnish file DEBUG.KBR to the person who will be fixing the problem with the application. These actions will minimize the documentation

you need to provide to the person who will be fixing the application problem, and ensures that you have accurately documented the debugger actions necessary to recreate that problem.

DEBUG.KBR also can be used to archive recreate steps for problems with your application. Copy DEBUG.KBR to a unique file name in the archive you use for past problems.

4. Code Path regression testing of your application.

Using carefully selected steps, it may be possible for you to create a replay session that inserts breakpoints and alters memory contents in order to exercise certain code paths in your application. You can then replay the session at a later date to ensure that your application performs or continues to perform as intended.

Keystroke recording and playback is limited to the keyboard used by the debugger program. You can't, for example, record or playback keystrokes that you typed into the 50-key keyboard attached to a 4690 terminal.

When replaying a session, note that the replay will occur as fast as possible and is intended to occur without interruption of any kind. If you want to cause it to pause at any point so that you can inspect variables, you need to design that pause into the replay session by requiring an external dependency (such as a terminal keyboard or scan action), and building in the Display Variable actions into the replay session for the variables you want to visually inspect.

Using Data Breakpoints

Beginning with Version 4.16 of the debugger, breakpoints can be set so that a trap occurs when selected, specific variables are changed.

To set a data breakpoint, move the cursor in the source window to any line and column that has the name of the data variable you want to monitor. Press CTRL+F4 or ALT+F4. A message at the bottom of the screen states that a data breakpoint is set, and gives the selector:offset of the breakpoint and the name of the variable that is so marked.

To set a data breakpoint (indirectly) based upon the current contents of a pointer, move the cursor to the source line and column that has the name of the pointer. Press SHIFT+F4. A message at the bottom of the screen states that a data breakpoint is set, and gives the selector:offset of the pointer and the selector:offset of the pointer contents (the actual breakpoint). This breakpoint is static and must be set and used carefully. If you subsequently change the value of the pointer, in the debugger or with your program, the breakpoint address remains at the old address that you originally set. You will need to clear and reset the breakpoint to establish the new breakpoint address. This feature is recommended only for special cases whereby the breakpoint address never changes.

To run to a data breakpoint, press F5 or F6. The data breakpoint is never automatically cleared, but remains set until you explicitly clear it.

Data breakpoints are cleared just like they are set. Move the cursor to any source window line and column that has the name of a data variable you previously set. Press CTRL+F4, ALT+F4 or SHIFT+F4. A message at the bottom of the screen states that the data breakpoint is cleared, and gives the selector:offset of the breakpoint.

You can set a data breakpoint on up to four storage locations. The size of the field beginning at that breakpoint address will be set to two bytes for all variables that are of type INT or UINT, or will be set to four bytes for all variables that are of type LONG, ULONG, or CBASIC String. Field lengths for all other types are set to one byte.

A Data Breakpoint is a trap rather than an exception. Memory is modified, and then the trap occurs. When the trap occurs, the CS:IP that is reported to the Debugger is the next Assembler instruction after the Assembler instruction that modified the data. The Debugger reports the breakpoint with the information it is given, i.e. this next address. It makes no special attempt to highlight the offending instruction. This might take some getting used to, especially when the offending instruction is the last statement in a complex if or else clause. To help you notice this, the name of the variable that caused the breakpoint is shown at the bottom of the screen in the 'Stopped at Data Breakpoint' message. In cases where a pointer data breakpoint traps, the pointer address and the trap address are both shown instead of the pointer variable name.

Data breakpoints are not supported for any machine that is 386-class or older. You need at least a 486 or Pentium.

To use Data Breakpoints, you need 4690 OS V3R2 or later, or a modified kernel from V3R1 (ADXCT8SL.286/BSX and ADXRT8GL.286/BSX). Previous levels of 4690 do not support Data Breakpoints. If you are running the debugger against a terminal application, remember to run terminal load-shrink (ADXRTCCL) and reload the terminal after you updated the new kernel on your system.

The following two figures illustrate setting and hitting a data breakpoint. In Figure 4, the cursor was placed on 'TS.IN.IPL' and ALT+F4 was pressed. A message at the bottom of the screen states that the breakpoint was set.

In Figure 5, F6 (Run) was pressed. A message at the bottom of the screen states that a data breakpoint was hit, and shows the name of the variable at that data breakpoint address. Notice that the next sequential instruction points to the statement after the statement that caused the breakpoint to occur.

```

TS.IN.IPL = -1           !
TS.MSRSEP$ = CHR$(13)+CHR$(61)  !
TS.MSR.ONKBD = -1      ! as
DEBUG=EAMTS1DL.CDU <DEBUG=EAMTS11C.LCU>
Data breakpoint set to 02DF:198E for <TS.IN.IPL>

```

Figure 4. Setting a Data Breakpoint

```

TS.IN.IPL = -1           !
TS.MSRSEP$ = CHR$(13)+CHR$(61)  !
TS.MSR.ONKBD = -1      ! as
DEBUG=EAMTS1DL.CDU <DEBUG=EAMTS11C.LCU>
Stopped at Data Breakpoint for <TS.IN.IPL> in <main>

```

Figure 5. Reporting of a Data Breakpoint

Some Data Breakpoint Limitations

Data breakpoints are based upon a specific selector:offset and a specific process ID. If, for example, you set a data breakpoint on 0067:13BA for TS.IN.IPL, that breakpoint will only trap at that selector:offset combination. If another part of your program addresses TS.IN.IPL as a different selector:offset, the breakpoint will not trap, even though your breakpoint variable changes. Furthermore, if a different process or the 4690 kernel changes TS.IN.IPL, the breakpoint also will not trap. These limitations are due to Intel architecture. Only assignment statements (typically, assembler MOVs) within your process will trap.

A Data Breakpoint trap occurs at every attempt to alter the data at a breakpoint address. Whether or not the new value is different from the old value is immaterial. For example, if TS.IN.IPL has a current value of 5, then the statement TS.IN.IPL = 5 will trap, even though the value did not actually change. You can't prevent the trap from occurring, but you can type D D from a source window and use the C Condition so that these unchanged values do not stop the debugger from continuing on at that point.

When setting a data breakpoint, the debugger establishes the breakpoint address as the address of the variable. It disregards any array indices that may or may not be specified on the statement you use to set the breakpoint.

In similar fashion, BASIC strings are organized as a pointer and a length. The debugger establishes the breakpoint on the address of the string descriptor, which is a pointer. Therefore, a data breakpoint trap would occur for a string only if the pointer to the string descriptor changes.

Data breakpoints are intended to be set on scalar variables, such as INTEGER*2 variables. It is inconvenient, for example to set a breakpoint on the second element of an INTEGER*4 array or a string array. To do so requires use of a pointer variable that is assigned to the value of the address you want to monitor, and then using Shift+F4 to set the breakpoint. Assignment of the address to the pointer variable can be made with an explicit assignment statement in your program, or it can be manually set using Insert, Window - Storage, and then typing in the value. Note that a re-DIM of an array may change it's address!

The use of a data breakpoint on a stack variable is not recommended, except under very carefully-controlled conditions. The breakpoint will remain set after the current function returns, unless you explicitly clear it. The next function call is likely to map a different variable on top of that address, which could lead to a breakpoint in an obscure part of your program. A better solution is to make a code change to assign a value to a GLOBAL, and then set a data breakpoint on that GLOBAL.

Data Breakpoint Reporting

A breakpoint trap occurs after every attempt to change any breakpoint variable. Every trap is reported to the debugger by the OS, and the debugger must process it. The debugger can either report that condition to you and stop, awaiting further action, or it can report the condition and then continue running.

The Display - Data breakpoints window allows you to specify the conditions which cause a stop. See Figure 5B for an example of this window.

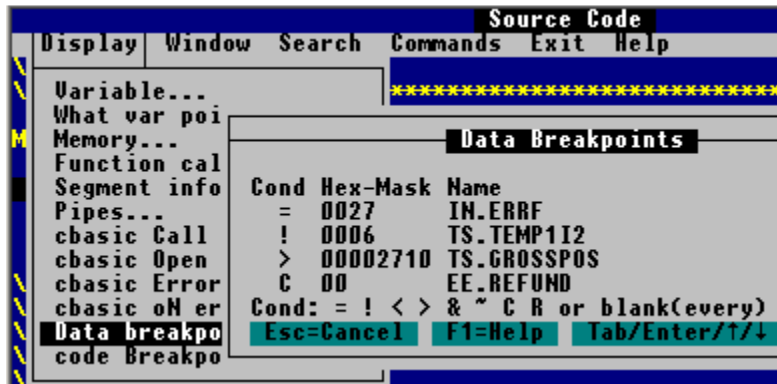


Figure 5B. Data Breakpoint Conditions

With the example shown in Figure 5B, a stop will occur if any of the following conditions are true:

- If IN.ERRF is set to hex 27 (decimal 39). If IN.ERRF is set to any value other than hex 27, the source/ assembler and data windows are updated, but the debugger continues running.
- If TS.TEMP1I2 is set to any value other than 6. If TS.TEMP1I2 is set to 6, the source/ assembler and data windows are updated, but the debugger continues running.
- If TS.GROSSPOS is set to any value greater than 10000 (hex 2710). If TS.GROSSPOS is set to any value from 0 through 10000, the source/ assembler and data windows are updated, but the debugger continues running.
- If EE.REFUND is set to any value that is different from its previous value. If EE.REFUND is modified, but the new value is the same as the old value, the source/ assembler and data windows are updated, but the debugger continues running.

If a breakpoint is reported and all of the specified conditions are false, the debugger continues on by internally issuing a <F6> Run. The storage window is updated before the <F6> occurs.

Cond operators and actions that you can specify with D D are as follows:

(blank)	Stop at every breakpoint trap.
C	Stop only when the variable actually changes in value from the last breakpoint.
=	Stop only when the new value is equal to Mask.
!	Stop only when the new value is not equal to Mask.
<	Stop only when the new (unsigned) value is less than Mask.
>	Stop only when the new (unsigned) value is greater than Mask.
&	Stop only when any Mask bits in the variable are on.
~	Stop only when any Mask bits in the variable are off.
R	Stop on every read or write access of that variable.

The R condition implies that a breakpoint trap will occur every time that variable is changed or read. Testing in an IF statement, or the right side of an assignment statement are two examples of a read access. If your program is very complex, you may wish to use this feature to determine where in your program that particular variable is tested or copied. Mask is immaterial with condition R.

Only one Cond operator can be specified for each variable. If you want a condition such as \leq , use $<$ and then adjust Mask accordingly. If the set of Cond operators shown above is not adequate to specify a complex stop condition, it is suggested that you change your source code to look for that condition in an 'if' statement, and then set a code breakpoint in the 'then' clause of that new 'if' statement.

The Mask value is always specified in hex, and is considered to be unsigned. If use of the sign is important, you might have to make a code change as outlined in the previous paragraph.

When a data breakpoint is first set for a variable, or if it is cleared and then re-set, the Cond operator is set to blank (stop at every breakpoint) and the Mask is set to zero.

While the D window is shown, press Enter or Tab to jump to the next Cond or Mask field.

Displaying the CBASIC Call History

The CBASIC Runtime libraries, SB286L.L86 (controller) and SB286LT.L86 (terminal) were changed to add a function call trace buffer in February 2003 with defect 8000920 . You must link your application with one of these modified L86 files in order to be able to view the Call History in the debugger. This buffer is named 'tracebuffer', its size is 'tracesize' and its current entry is 'traceindex'. You must not use these names as publics in your applications. For each subroutine or function call, an entry is added to this trace buffer, consisting of the CS:IP of the called function, the current BP, and a count that represents the call nesting level at that point. The nesting level is incremented for each CALL and decremented for each RET. The buffer always wraps when full. GOSUBs are not traced.

Do not confuse this feature with Display - Function call trace. The Function call trace shows an interpretation of the current stack, which only contains the functions that are currently active. The cbasic Call history shows a trace of the last 25 calls that occurred in your application, many of which could have already completed and returned to their calling function.

The debugger shows this trace when you type D C (Display - cbasic Call history) from a source or assembler window. An example is shown in figure 6 below.

Calls are shown from oldest to newest, with the bottom entry marked LAST. Calls are indented according to their nesting level, so that you can easily determine the parent function for each call.

In this example, the last function called is TSU23.FB. It was called by TSUPEC23. Previous to this call, TSUPEC23 called TSU23.CPN. The arrow next to "current" near the bottom of this window is pointing at TSUPEC23, which is the function that is currently executing.

#	address	Subroutine/Function calls (oldest first)	msec
32	01FF:009A	?DAY.OF.WEEK	29
33	0187:00E5	OPEN.DESPOOL.LOG	
34	00E7:18D7	TSUPEC27	
35	0227:003B	GAT5EC01	
36	01AF:345B	GAT1EC02	
37	0187:5B22	EETSEC38	
38	01FF:009A	?DAY.OF.WEEK	43
39	01F7:0C89	TSDSEC01	
40	01F7:0B40	RESTORE.GUI.SUBSTATE	
41	015F:0534	TSHIECDS	
42	00E7:17BD	TSUPEC23	
43	0437:9932	TSU23.CPN	
LAST	0457:7854	TSU23.FB	****

Figure 6. CBASIC Call History

You can use the PgUp, PgDn, UpArrow and DownArrow keys to show other entries in the CBASIC Call History window.

One entry in this window is highlighted. You can go directly to the display of the highlighted function by pressing Enter. To change the highlighted line, press the UpArrow, DownArrow, PgUp or PgDn keys. In the example shown above, Enter is equivalent to ESC D F ACCESS.RELOAD.FILE.

If a local function call cannot be resolved, it is marked with an offset from its nearest public function, or marked with a name of ?Unknown?. You cannot highlight and go to that function from this menu. One alternative is to write down the selector and offset, switch to the Assembler window, Search Function, and

type that ssss:oooo. Another alternative is to change and recompile your source code so that those functions are PUBLIC, changing the names if necessary so as to avoid clashing with existing PUBLIC names.

Functions with names that do not completely fit in the window cannot be selected from this window.

For comparison or additional analysis, you can print (to a file) all entries in this trace by pressing <F2>. Entries are printed to file DEBUGLOG. If you wish, you can start another command window and view or print this file while the debugger is still running.

The default number of entries in this trace buffer is 25. If the default of 25 is used, the debugger will increase this number to 400 during each debugger session, and will also turn on the timestamp trace (see below). If you wish to use a different number of entries, you can patch the number of entries to a value between 26 and about 8,000 by running the tool `settrace` against your postlinked application.286 file and then rerunning the debugger. The debugger will then use that number of entries. A sample use of `settrace` is:

```
SETTRACE T eamts101.286 1500
```

If you are running `settrace` on a controller application, specify C for the first parameter (in place of T).

Each time you relink your application, you must rerun `settrace` if you do not want the number of entries in the trace buffer to be 400.

When the first call occurs in your application, CBASIC allocates the trace buffer from free memory based upon the number of entries specified. Each entry is eight bytes in length, consisting of CS, IP, BP and the nesting level of that particular call.

Hotspot Analysis using Timestamps

Problems in your program are not always limited to incorrect function. Sometimes, slow but accurate performance is unacceptable and must be improved. It is often difficult to precisely predict where long run times or delays occur in a program. To assist with this analysis, the CBASIC runtimes were changed in October, 2003 to capture timestamps at the beginning of each function call. The method saves them in a new global buffer, 'timebuffer'. This feature is turned off, by default, to minimize run time in working stores.

If timestamps are captured, the debugger adds them at the end of each CBASIC Call History line. Timestamps are shown in milliseconds. The bottom entry in the trace has a value of '*****' because its ending time is not yet known. The value shown by the debugger on each line is determined by subtracting the relative timestamp value when the function on that line was called from the relative timestamp value when the function on the line below it was called, and then showing the difference. Values of less than one millisecond are not shown.

You can use these timestamps to analyze your program for relative hotspots. In Figure 6 above, ACCESS.RELOAD.FILE uses the largest amount of time, compared to all other functions in the current trace window. By comparison, TMCFR.INIT uses about a third as much time as ACCESS.RELOAD.FILE, and most other functions use relatively little time. NOTE: there will be variations in the time shown from one run to another. If a time value seems unrealistic, it is suggested that you rerun to determine if the time is repeatable. Another task may have pre-empted your application at that point and distorted the results.

If breakpoints are hit, or if you have operator intervention used with your debugging script, these long periods of time can appear to dominate the usage of time for the functions active at those times, and usually should be ignored. Processor time values over 9999 (ten seconds) are shown as '*****'.

The timestamp trace is turned off in the CBASIC runtimes by default, to minimize run time when you don't need to capture it. You can turn on the timestamp trace by running the tool `settime` against your

postlinked application.286 file before you run the debugger. `settime` and `settrace` can be run in either order. A sample use of `settime` is:

```
SETTIME eamts101.286
```

NOTE: if the debugger changes the number of trace entries to 400 from the default of 25, it will also unconditionally turn on the timestamp flag. If you do not want the timestamp trace turned on, then you must use `settrace` to change the number of trace entries to any number other than 25. When the number of trace entries is not 25, the debugger will not change that value, nor will it alter the timestamp flag.

For best results, eliminate or minimize concurrent operations on your computer while collecting processor timestamp values. Functions that are waiting for an I/O operation, or are ready to run but are delayed by other running tasks on a heavily-loaded system will show that they are accumulating time, just as if they were actually consuming processor cycles.

Be careful when analyzing hotspots, even when you are running on a system that has no other running tasks. The time shown for a function is not necessarily consumed totally within that function, but rather is consumed from the time that that function was called until the next function call occurs. This time could be spent in the parent function between calls, or could even be in the parent's parent if the parent returned between calls. See `OPEN.FOR.SL.STR` in Figure 6. The time shown for it could have been consumed in it, but could also have been consumed in its parent, `UPDATE.SYNC.FILE.RECOVERY`, or in the parent of `UPDATE.SYNC.FILE.RECOVERY` (not shown on this screen). The debugger has no way to determine which of those three functions consumed the time. Some time could also be consumed in an `ON ERROR` routine or in a `GOSUB`. If you require a more precise timestamp history, you probably will have to modify your source code. One way is to create one or more new functions that just return to their caller. Then insert statements to call those functions in places where you wish to establish timing checkpoints. Excellent places to insert these new calls are right before each `RETURN`, and right before and after each block of code that you think might be causing the timing problem. Use of unique names for each function called can help you to more quickly locate where your hotspots are occurring.

Monitoring CBASIC or C Function Calls

With large or complex applications, it may be difficult for you to understand or remember the flow of control with regard to the order of function calls.

CBASIC programmers can use the run to CBASIC function call feature to assist with this task.

In a source or assembler window, press `<CTRL>F2`. The debugger will free-run from the current `CS:IP` until any CBASIC function is called, stopping at the first executable statement in that function. You can then note the name of the function at that point and explore its purpose by viewing code, single-stepping or setting breakpoints. If that function is of no interest to you, you can continue to press `<CTRL>F2` until the desired function is hit, or resume other debugging actions at any time.

The `<CTRL>F2` action remains in effect until the next CBASIC function is called, and then it is canceled. Intervening events, such as code or data breakpoints, or `<CTRL>BRK` does not cancel it. When any of these intervening events occurs, the next CBASIC function call will trigger a breakpoint, regardless of the key used to continue on from that intermediate point.

Only genuine CBASIC function and subroutine calls are traced in this manner. `GOSUBs` are not traced.

CBASIC or C programmers can use the `Command - toggle func bPs this file (C P)` action for a slightly different approach to monitoring function calls. It sets or clears breakpoints at the beginning of all functions in only the currently-displayed source file. You can repeat this action for other source files to add additional breakpoints if you wish. All of these breakpoints remain set, on a per-file basis, until they are cleared in the same fashion for each file, or they are individually cleared by pressing `<F4>` on each breakpoint line.

Displaying the list of CBASIC Open Files

The debugger can find and display the session numbers and the names of the files that are associated with each open session. From the Source or Assembler window, type D O at any time. The list of open files is displayed. If you have more than 13 open sessions, you can use the PgUp, PgDn and arrow keys to show other entries in the list. The example in figure 7 below, shows that session number 3 is open against file EAMTERMS on the controller. Sessions 1-2, 5, 7-10 and others are not open. In this example, sessions 36 and above can be viewed by pressing PgDn or DownArrow. The debugger shows only sessions 0 through 99.



```
Source Code
Display Window Search Commands Exit Help

Variable...
What var points to
Memory...
Function call trac
Segment informatio
Pipes...
cbasic Call histor
! cbasic Open files
! cbasic Error trace
cbasic oN error ro
Data breakpoints
C code Breakpoints

IRRF_LOCAL_FILES$ =
TS.SIGNED_ON = 0
TS.GROSSPOS = 0
TS.GROSSNEG = 0

TS.IN.IPL = -1
TS.MSRSEP$ = CHR$(1

- \EAMT10L.CDU < - \EAMT11C.LCU>

CBASIC Open Files

Session  FileName or Define
3 <R::EAMTERMS>
4 <R::$AMITEMR>
6 <R::EAMEEDS3>
11 <R::$AMTENDU>
20 <R::EAMSDDESC>
21 <R::EAMEESL3>
24 <R::EAMX:001>
27 <R::EAMTRANA>
30 <ADXPIA>
32 <ADXPII>
33 <ADXPIK>
34 <ADXPPIPR>
35 <ADXPPIPD>

Esc=Cancel  F1=Help  ↑  ↓  |
```

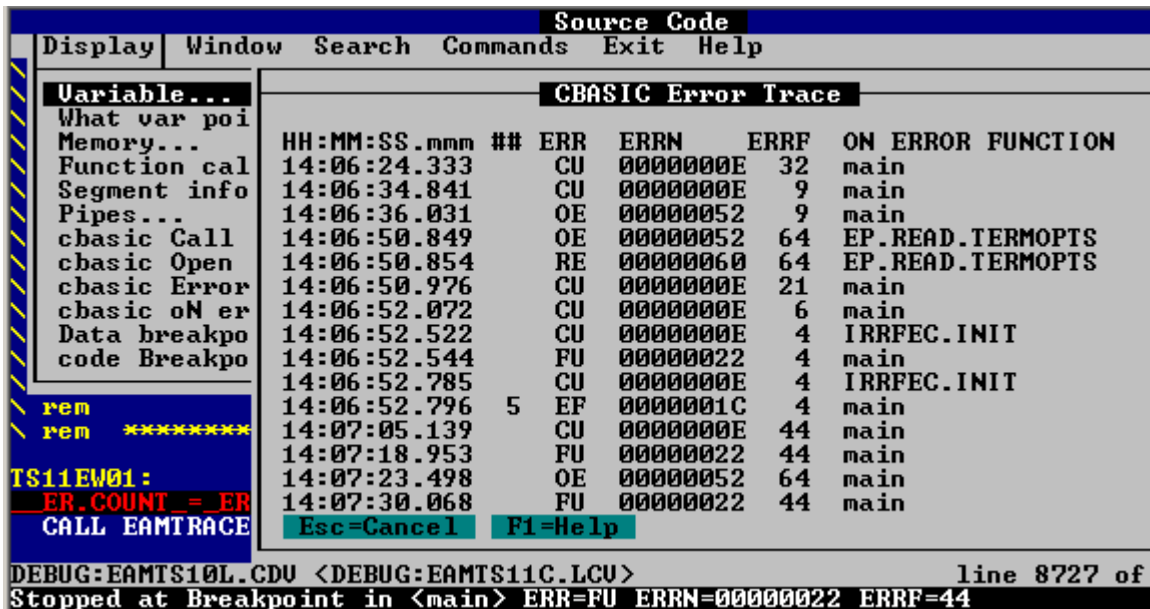
Figure 7. CBASIC Open Files

Displaying the CBASIC Error Trace

The CBASIC runtimes maintain an internal error trace buffer. It contains fifteen entries, and wraps when full. Each entry contains the time when the error occurs, the two-letter ERR code, the four-byte ERRN code, the session number (ERRF), and optionally the address of the ON ERROR routine at the time each error occurred. When finished, press ESC to return to the Source or Assembler window. An example is shown in figure 8 below.

In November of 2004, CBASIC runtime defect 1796 altered the timestamp from hours and minutes, to hours, minutes, seconds and milliseconds. It also suppresses consecutive, duplicate trace entries. The '##' count is the number of suppressed entries. The time stamp shown for a set of duplicate entries is the time associated with the first entry in that set. The debugger supports either trace format. You must obtain and link in the updated SB286L.L86 or SB286LT.L86 in order to see this new trace format in the debugger.

If you use the Terminal Big memory model SB286LT.L86, and it contains defect 2096 (June, 2005), CBASIC traces the address of the current ON ERROR routine each time an error occurs. The debugger shows the first 16 bytes of the name of the enclosing function for each of those ON ERROR routines. If you are using a version of SB286LT.L86 prior to this defect, the debugger leaves this column blank.



HH:MM:SS.mmm	##	ERR	ERRN	ERRF	ON ERROR FUNCTION
14:06:24.333		CU	0000000E	32	main
14:06:34.841		CU	0000000E	9	main
14:06:36.031		OE	00000052	9	main
14:06:50.849		OE	00000052	64	EP.READ.TERMOPTS
14:06:50.854		RE	00000060	64	EP.READ.TERMOPTS
14:06:50.976		CU	0000000E	21	main
14:06:52.072		CU	0000000E	6	main
14:06:52.522		CU	0000000E	4	IRRFEC.INIT
14:06:52.544		FU	00000022	4	main
14:06:52.785		CU	0000000E	4	IRRFEC.INIT
14:06:52.796	5	EF	0000001C	4	main
14:07:05.139		CU	0000000E	44	main
14:07:18.953		FU	00000022	44	main
14:07:23.498		OE	00000052	64	main
14:07:30.068		FU	00000022	44	main

Figure 8. CBASIC Error Trace

Displaying the CBASIC ON ERROR Routine

The CBASIC runtimes maintain a global variable oerr which contains a pointer to the current ON ERROR routine. You can go directly to the display of the source code of that error catcher by pressing DN from any source window. Do not use global variable oerr for any other purpose in your program.

Interpreting CBASIC Error Codes

When a runtime error occurs, CBASIC provides functions ERR and ERRN to allow you to determine what type of error has occurred. You can lookup these codes in the Toshiba 4680 Basic Language Reference to help you understand the reason for the error and some possible ways in which you can recover from those errors.

The debugger provides a way for you to quickly interpret these errors. Whenever ERR and ERRN are set (typically in your ON ERROR routine), you can press H C to get a popup window that describes the reason for the error. An example is shown in Figure 9.

The debugger scans file DEBUG:CBASICRC.DAT for a match with ERR AND ERRN, and displays up to four lines of description text for that combination. Within this file, a definition of 'variable' is considered to be a wildcard for the comparison with ERRN. You can update this file with additional or modified ERR/ERRN descriptions if you wish. The second through fourth display lines begin in columns 76, 151 and 226. Columns 75, 150 and 225 must be blank. NOTE: if you reinstall the debugger, the default CBASICRC.DAT will overlay your changes.

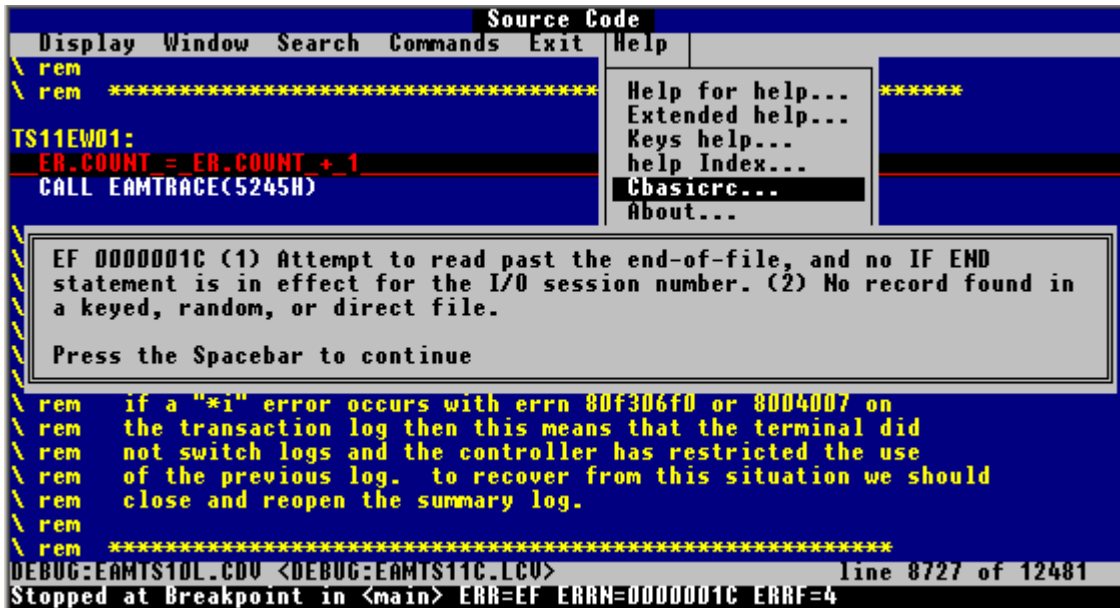


Figure 9. Help for CBASIC Return Codes

Appendix A. Source Window Keys

The following is a quick reference chart for the source window. See "General Keyboard Functions" on page 17 for a detailed explanation.

Key	Function
F1	Displays context-sensitive help message.
F2	Single step the target program.
Ctrl+F2	Run to the next CBASIC function call.
F3	Exit the program and debugger.
F4	Set or reset a breakpoint.
Ctrl+F4 or Alt+F4	Set or reset a data breakpoint.
Shift+F4	Set or reset a data pointer breakpoint.
F5	Run program to the cursor or breakpoint.
F6	Run the target program.
Ctrl+F6	Set the CS:IP to point to the source line at the cursor.
F7	Place the cursor at the instruction to be executed. Show current function name.
Alt+F7	Return to displaying the source window before the last Run (F2, F5 or F6), or the last Search - Function/String.
Ctrl+F7	Display the calling statement for the current function.
F8	Locate the function name at the cursor.
Shift+F8	Locate a local or global function of the name you enter.
F10	Jump to the action bar.
Ctrl+Break	Stop running program.
Spacebar	Single step the target program.
Plus(+)	Show the variable window.
Equal(=)	Display the variable at the cursor.
Minus(-)	Hide the variable window.
*	Display the data pointed to by the variable at the cursor.
&	Display all addresses of the name of the variable at the cursor.
Ins	Place the variable at the cursor into the storage window.
Up Arrow	Move the cursor up one line.
Down Arrow	Move the cursor down one line.
Right Arrow	Move the cursor right one word.
Tab	Move the cursor right one word.
Backspace	Move the cursor left one word.
Left Arrow	Move the cursor left one word.
SHIFT+TAB	Move the cursor left one word.
PgUp	Scroll to the previous window.
PgDn	Scroll to the next window.
Home	Move the cursor to the first line.
Ctrl+HOME	Scroll to the first window.
End	Move the cursor to the bottom.
Ctrl+END	Scroll to the last window in the current view.
Ctrl+ALT+TAB	Show the target program's screen.
@	Show the CS:IP and function name of the line at the cursor.

Appendix B. Assembler Window Keys

The following definitions are in effect in the Assembler window: See "General Keyboard Functions" on page 17 for a detailed explanation.

Key	Function
F1	Displays context-sensitive help message.
F2	Single step the target program (trace) with code breakpoints.
Ctrl+F2	Run to the next CBASIC function call.
F3	End the debug session and the program being debugged.
F4	Set or reset a program execution breakpoint on the cursor line.
F5	Run program to the cursor or breakpoint.
F6	Run the target program.
F7	Switch to the file containing the statement or instruction that is about to be executed and place the cursor on that line. Show current function name.
Alt+F7	Return to displaying the source window before the last Run (F2, F5 or F6), or the last Search - Function/String.
Ctrl+F7	Display the calling statement for the current function.
F8	Locate the function name at the cursor.
F10	Jump to the action bar.
Ctrl+Break	Stop running program or application being debugged.
Spacebar	Single step the target program (no trace) with code breakpoints enabled.
Plus(+)	Show the variable window with the selected variable values.
Minus(-)	Remove the variable window.
Up Arrow	Move the cursor up one line.
Down Arrow	Move the cursor down one line.
PgUp	Scroll to the previous window in the current view.
PgDn	Scroll to the next window in the current view.
Home	Move the cursor to the first line of the current window.
Ctrl+ALT+TAB	Toggle to/from the target application's screen.
End	Move the cursor to the last line of the current window.

Appendix C. Storage Window Keys

The following is a list of the keys that are available in the storage window. See "General Keyboard Functions" on page 17 for a detailed explanation.

Key	Function
Up Arrow	Move the cursor up one line.
Down Arrow	Move the cursor down one line.
Right Arrow	Edit storage using the formula; move the cursor.
Left Arrow	Move the cursor.
Backspace	Move the cursor (dragging).
Tab	Edit storage using the formula.
PgUp	Show the previous screen in the current file.
PgDn	Show the next screen in the current file.
Ctrl+Home	Display the first screen in the current file.
Home	Move the cursor to the first line on the current screen.
End	Move the cursor to the last line on the current screen.
Ctrl+UP	Shrink the storage window by one line.
Ctrl+DOWN	Expand the storage window to show one more line.
Ctrl+Enter	Add a blank line to the storage window.
Ctrl+Delete	Delete line from the storage window.
Ins	Toggle insert mode when editing storage.
Delete	Delete characters in the formula or storage display area.
Spacebar	Type over characters in the formula.
F1	Display help.
F8	Print all global variables to file DEBUGLOG.
F9	Change the formatting of the variable at the cursor.
Ctrl+u	Move the line at the cursor up by one line, swapping it with the line above it.
Ctrl+d	Move the line at the cursor down by one line, swapping it with the line below it.
Ctrl+t	Move the line at the cursor to the top of the storage window. All lines above it are moved down by one line.
Ctrl+b	Move the line at the cursor to the bottom of the storage window. All lines below it are moved up by one line.

Appendix D. Helpful Hints

The following sections contain helpful hints, tips, and information pertaining to debugger usage. "Appendix E. Problem Determination" on page 46, contains problem determination hints and limitations.

Finding Your Source and other Files

Source Files

The debugger searches in the following order for the source files that match the target 286 file:

1. The path specified when invoking DEBUG, for example,

```
DEBUG c:$test$myprog
```

2. The current directory
3. The environment variable (DEBUGSRC)
4. The directories in the PATH environment variable.

Use the DEBUGSRC environment variable for source files not in the same directory as the resulting object files or if the directory is not in the path environment variable. Use the following example to set the DEBUGSRC environment variable:

```
DEFINE DEBUGSRC=C:$MYPROG$SRC
```

Executable Files

The debugger looks in the following places (in order) for the source files that match the text contained in the target 286 file:

1. The path specified when invoking DEBUG, for example,

```
DEBUG c:$test$myprog
```

2. The current directory
3. The directories specified in the DEBUGSRC environment variable
4. The directories in the PATH environment variable.

Finding and Displaying Source Files

Search - Function typically is used to locate and display the source code of any public function. There may be times when you know the name of the file you would like to display, but can't remember the name of any public function in it. DEBUGCDV adds a filename.OBJ public function symbol for every OBJ file included in the CDV file. This feature allows you to switch to any file based upon the name of the file. As with other public functions, you don't need to enter the entire name, just the portion of it that will uniquely identify it from other functions and file names. If more than one file name or function name matches the partial name you enter, then the Select a Symbol Name popup will allow you to select one of the names from a menu. Supermarket Application example: if you type in s f tsfb, then a popup will allow you to select from EAMTSFBX.OBJ, EAMTSFBY.OBJ and EAMTSFBZ.OBJ. One of the public functions in that file will be displayed (typically, the function that is alphabetically first).

Function Call Trace

To display the program call stack (the Function Call Trace function), select the display pull-down and the function call trace option. This displays a list of active function calls that show the route taken by the program to get to the current place. The debugger assumes a standard far call (four-byte) using prologue

and epilogues. Each function executes the standard entry and exit code at the top and bottom of the function. Sample code for a function without arguments follows:

```
Top:
    push    bp
    mov     bp, sp

Bottom:
    mov     sp, bp
    pop     bp
    ret
```

NOTE: If your function does not contain any automatic data, (data that resides on the stack) then there may be no stack frame generated for that function. In that case, the debugger cannot include that function in its call stack display.

Single Step Delay

If you single step across a source line that compiles into a lengthy set of machine instructions, it may take a long time for that line to execute. The debugger may appear to be stopped or halted. However, it is not stopped, it is single stepping many instructions.

If you single step across a source statement that causes an I/O operation, it must complete this operation prior to the debugger regaining control. If the instruction requires input from the operator, you need to switch to the application screen and supply that input.

If the application appears locked, press Ctrl+Break to stop execution. If the application is not waiting for an I/O or an OS routine, the debugger will regain control within a second or two.

Single Step Problems

If you are using "IF END#" or "ON ERROR GOTO" BASIC statements, you may have problems with single step. These statements transfer control of the system without returning. This sometimes causes the Debugger to appear hung during a single step operation, because it always expects CALL statements to eventually return to the instruction after the CALL. To stop after these statements, place a break point after the label to which control may be transferred in your ON ERROR routines. You can quickly locate your current ON ERROR routine by pressing D N from a source window.

You will lose control if you single step over a RESUME statement. The debugger will run to the next breakpoint, because a RESUME never returns.

Problems with Single Step can also be caused by SWI's if you have a 4690 version prior to V3R2, or do not have the 4690 V3R1 kernel fix for this problem. Symptoms of the problem are usually a PV in your application due to damage in the stack of the application you are debugging.

Missing or Mismatched Source Files

If you are switched to an ASM screen when hitting a breakpoint or stepping into a function, the source file (LCV, BAS, C) may be missing. Try copying that file to your DEBUG directory and then pressing Window - sOource. If source code is not shown, your CDV file is probably missing line numbers for that file.

If you see blue on white lines when paging through a source file, hitting a breakpoint or stepping into a function, the version of that source file (LCV, BAS, C) may be incorrect. Try copying the correct version of that file to your DEBUG directory and then pressing any combination of PgUp, PgDn, Ctrl-Home or Ctrl-End until you see that the condition is corrected. The debugger retains about 1000 - 2000 lines of source code in its in-memory buffer, and use of these keys will usually refresh the buffer from the source file on disk. If this still does not show the correct source code, exit the debugger, rebuild that LCV file and

your application.CDV files, ensure that the newly built versions of both files are in your debugger directory, and then restart the debugger.

Where is a Specific Variable Used?

With complex applications, global variables are often used to save information about the state of the application, or to communicate arguments or results between functions in other source files. If a variable is only used by one or two source files, finding their references is usually of little concern. However, as the application becomes more complex or the use of a specific variable becomes widespread, this question becomes more difficult to answer.

The debugger can assist you in three ways:

- Display the file you think uses that variable and perform a Search - String. Use Next until the desired occurrence is found.
- Set a Data Breakpoint on the desired variable and then use D D to assign a cond of R. As your application is running, each time it references that variable (read or write), a breakpoint will occur.
- The Search - Variable feature shows you the names of all of the source files that use that variable. You can then select any of those files for display in the Source window, and repeatedly use Next to show each use of that variable in that source file.

Finding Non-Public Functions

With previous versions of the debugger, Search - Function can be used to find functions that are exposed with the Public/Global attribute. If a particular function is not marked PUBLIC, you can't search and find it this way.

You can now use <SHIFT>+F8 to perform a Local+Public search. Enter the name of the desired function, or a sufficient portion of the name of it so as to uniquely identify it. All files will be searched for local or public functions that contain that (partial) name. If a function is found with that exact name, or if the name you entered is found as a portion of only one function, that function will be displayed. If the name you entered is found as a portion of more than one function, then the Select A Symbol screen will allow you to select the desired function name from a list.

If two or more functions with the same name are identified, only the first occurrence of that name can be found this way. If you desire a different instance of that name, you must determine which file contains that instance, switch to that file, and then Search - String until the desired instance is found.

Scoping Problems with Variables

If the contents of a variable seems to be wrong, the debugger may be displaying an instance of that variable other than what you want. It is permissible, though confusing, to use the same variable name as a global, local and stack variable, even in the same source file. Each occupies a different address and are isolated from each other, just as if they had different names. In large applications such as SA or GSA, this sometimes occurs. The compiler is fully aware of the correct scope, but the debugger could choose the wrong one to display.

If you suspect that the debugger is choosing the wrong variable to display, move the cursor to the name of the variable in the source window and press &. The debugger will tell you how many different addresses are used for that variable name, throughout your entire application. If more than 1 instance occurs, a popup will show you all instances in groups of five at a time. Press Spacebar to see the next group. You can type any of those addresses into the storage window to see the contents at that address.

Another possible use of this feature is to determine if a variable is incorrectly scoped. If you need to have a variable shared between two or more files, then there should probably be one address for that name. If there are two or more, then one of them is not being shared. Conversely, if you expect to damage a variable that you know is Global, then you probably will want at least two occurrences of that variable.

Tips with Examples

See file DEBUGGER.TIP on installation diskette #2, or folder DISK2 on the CD-ROM disc, for additional tips and some examples to help you become more productive in using the debugger.

Appendix E. Problem Determination

Several problems are caused by incorrect debugger installation. The debugger install program creates debugger logical names. Therefore, the debugger may not operate correctly if the install program was not used. Refer to "Installation" on page 6 for information.

Assembler Screen Displayed

If you invoke the debugger on your program and see assembler code instead of your source code, there are several possible causes:

1. You did not compile and link your program with the correct options. This causes the debugger to use incorrect information.
2. The debugger cannot find the source file or .LCV file. If using 4690 BASIC, look for the .LCV file. The .LCV file is created by the DEBUGFRM program.

NOTE: The lower left corner of the debugger displays the source code file name it tried to find. You can easily recover from this condition by copying the needed file to your DEBUG directory and then pressing Window - sOurce.

3. You have renamed a source file.

NOTE: Check the lower left corner of the debugger for the name of the source code the debugger is trying to use. You can easily recover from this condition by renaming the file correctly and then pressing Window - sOurce.

4. You need to run the version of DEBUGCDV that has a fix for ASM modules. Versions of DEBUGCDV prior to March, 2002 discard symbols for all source modules that are included after any ASM module in your linker input list. You can bypass this problem by placing all ASM modules last in your linker input list, however use of the latest DEBUGCDV is recommended.

Unable to start Session error

This problem is usually caused by a back level of 4690 OS installed. Try rebooting your controller and reloading affected terminals.

Insufficient memory available will cause similar indications. Make sure that enough memory is available to the debugger. With modern systems, this error is unlikely.

Another cause of this error is trying to run a background or terminal application and not specifying the complete path name of the program when starting the debugger. An example is using `DEBUG /B BACKGRND` instead of `DEBUG /B SYSTEM:BACKGRND`.

For return codes that are a negative number such as FFFFFBC7, you may be able to obtain some help by referring to the section describing ADXSERVE in the 4690 Programming Guide.

Unable to Display Variable in the Storage Window

To display or scope a local variable with the debugger, display the source code that uses the variable, place the cursor on the name of the variable, and press the INSert key. The debugger will attempt to locate the variable. If the variable cannot be found by the debugger, an error will show on the message line.

If you are having problems displaying variables by entering them in storage window, check for proper scope of data. The debugger can only display variables within the scope of the executing program. Local

variables and parameters are only known within the routine where they are declared. These variables only exist while the routine is active.

Global variables may be local to the source module or may be known in all modules. This depends on the way they were declared. If the debugger cannot find the definition for the desired variable in the 'type' section of the currently-displayed file, it will search the 'type' section of all files for the definition of any variable at that address. If a match still does not occur, then it will display 16 bytes of hex data beginning at that variable's address.

The rules for the declaration and scope of variables is beyond the range of this manual. Refer to the Toshiba 4690 BASIC Language Reference under the heading "Scope of Data" or your favorite programming manual for more information on the definition and scope of variables.

Unable to run Debugger in the Terminal

The 4690 Application Debugger does not support some terminals, such as the 4683 Model 2 terminal.

Unable to start Help Session error

The help session error is generally caused by the HELP: logical name incorrect or not defined. Check the controller user logical names configuration. If you did not install the debugger using INSTALL.BAT from the first debugger installation diskette or the CD-ROM disc, do so now. HELP: should usually point to C:\DEBUG\.

Another cause of this error is the HELP.286 file has been erased from the SYSTEM: (ADX_SPGM) subdirectory. HELP.286 is the program that supports the help function for the debugger. The .HLP files are the text files that contain the information for the help screens.

Executable Lines incorrectly displayed

For continuation lines or multi-line statements, differences in color are normal.

If single-line statements are incorrectly displayed, ensure that you ran DEBUGFRM on the source code and that you did not use a %NOLIST or %DEBUG -V in the source code.

If the lines appear as blue on white, it signifies that the debugger believes there is a mismatch between the CDV file for your application and the current LCV file that is being displayed. Ensure that the LCV file in your debug directory is current. If necessary, recompile and rebuild the LCV file and CDV file.

Unable to see the correct Application Screen

The 4690 Application Debugger uses a virtual console. Many programs also use a virtual console. The debugger will switch to the first available virtual console. This may not be the application screen.

If you have a FUNCTIONKEYS ON in your program, another virtual console is created. This is usually the first console found by the debugger. To work around this condition, set FUNCTIONKEYS OFF in your program until debugging is complete.

String Variable is displayed as a Long Integer

The string variable must be reformatted. Use the F9 key on the storage screen. This is a normal for releases of 4690 BASIC prior to Version 3, and for strings that are out of scope.

Application Abends after using Ctrl-F6 to set CS:IP

Changing the CS:IP during program execution can be a very useful feature in cases where you want to skip execution of a statement, or change variables and then re-execute a statement. *It must be used with caution.* Note that the debugger only changes the CS:IP. It does not change any other registers, does not alter any memory or stack contents, does not adjust the stack pointer or stack segment, and does not change the condition code. Your failure to consider these possible side effects can cause your program to run incorrectly or abend if memory or registers are inappropriately setup for the new CS:IP. These effects can appear or be worsened if the prior breakpoint was not at the beginning of a source statement (i.e, after a data breakpoint in the middle of a source statement).

It is recommended that you use this feature only to execute a different statement in the same function. Do not use this feature to alter the CS or IP from one function to another, unless you know what you are doing and are prepared to take the consequences. A subsequent CALL, RETURN, or access of a stack variable will fail if the stack was inappropriately setup when you switched to the new CS:IP.

If your application becomes unstable after using this feature, restart it and review Assembler code in the areas involved. If you are unfamiliar with Assembler, you should seek guidance from someone in your organization who has such experience before continuing to use this feature.

Known Problems and Solutions

1. Debugger cannot display an array of structures

An array of structures is not supported by the debugger display routines.

2. Debugger does not display certain variables.

Some variables cannot be displayed by the debugger because of formatting problems (i.e. bad structures, pointers, memory has been freed,...).

3. Dump on Ctrl+Break, or Ctrl+Break does not work:

4690 V3R2 or a kernel with fixes on V3R1 is required. Ctrl+Break does not work at prior OS levels.

4. Out of Memory:

The application code must be post linked. This is especially true with terminal code. If you receive a return code of C040400D, the terminal that you are loading does not have sufficient memory for the debugger and your application. Try reducing the number of applications to correct out of memory problems. In the terminal try reducing the size of the ram disk.

5. Consoles:

Auxiliary console is not supported. If you require use of the debugger on an auxiliary console, see DEBUGGER.TIP on the second installation diskette, or folder DISK2 on the CD-ROM disc, for a partial-function workaround. Note that even with this workaround, use in that environment is not supported.

6. Code Breakpoints:

Some instructions will look like they accept a breakpoint, but since no code exists for that line of source code, the program will not stop at the breakpoint. Some examples are the WEND, FOR or NEXT. Set your breakpoints at the previous or following instruction.

7. Data Breakpoints do not break:

4690 OS V3R2 or later is required. CBASIC String breakpoints will only trap if the string pointer changes.

See directory DEBUGCK on debugger installation diskette #2, or folder DISK2 on the CD-ROM disc, for a sample program to test data breakpoint capability on your 4690 system.

8. Function Call Trace:

Some of the compilers will generate near calls for local functions. This creates a problem for the debugger while doing a Function Call Trace. If you are having problems with this operation, try making your functions external and recompiling.

9. ASCII interpretation:

When using the storage window to display arrays or structures, an ASCII interpretation may not be available on the right side. This occurs if the element of the array or field name extends into the hex display area. To see the ASCII interpretation display the specific element or specific field.

10. A PV occurs when stepping into a function, or when a breakpoint is hit on a FUNCTION or SUB statement:

This was a bug in the 4690 OS kernel. You need V3R2 or a modified kernel from V3R1. Fixes are not available for prior levels.

11. ASM code is shown when you step into a function or hit a data breakpoint:

The LCV version of the file being displayed does not exist or the line number option was not set when that file was compiled. If the LCV file exists, run DBINFO against your application.CDV file to determine if the current source file contains line numbers. If your CDV file contains line numbers, you can easily recover from this condition by copying the needed LCV file to your DEBUG directory and then pressing Window - sOurce.

12. Your application behaves in unusual fashion when run from the debugger, but behaves normally when run from the command line or when loaded normally into the terminal:

Ensure that all of your application files (*.286, *.CDV, *.LCV) are at the same level of code. If you are running your application from a different controller, both controllers must have the same set of application files. Mismatches can occur in many ways:

- when you change a source file, recompile, relink and run DEBUGCDV, but do not copy all of the changed files to the directory where the debugger expects to find them.
- when the link or DEBUGCDV step fails and you still copy all the files to the directory where the debugger expects to find them.
- when you debug a new level of the application, but there are old LCV files in the directory where the debugger expects to find them, and those LCV files are referenced during the debugger session.

Symptoms that can occur when a mismatch exists are:

- When displaying source code, comment lines are shown as white on blue, which indicates that the debugger thinks that those lines are executable.
- When displaying source code, some executable lines of code are shown in yellow, or some comment lines are shown in white.
- When displaying source code, you cannot set a code breakpoint where you want, because the debugger thinks that that line is a comment line.
- When setting a breakpoint and pressing run, or just pressing single-step, the program you are debugging could abend or hang, because you are in a spot different from where you think you are, and you just placed the breakpoint at a non-instruction boundary. The abend might occur when that instruction is executed, or it might occur at a far later time because that instruction, when it executed, damaged an area of memory or jumped to an inappropriate address.
- Single stepping through a program appears to skip executable lines of code, or single-steps through each line of a set of contiguous comment lines.
- Variables will change or function calls will occur for no reason that seems apparent to you by looking at the source code.
- The debugger fails to stop at the beginning instruction of your program when you start it, showing only the debugger initialization screen. Your program behaves as if the debugger is not controlling it.

13. When displaying memory or typing an address into the storage window of the form ssss:0000, you get the message Can't Display.

Check the selector and offset you specified. You will get this error if the offset+0 address is below the starting address of the selector you specified, or if the offset+15 address is beyond the end of the selector you specified. Adjust the 0000 value so that all 16 bytes are within the boundaries of that selector.

14. Unable to start session.

Specify a fully-qualified path name, including a define if needed. If your application references a Shared Runtime Library, ensure that it is in a location that your application can find.

15. When pressing = or D V to display a variable, it cannot be found.

If you know from your source code that the variable really does exist, there are two possible explanations:

- you might have a mismatching LCV file, or
- the name of the variable is not in your application.CDV file.

You can run DBINFO to determine if the variable exists in the CDV file. Even though a variable is defined as a global within a current source and LCV file, it will be included in the MAP file by the linker *only if a source file uses it*. If no source files reference or change it, then the linker excludes it from the MAP file. If it is not in the MAP file, it will not be in the CDV file. Furthermore, any global variable that is not referenced in a CBASIC source file will be discarded by the compiler for that source file.

A third possible explanation involves the use of the period (.). If you type in a name such as TS.IO.RE, and there are no variables beginning with that (partial) name, the debugger searches again for the symbol TS. This is because the period character is defined as being a separator between the name of a structure and the name of one of its members. In this case, you may get a message that indicates 100's of symbols match that name, because it is so short. An alternate choice might be to type in TS.IO if you know that there are symbols beginning with that string.

Symbol Limits

DEBUGCDV creates a structure in its output *.CDV file that contains every symbol in your *.MAP file. Each symbol name is included in its entirety, along with 7 bytes of overhead per symbol. The debugger limit for the size of this structure is 65,535 bytes. Although there is a separate structure for each compiled unit, there is only one structure for all global symbols. If the size of any structure is close to the limit, a warning message is printed by DEBUGCDV. If the total of all symbols would exceed 65,535 bytes, symbols from that point on are discarded. If a symbol is discarded, the debugger will not be able to find it.

To reduce the space needed for symbols, you can use the /Y## flag when you run DEBUGCDV. ## is a value between 10 and 99. This value specifies the maximum length of any symbol to place in the symbol structure. Symbols longer than this number of characters will be truncated to that length. When accessing in the debugger, only that short name is used.

Another way to reduce the space requirements of symbols for use in the debugger is to avoid using very long global symbol names, or to avoid placing longer names into the global data section.

To minimize the impact of this limit for most programs, DEBUGCDV attempts to split symbol sections that are greater than 64KB, shifting symbol names into the prior section until the overflowing section uses less than 64KB, or the prior section is close to 64KB. This feature nearly doubles the limit, on average, to about 127KB.

This method is not foolproof, but works in most cases because the overflowing section is last, and the prior section is usually a small library function. DEBUGCDV makes no distinction between global variables and function names. If a function name is shifted to the prior section, then source code can't be displayed in the debugger for that function. This feature is turned on by default. To turn it off, specify switch /2 when you run DEBUGCDV.

16. The message CS:IP is in <functionname> can specify the wrong function name.

When you press F7 to determine the name of the current function, the debugger uses the current CS and IP to determine the name of the function that contains that address. Because the compiler only exports function names that are public, the function name cannot be determined correctly unless it is public! If the current function is a local, non-public function, the debugger has no way to determine that that is the case. It can only assume that the current function is the closest public function immediately prior to or at the current CS:IP.

17. When pressing = or D V to display a variable, a message like xxxxxxxx is incorrect is displayed.

You will get this message if you are trying to display a variable that actually is the name of a function.

18. When attempting to exit from the debugger, a popup states that the window must be killed. Follow the instructions to use <AltSys>-W, then <F8>. The debugger loses capability to exit normally back to 4690 OS after two CHAINs have occurred. This is a limitation seen only on debugger exit.

Debugger hints

You can use the debugger to debug your user exit code without the application source. For example, you can start the debugger using the name of the application as the debugged program. This will start the application and the assembler code for the application will display on the debugger display. Use the fast path S and F and enter the name of your user exit. When you press ENTER, the cursor will go to that function. Then use F5 (Run to Cursor) or C C to run the program to the user exit selected. Use F2 (Single Step into Function) or C F to step into the user exit. Don't forget to switch to the application (Ctrl+ALT+TAB) screen and supply the user input if your application requires it. Ctrl+ALT+TAB will switch the display back to the debugger screen. Most of these start-up steps can be automated. See "Keystroke Recording and Playback" on page 27.

Remember that you can press PrtSc from any debugger screen to capture the current contents of that screen to paper for future use. If you want that screen in soft-copy, ensure that your printer is powered off or disconnected, press PrtSc and copy the desired file from ADX_IOSS:ADXIOSPD.???.

If you do have a problem

If you have a problem with the debugger, gather as much information as possible to help debug your problem. Save the source files and files created on diskette. Write down the command line parameters and as many of the symptoms as possible. These steps will help the debugger developers recreate and resolve your problem efficiently. For some classes of problems, it may be necessary for you to furnish your entire DEBUG directory and any additional files necessary to recreate the problem.

You may wish to use Keystroke Recording to accurately document your problem for Toshiba service. See "Keystroke Recording and Playback" on page 27.

Application Debugger Support

The debugger is provided as-is, with no support. Refer to P85703 (5799-RXE) or P85704 (5799-RXF) for support options.

You may submit suggestions to TGCS for improvements to future versions of the debugger, although acceptance by TGCS must not be considered as an agreement to provide that fix. All submissions become property of TGCS.

Debugger Tools

The following tools are available to assist you with formatting the information used by the debugger.

- **DBGINFO**

DBGINFO will help the user debug problems with the linker debug information. DBGINFO will format the .DBG file created by LINK86. The linker version that created the .DBG file is displayed. The module names are displayed along with the lengths of the type information, symbol information, line number, and public symbol sections.

This information will allow the user to verify that the correct debug information was created for a module that is being debugged.

The following are the DBGINFO command formats:

```
DBGINFO          - display help
DBGINFO ?        - display help
DBGINFO path     - display .DBG file from specified path
```

- **DBINFO**

DBINFO will help the user debug problems with the CodeView debug information. DBINFO will format the .CDV file created by DEBUGCDV. For more information on the information displayed see the Microsoft (R)** C Developer's Toolkit Reference.

The following are the DBINFO command formats:

```
DBINFO          - display help
DBINFO ?        - display help
DBINFO path     - display .CDV file from specified path
```

- **SETTRACE**

This tool allows you to alter the number of trace entries kept in the CBASIC call trace buffer, which defaults to 25 entries. Any value between 25 and 8,000 can be specified. SETTRACE must be run against your postlinked application file. NOTE: if the number of entries is set to the default of 25 when the debugger loads your application, it will automatically change it to 400 for that session, and will also turn on the timestamps flag, just as if SETTRACE and SETTIME were run.

The following are the SETTRACE command formats:

```
SETTRACE          - display help
SETTRACE ?        - display help
SETTRACE T pathname - change the number of trace buffer entries in the specified program.
```

- **SETTIME**

This tool allows you to indicate that you want timestamps captured at each CBASIC function call. SETTIME must be run against your postlinked application file. See the NOTE in SETTRACE, above.

The following are the SETTIME command formats:

```
SETTIME          - display help
SETTIME ?        - display help
SETTIME pathname - capture runtime information.
```

Appendix F. Information for Microsoft Windows Users

The preferred Microsoft Windows development platform is Windows 2000 Professional. Limited testing has been performed on Windows 98, NT and XP platforms. Do not use Microsoft Vista.

The following files are available for Microsoft Windows users:

- DEBUGCDV to create a .CDV file for debugger use.
- DBGINFO to format a .DBG file.
- DBINFO to format an existing .CDV file.

BASIC programs can be compiled and linked using Microsoft Windows. You must have 4690 BASIC.EXE version 3 or later, and LINK86.EXE version 2.39 or later. Compile BASIC programs using Microsoft Windows by renaming the BAS.BAT file to a BAS.CMD file, setting up your Microsoft Windows path to include the Microsoft Windows executables, adding the include files used in your program to the DPATH, and executing the BAS.CMD file.

Some prior versions of DEBUGCDV.EXE fail with a memory allocation error when run against a large application on a Windows 98 or XP platform. This problem is fixed with the current version 4.

Appendix G. Program Messages and Actions

DEBUGFRM Messages and Actions

The following is a list of the possible errors from the DEBUGFRM program and their associated actions:

Message	Description
Elapsed time: nnnn second(s).	Information
Error allocating listing file buffer.	Could not allocate enough memory for listing file buffer. Free memory by reducing number of programs running and retry
Error creating xxxxx file.	Error attempting to create the LCV file.
Error locating ":" in source line.	Contact your TGCS program support representative.
Error opening xxxxx file.	Error attempting to open the BASIC LST file.
Error writing listing buffer.	Contact your TGCS program support representative.
Format Basic listing file complete.	Information
Formatting Basic listing file ...	Information
Invalid buffer size requested.	Input correct buffer size from 1 to 31.
Invalid file name specified.	File name is too long. Reduce file name length and retry.
Invalid line length error.	Make sure that the I or the X option was not used during compile.
Read listing file error.	Contact your TGCS program support representative.
Seek error backing up source.	Contact your TGCS program support representative.
Source line number error.	Make sure the your source file does not contain a %NOLIST command. If your file does not contain a %NOLIST command, you need to contact your TGCS program support representative and supply the

DEBUGCDV Messages and Actions

The following is a list of the possible errors from the DEBUGCDV program and their associated actions:

Message	Description
Allocate line number buffer error.	Could not allocate enough memory for line number buffer. Free memory by reducing number of programs running and retry.
Allocate memory error.	Release memory in your controller by reducing the number of programs running and retry.
Allocate symbols buffer error, size = xxxxx	Could not allocate enough memory for symbols buffer. Free memory by reducing number of programs running and retry.
Allocate type buffer error, size = xxxxx	Could not allocate enough memory for type buffer. Free memory by reducing number of programs running and retry.
Convert debug information complete.	Information
Converting debug information...	Information
Debug information files not found.	The files needed to build a CDV file cannot be found. The files are the .SYM, .DBG, and .MAP or the .SYM, .MAP, and .LIN files.
Elapsed time: xxxxx second(s).	Information
Error creating xxxxx file.	Error attempting to create the CDV file.
ERROR - symbol <s> is ignored because of 64KB limitation.	The symbol shown would cause the symbol structure in your CDV file to exceed 65535 bytes. It is discarded. See Symbol Limits on page 51 .
Find module for data segment update error.	Contact your TGCS program support representative.
Find module index error.	Contact your TGCS program support representative.
Invalid file name specified.	File name is too long. Reduce file name length and retry.
Length keyword not found in .MAP file error.	Contact your TGCS program support representative.
Line number buffer overflow error.	A module contains too many line number records. Reduce the number of lines in the module.
Line number module name not found.	Contact your TGCS program support representative.
Link with shared runtimes error.	The application was linked with the shared runtimes. Re-link the application with the noshare (NOSH) option.
Module index error.	Use the map[all] option on the link phase. This option should be used for information in the .L86 files.
No line number information found.	DEBUGCDV did not find any line number information. Source level debugging will not be possible.
No line numbers for module error.	A module is in the LIN file with no line numbers detected. Check format of the LIN file.
Read .MAP file error.	Contact your TGCS program support representative.
Read line information in debug file error.	This error will occur if the executable and source files do not match. It is caused by changing the source files and not recompiling or an old level of source or executable in the search path. Correct this condition and continue. If the problem persists, contact your TGCS program support representative.
Read line length in debug file error.	Contact your TGCS program support representative.
Read public length in debug file error	Contact your TGCS program support representative.
Read symbol information length in debug file error.	Contact your TGCS program support representative.
Read symbols information in debug file error.	Contact your TGCS program support representative.
Read type information in debug file error.	Contact your TGCS program support representative.

Seek public information in debug file.	Contact your TGCS program support representative.
Seek type information in debug file error.	Contact your TGCS program support representative.
Segment overflow error.	Contact your TGCS program support representative.
WARNING - Module index n contains b bytes of symbol information. The max limit is 65535.	The cumulative length of all symbols cannot exceed 65535 bytes for any module. This message is printed because you are close to the limit. See Symbol Limits on page 51.
WARNING - zero length name at file pos x, skipping.	The *.DBG file contains a structure with a null global name. The entire structure is discarded. You can ignore these messages if you are able to locate all of the symbols you need in the debugger. See Symbol Limits on page 51.
Writing object module data...	Information
Writing public data...	Information
Writing subsection directory...	Information
Writing symbol information...	Information
Writing trailer...	Information
Writing type information...	Information

DEBUG Messages and Actions

The following is a list of the possible errors from the DEBUG program and their associated actions:

Message	Description
xxxxxxx.LCV does not match its OBJ file	The LCV file does not correlate with the line number information present for it in the CDV file. You probably recompiled that module and forgot to copy the LCV file for it into the current directory.
xxxxx Interrupt yyyyy	The application has stopped due to an execution error. yyyyy is the location of the error. xxxxx is one of the following errors: <ul style="list-style-type: none"> • Divide • Overflow • Bound • Invalid Opcode • Processor extension not available • Processor extension error • Stack fault
xxxxx is incorrect	The debugger could not determine what type of expression you entered. Try an expression of a different form.
xxxx:xxxx is in <functionname>	The cursor is located at the specified selector:offset, which is in contained within the indicated function or subroutine.
ssss:oooo module unknown	A trap occurred, but the debugger could not determine which module contains the trap.
xxxxx not active	The variable you attempted to display is not currently in scope.
xxx symbols match ssssssss	You typed a partial-match symbol ssssssss. More than 100 symbols contain that partial name. Retry with a longer or different name. If the name you typed contains a period, use a shorter part of that name. A re-scan may have been performed based upon the token prior to the period, which is by definition the name of a structure.
Application Exited	The application exited normally
Application not postlinked	The application 286 file must be postlinked using the POSTLINK utility. Postlink the application 286 file and retry.
Application path too long	Full application name must be less than 30 characters. The application name length can be reduced by using logical names.
Appl waiting xxxxx	The application is waiting on the operating system or user input.
Attempt to access console table failed. XXXXXXXX	An error occurred while reading the keyboard. Contact your TGCS program support representative.
Attempt to halt failed. XXXXXXXX	4690 OS did not halt the application. Contact your TGCS program support representative.
Attempt to load source failed.	Probably a memory allocation failure.
Auxiliary console is not supported.	Use the main console. 25 rows are required.
Back level .CDV file format. Re-run DEBUGCDV	This occurs from debug if the user is attempting to use a back level .CDV file. All .CDV files must be rebuilt for the new version 4.
Background applications do not have windows	You have attempted to switch to an application that has no display window.
xx breakpoints were set. xx breakpoints were cleared.	The sequence C P was used to set or clear a breakpoint at the beginning of each function in the current file.
Can't continue	An unrecoverable error has occurred while single stepping

	a function, stepping into a function or displaying a function call stack.
Can't display	The address of a variable to be displayed is currently invalid.
Can't evaluate expression	The debugger cannot determine the result of the expression you entered. Examine the expression you entered. You may have made a typing error, or the debugger may not be able to resolve that type of expression.
Can't find xxxxx	The indicated symbol does not exist. Try a different name or a shorter name. If that still fails, run DBINFO to determine if the name exists in your CDV file.
Can't modify this data	You tried to modify data that cannot be changed.
Can't set breakpoint	The debugger could not find the address where the breakpoint should be set.
Can't Step Instruction	The debugger can not step an instruction that will step into global address space.
cancel error, XXXXXXXX	A STOPSESSION command failed. Contact your TGCS program support representative.
Cannot find : in xxxxx	Contact your TGCS program support representative.
Cannot Find Entry Point.	The debugger could not find the starting point of your program.
Cannot Switch to Assembler - Make sure cursor is on a valid source line	The debugger needs the cursor to be on an executable line. This allows the debugger the find the corresponding assembler statement.
Close Communications Pipe Error, XXXXXXXX	s_close failed. Contact your TGCS program support representative.
Constant required	A variable value was encountered in an expression where a constant value is expected.
Could not display the Application's window	SHOWAPPWINDOW failed. Contact your TGCS program support representative.
Could not read Help pipe, XXXXXXXX	s_read from DEBUG:HELP.286 failed. Contact your TGCS program support representative.
CS:IP is in <functionname>	The current instruction is in public function functionname.
CS:IP was changed from ssss:0000 to ssss:0000.	The address of the current instruction was changed because you pressed Ctrl+F6.
Cursor must be on a variable name	Attempted to use the 'INS' or '=' key where there was no variable.
Data Breakpoint cleared at xxxx:xxxx	A data breakpoint was cleared.
Data Breakpoint set to xxxx:xxxx for <symbol>	A data breakpoint was set on a variable.
Data Breakpoint was NOT added - only four are allowed.	The Intel architecture provides capability for only four breakpoint addresses. Clear one of your existing data breakpoints before you set any more.
DBIF [filename.LCV] does not have line numbers. ASM code is shown.	You compiled filename with the line number option off.
DBIF aborting search for s due to a possible data structure problem.	Symbol s contains an abnormal name length. This error is usually caused by too many global symbols.
DBSEGS get address error, rc = XXXXXXXX	Check to be sure the link did not use the MAP[ALL] option or map the shared runtimes. Contact your TGCS program support representative.
DBSEGS.1 selector error DBSEGS.2 vtype=n	A selector in your application could not be read. Contact your TGCS program support representative.
DEBFILE get address error, rc = XXXXXXXX	Check to be sure the link did not use the MAP[ALL] option or map the shared runtimes. Contact your TGCS program support representative.
DEBFILE.1 selector error DEBFILE.2 selector error	A selector in your application could not be read. Contact your TGCS program support representative.

Debug marker not found at EOF	Your CDV file is probably corrupt. "NB00" is missing.
Debug System Error nnnn	An unrecognized "stop" code was passed from the OS for the controlled application. Contact your TGCS program support representative.
Define error for xxxxx, XXXXXXXX	Contact your TGCS program support representative.
Divide Interrupt yyyyy	The application has stopped due to a divide error. yyyyy is the location of the error.
DosAllocSeg failed. RC = XXXXXXXX	Memory could not be allocated to store the LDT. Contact your TGCS program support representative.
DosPtrace Anomaly	An unrecognized completion code has occurred while the OS was stepping or running your application. Contact your TGCS program support representative.
DOSRAID 2nd KCTRL failed, rc=XXXXXXXX	An error occurred while setting up the CTRL-ALT-TAB key. Contact your TGCS program support representative.
DOSRAID Allocate LDT memory error	Could not get enough memory for operation. Free memory by reducing number of programs running and retry.
DOSRAID Cancel error, rc=XXXXXXXX	DOSSTOPSESSION failed. Contact your TGCS program support representative.
DOSRAID DBG VC Create failed, rc=XXXXXXXX	The virtual console for the debugger could not be created. Contact your TGCS program support representative.
DOSRAID First order failed, rc=XXXXXXXX	An attempt to show the application window failed. Contact your TGCS program support representative.
DOSRAID get address error, rc=XXXXXXXX	Check to be sure the link did not use the MAP[ALL] option or map the shared runtimes. Contact your TGCS program support representative.
DOSRAID Get DBG VC table error, rc=XXXXXXXX	s_get for the debugger Virtual Console table failed. Contact your TGCS program support representative.
DOSRAID Get VC table error, rc=XXXXXXXX	s_get for your application's Virtual Console table failed. Contact your TGCS program support representative.
DOSRAID KCTRL failed, rc=XXXXXXXX	CTRL-ALT-TAB could not be set up as the return key. Contact your TGCS program support representative.
DOSRAID Last order failed, rc=XXXXXXXX	The Debugger Virtual Console could not be shown. Contact your TGCS program support representative.
DOSRAID Load error for xxxxx, rc=XXXXXXXX	Your application could not be loaded. If the rc is not helpful, contact your TGCS program support representative.
DOSRAID ptrace unsupported command, command=x	The debugger requested an unsupported kernel command. Contact your TGCS program support representative.
DOSRAID Read after give failed, rc=XXXXXXXX	A keyboard error occurred after giving control to the application. Contact your TGCS program support representative.
DOSRAID Reset STDERR failed, rc=XXXXXXXX	s_define failed for STDERR. Contact your TGCS program support representative.
DOSRAID Reset STDIN failed, rc=XXXXXXXX	s_define failed for STDIN. Contact your TGCS program support representative.
DOSRAID Reset STDOUT failed, rc=XXXXXXXX	s_define failed for STDOUT. Contact your TGCS program support representative.
DOSRAID Set DBG VC table error, rc=XXXXXXXX	s_set for the debugger's Virtual Console table failed. Contact your TGCS program support representative.
DOSRAID Set environment error, rc=XXXXXXXX	An attempt to get or set STDIN, STDOUT or STDERR failed. Contact your TGCS program support representative.
DOSRAID Set STDERR failed, rc=XXXXXXXX	s_define failed for STDERR. Contact your TGCS program support representative.
DOSRAID Set STDIN failed, rc=XXXXXXXX	s_define failed for STDIN. Contact your TGCS program support representative.
DOSRAID Set STDOUT failed, rc=XXXXXXXX	s_define failed for STDOUT. Contact your TGCS program support representative.
DOSRAID Set Virtual Console table error, rc=XXXXXXXX	A switch to your application's Virtual Console table failed. Contact your TGCS program support representative.

DOSRAID Unable to get console table, rc=XXXXXXXX	s_get failed for the debugger console table. Contact your TGCS program support representative.
DOSRAID Unable to give keyboard to APP, rc=XXXXXXXX	Keyboard control could not be switched to your application.. Contact your TGCS program support representative.
DOSRAID Unable to give keyboard to DBG, rc=XXXXXXXX	Keyboard control could not be switched to the debugger. Contact your TGCS program support representative.
DOSRAID Unable to Open keyboard file, rc=XXXXXXXX	s_open failed for the Virtual Console Keyboard. Contact your TGCS program support representative.
DOSRAID Unable to Open screen file, rc=XXXXXXXX	s_open failed for the Virtual Console Screen. Contact your TGCS program support representative.
DOSRAID Unable to reset Console Fnum, rc=XXXXXXXX	The Virtual Console could not be switched from your application back to the debugger. Contact your TGCS program support representative.
DOSRAID VC Create failed, rc=XXXXXXXX	The application Virtual Console could not be created. Contact your TGCS program support representative.
Error closing xxxxx	s_close failed for the indicated file. Contact your TGCS program support representative.
Error closing handle	s_close failed for an open file. Contact your TGCS program support representative.
Error creating xxxxx	The indicated file could not be created. Contact your TGCS program support representative.
Error opening xxxxx	The indicated file could not be opened. Contact your TGCS program support representative.
Error reading handle	An s_read failed. Contact your TGCS program support representative.
EWTTTERM Appl Cancel	The debugger kernal EWTTTERM received a request to cancel the controlled application. This happens when you exit the debugger or restart the application using C A Y.
EWTTTERM is ending	The debugger kernal EWTTTERM is terminating. This happens when you exit the debugger or restart the application using C A Y.
EWTTTERM is running	The debugger kernal EWTTTERM has just started, or has just canceled your controlled application. This happens when you exit the debugger or restart the application using C A Y.
Exception before procedure.	The application had program exception prior to entering the procedure you tried to step into.
Exit	The application exited normally.
Find 286 failed for ssss.	An executable program could not be found. Contact your TGCS program support representative.
Find module index error	Use map[all] on the link step. Problem is caused by codeview information in an .L86 file.
FRopen failed, RC = XXXXXXXX	An error occurred while opening a source code file.
Function not available after exit	The selected function is not available after the application has exited.
get 1st line number in module error rc=XXXXXXXX	The main program was not found, or line number information is not present for it. Contact your TGCS program support representative.
Get system variables error rc=XXXXXXXX	The debugger could not access the system variables. Please report this to your TGCS representative.
go error, XXXXXXXX	Your application could not be run. Contact your TGCS program support representative.
GO - AppPTB.cmd =d, rc=XXXXXXXX	The debugger requested an unsupported command. Contact your TGCS program support representative.
halt error, XXXXXXXX	Your application could not be stopped. Contact your TGCS program support representative.
Help could not be displayed, XXXXXXXX	A Help request could not be sent to the help process.

	Please report this problem to your TGCS representative.
Help not available	There is no help available for this screen. Please report this problem to your TGCS representative.
Help process did not start, no help will be available	The debugger starts a separate program to handle help requests. This program could not be started. The debugger will still work, but you will not be able to see any help screens. Check to verify that HELP.286 is in the SYSTEM: (ADX_SPGM) subdirectory and the .HLP files are in the HELP: (usually DEBUG) subdirectory.
I cannot display help for this entry, XXXXXXXX	The help process could not display the requested data. Please report this to your TGCS representative.
I cannot display this memory location - xxxxx	The memory address you entered cannot be accessed by the debugger.
I cannot find the variable xxxxx	The debugger cannot find the variable name you entered. It may be misspelled, or the debugger may not have the necessary symbol information to show this variable.
I cannot modify data at this time	You pressed tab or right arrow to modify a line of data, but the debugger cannot change that data.
Improper use of /m switch	You must use the /m switch when debugging applications that are compiled for Medium model. If your application is not medium model, do not use the /m switch.
Incorrect expression	The expression entered in the variable window is not valid
Incorrect PID or Sequence Number on Message	A Process ID for the controlled application is incorrect, or a message was received out of sequence. Contact your TGCS program support representative.
INIT01: Unable to get Parent Console number, rc=XXXXXXXX	s_get failed. Contact your TGCS program support representative.
INIT02: Unable to get the Console table, rc=XXXXXXXX	s_get failed. Contact your TGCS program support representative.
INIT03: Unable to create root V-console, rc=XXXXXXXX	The debugger's Virtual Console could not be created. Contact your TGCS program support representative.
INIT04: Unable to get Root V-console table, rc=XXXXXXXX	s_get failed for the debugger's Virtual Console table. Contact your TGCS program support representative.
INIT05: Unable to set ROOT V-console table, rc=XXXXXXXX	s_set failed for the debugger's Virtual Console table. Contact your TGCS program support representative.
INIT06: Reset STDIN failed, rc=XXXXXXXX	STDIN could not be redirected. Contact your TGCS program support representative.
INIT07: Reset STDOUT failed, rc=XXXXXXXX	STDOUT could not be redirected. Contact your TGCS program support representative.
INIT08: Reset STDERR failed, rc=XXXXXXXX	STDERR could not be redirected. Contact your TGCS program support representative.
INIT09: Unable to Open screen file, rc=XXXXXXXX	The debugger's virtual console could not be opened. Contact your TGCS program support representative.
INIT10: Unable to Open keyboard file, rc=XXXXXXXX	The debugger's keyboard could not be opened. Contact your TGCS program support representative.
INIT11: Get environment error, rc=XXXXXXXX	s_get failed for STDIN, STDOUT, STDERR. Contact your TGCS program support representative.
INIT12: Set environment error, rc=XXXXXXXX	s_set failed for STDIN, STDOUT, STDERR. Contact your TGCS program support representative.
INIT13: Unable to get console table, rc=XXXXXXXX	s_get for the debugger's Virtual Console table failed. Contact your TGCS program support representative.
INIT14: Unable to reset Console table, rc=XXXXXXXX	s_set for the debugger's Virtual Console table failed. Contact your TGCS program support representative.
INIT15: Unable to give keyboard to DBG, rc=XXXXXXXX	Keyboard control could not be established. Contact your TGCS program support representative.
INIT16: Unable to put debug window on top, rc=XXXXXXXX	The Virtual Console could not be assigned to the debugger. Contact your TGCS program support representative.

INIT17: Unable to allocate LVB memory, rc=XXXXXXXX	Memory could not be allocated for the display buffer. Contact your TGCS program support representative.
INIT18: Unable to allocate character plane	Memory could not be allocated for the display buffer. Contact your TGCS program support representative.
INIT19: Unable to allocate attribute plane	Memory could not be allocated for the display buffer. Contact your TGCS program support representative.
INIT20: Unable to create Help V-console, rc=XXXXXXXX	The Virtual Console could not be created. Contact your TGCS program support representative.
INIT21: Unable to put help screen on bottom, rc=XXXXXXXX	s_order failed. Contact your TGCS program support representative.
INIT22: Unable to get help's v-console table, rc=XXXXXXXX	s_get failed. Contact your TGCS program support representative.
INIT23: Unable to set help's v-console table, rc=XXXXXXXX	s_set failed. Contact your TGCS program support representative.
INIT24: Unable to save define for stdin, rc=XXXXXXXX	The value of STDIN could not be determined. Contact your TGCS program support representative.
INIT25: Unable to save define for stdout, rc=XXXXXXXX	The value of STDOUT could not be determined. Contact your TGCS program support representative.
INIT26: Unable to save define for stderr, rc=XXXXXXXX	The value of STDERR could not be determined. Contact your TGCS program support representative.
INIT27: Unable to redefine stdin for help, rc=XXXXXXXX	STDIN could not be redirected. Contact your TGCS program support representative.
INIT28: Unable to redefine stdout for help, rc=XXXXXXXX	STDOUT could not be redirected. Contact your TGCS program support representative.
INIT29: Unable to redefine stderr for help, rc=XXXXXXXX	STDERR could not be redirected. Contact your TGCS program support representative.
INIT30: Create To Help pipe failed, rc=XXXXXXXX	A pipe could not be created. Contact your TGCS program support representative.
INIT31: Create From Help pipe failed, rc=XXXXXXXX	A pipe could not be created. Contact your TGCS program support representative.
INIT32: Unable to start system:HELP.286, rc=XXXXXXXX	ADX_SPGM:HELP.286 could not be started. Contact your TGCS program support representative.
INIT33: Unable to restore stdin, rc=XXXXXXXX	s_define failed for STDIN. Contact your TGCS program support representative.
INIT33: Unable to restore stdout, rc=XXXXXXXX	s_define failed for STDOUT. Contact your TGCS program support representative.
INIT34: Unable to restore stderr, rc=XXXXXXXX	s_define failed for STDERR. Contact your TGCS program support representative.
INIT35: Unable to read init code from system:HELP.286, rc=XXXXXXXX	The Help process is not communicating with the debugger. Contact your TGCS program support representative.
INIT36: system:HELP.286 failed to initialize, rc=XXXXXXXX	The Help process did not initialize within 10 seconds. Contact your TGCS program support representative.
INIT37: Can't communicate with system:HELP.286 process	The Help process is not communicating with the debugger. Contact your TGCS program support representative.
INIT38: Open DEBUG:DEBUGLOG failed, rc=XXXXXXXX	Re-IPL your controller to activate logical names. If this fails, contact your TGCS program support representative.
INIT39: Write error, rc=XXXXXXXX	A write to DEBUG:DEBUGLOG failed. Contact your TGCS program support representative.
INIT40: Get pid number error, rc=XXXXXXXX	The Process ID of your application could not be obtained. Contact your TGCS program support representative.
Instruction Step in <function>.	One source or assembler instruction was executed. The current CS:IP is in the named function .
Insufficient free memory available for extended processing.	Could not get enough memory for operation. Free memory by reducing number of programs running and retry.
Insufficient memory to cache GDT or LDT for processing.	Could not get enough memory for operation. Free memory by reducing number of programs running and retry.
Interrupted System Call (SP may be wrong)	A break or error has occurred while processing a system

	call. The SP should not be trusted. Continuing from this point may not be possible.
Invalid Address	The address you attempted to display is not a valid address.
Invalid BASIC Format	You attempted to format a variable as a BASIC string and the variable is not in a format that can be converted.
Invalid dump file	Dump file contains an invalid number. Erase the APPLDUMP file and rerun EWTXDMP.
Invalid Message Length From Kernel, XXXXXXXX	A response message from EWTCONT.286 or EWTTTERM.286 is incorrect. Contact your TGCS program support representative.
Invalid number of parameters	Correct your command syntax and retry. Refer to the user's guide for more information.
Invalid Opcode Interrupt yyyyy	The application has stopped due to an invalid opcode error. yyyyy is the location of the error.
Invalid operand for xxxxx	The operand indicated in the message is not valid in the expression you entered.
Invalid pointer for xxxxx	This pointer contains an address that cannot be used by your application.
Invalid type for "-"	The variable or constant used is incompatible with the operator.
Invalid use of "."	This operator was used incorrectly in an expression.
Invalid value for data type	You have attempted to modify a variable with invalid data.
Invalid wait parameter, nn	Correct the wait parameter and retry. The range is from 1 to 32,767.
Invalid 286 file.	Contact your TGCS program support representative.
Kernal Error Exit, XXXXXXXX	DEBUG:EWTCONT.286 could not be started. Contact your TGCS program support representative.
lineno table load failed. xxxx xxxx	The Line Number table could not be loaded for the indicated selector and offset. Contact your TGCS program support representative.
Loaded RC=00000000	The debugger kernal EWTTTERM finished loading a controlled application, or the load failed with the indicated return code.
LoadingR::C:\DEBUG\EAMTS10L.286	The debugger kernal EWTTTERM received a command to start the indicated application.
Lost breakpoint in xxxxx at xxxxx	While removing breakpoints, a preexisting breakpoint could not be located. Contact your TGCS program support representative.
No data breakpoints are set.	Set at least one data breakpoint before using D D to set the breakpoint conditions.
No String has been entered	The function 'Find Next String' cannot be used until a string has been entered. Use the function 'Search For String' to enter a string to search for. Then use 'Find Next String' to find the next occurrence.
ON ERROR is not initialized..	You typed O to display the current ON ERROR routine, but no ON ERROR statement has been executed yet.
Open failed for xxxxx	The file named in the message could not be opened. Make sure the program being debugged and its CDV file are in the search path.
Overflow	The application terminated unexpectedly with an overflow error.
Panic nnnn -- Press any key to continue	An unexpected, unrecoverable error condition has occurred. Write down the code and contact your TGCS program support representative.
Printing n entries. Printed n entries to DEBUGLOG	The entire contents of the CBASIC Call History is formatted and saved to file DEBUGLOG in the current directory.
Program name not specified	The debugger was started with options but no program

	name. Specify the program name you wish to debug after the options
Protection Exception xxxxx	The application terminated unexpectedly with a protection exception.
Put the cursor on a variable to set a Data Breakpoint	You tried to set a data breakpoint on a name that is not a variable; or the variable is not in scope. See Unable to Display Variable on page 46.
Put the cursor on an executable source line	To determine the name of the function where the cursor is located, or to set the CS:IP, the cursor must be on a line of executable code.
read memory error, XXXXXXXX	An attempt to read memory in your application has failed. Contact your TGCS program support representative.
read registers error, XXXXXXXX	An attempt to read the registers in your application has failed. Contact your TGCS program support representative.
Resize storage window error	An attempt to make the window smaller or larger has failed. Contact your TGCS program support representative.
SD86RAID selector error xxxx for xxxx:xxxx xxxx	A selector in your application could not be read. Contact your TGCS program support representative.
select chained application error, rc=XXXXXXXX	An attempt to chain to another application failed. Contact your TGCS program support representative.
select current application error, rc=XXXXXXXX	An attempt to chain to another application failed. Contact your TGCS program support representative.
Set on CDOSExit failed, rc=XXXXXXXX	Your application terminated, and cleanup failed. Contact your TGCS program support representative.
SHOWCF Error - [filename.LCV] appears to mismatch its OBJ file at line n.	The CDV file and the indicated LCV file are out of sync. Check the date and time of both files, or rebuild them both.
single step error, XXXXXXXX	An error occurred while single-stepping your application. Contact your TGCS program support representative.
Sorry, this function is not available.	The function you selected is not implemented yet, but may be implemented in a future version of the debugger.
Source file >>>ssss<<<< not found	The indicated file was not found in your PATH.
Stack fault Interrupt yyyyy	The application has stopped due to a stack fault. yyyyy is the location of the error.
Stopped at Breakpoint in <function>.	A code breakpoint was hit. The current CS:IP is in the named function .
Stopped at Data Breakpoint for <symbol> in <function>.	The symbol shown was altered and execution stopped at the next sequential instruction, which is in the named function . You can continue by using single step or run.
String not found	The string you searched for is not found in the program source
Switch to offline mode failed, XXXXXXXX = xxxxx	The debugger could not switch the terminal into offline mode. Please report this to your TGCS representative.
Switch to online mode failed, XXXXXXXX = xxxxx	The debugger could not switch the terminal into online mode. Please report this to your TGCS representative.
Terminal application name too long	The application name must be 21 characters or less in length.
Terminal applications do not have windows	You have attempted to switch to an application that has no display window.
Terminal is already in offline mode	You attempted to put the terminal in offline mode when it was already in that mode.
Terminal is already in online mode	You attempted to put the terminal in online mode when it was already in that mode.
The Help function is not available for this session	You asked for help, but the debugger thinks the help process did not start correctly. There should have been a message displayed when you started the debugger, indicating the help process did not start. If this message was not displayed, contact your TGCS representative.
There are no pipes to display	The application has no pipes open.

There are no windows for terminal applications	You have attempted to switch to an application that has no display window.
There is no other file available	You have attempted to go to the next file when there is no other file available
Too many dynamic applications	You attempted to start a dynamic application and exceeded the number of dynamic applications allowed. Reduce the number of windows and running applications and retry.
'tracebuffer' was not found	The CBASIC call trace buffer is not present in your application.286 file. You must relink your application with the version of SB286L.L86 or SB286LT.L86 that was released in February, 2003, or later.
'tracebuffer' is not initialized	The CBASIC call trace buffer is not yet initialized because there have been no CALLs executed. Try again later.
TYPES case error... TYPES error...	The debugger found an unrecognized variable type. This error usually occurs when you display a variable that is out of context for the current file. One such example is a global variable that is not used in the currently-displayed file.
Unable to initialize 286.	Contact your TGCS program support representative.
Unable to open the CDV file.	DEBUG could not open the CDV file for the application you are attempting to debug. Be sure that the CDV file for the application is in the same location as the 286 file.
Unable to open the 286 file.	DEBUG could not open the 286 file for the application you are attempting to debug. Be sure that the 286 file is in the current directory or in the path specified on the DEBUG command line.
unable to read GDT and LDT info segs	The Global and Local Descriptor Tables could not be read. Contact your TGCS program support representative.
Unable to start session.	The application either is non-executable, or the OS refused to start it. Contact your TGCS program support representative.
Unknown code	The debugger passed an unsupported command to SD86RAID. Contact your TGCS program support representative.
Unrecognized option "xxxxx"	The option specified on the DEBUG command line is invalid.
Variable <varname> was not found.	The variable is not in your CDV file. You can run DBINFO to determine all of the names present in your CDV file.
VioGetBuf() returned nnnn	Initialization error. Contact your TGCS program support representative.
VioGetCurType() returned nnnn	Initialization error. Contact your TGCS program support representative.
VioGetMode() returned nnnn	Initialization error. Contact your TGCS program support representative.
VioSetMode() needed here	Initialization error. Contact your TGCS program support representative.
wait error, XXXXXXXX	An unsupported RC occurred when transferring control from the application back to the debugger. Contact your TGCS program support representative.
Write log file error, rc=XXXXXXXX	An error occurred while writing to DEBUG:DEBUGLOG. Contact your TGCS program support representative.
write memory error, XXXXXXXX	An attempt to write memory in your application has failed. Contact your TGCS program support representative.
write registers error, rc=XXXXXXXX	The debugger failed to update your application program's registers. Contact your TGCS program support representative.
You can't set a breakpoint on this line	You tried to set a breakpoint on a non-executable line, or a line that cannot be stopped on.
You can't stop execution on this line	You want to run to the cursor location, but the cursor is not

	on a valid stopping point.
You need to configure a User Logical File name for 'debug:'	'debug:' is assigned to 'C:\DEBUG\' when the debugger is installed from diskette or CD-ROM. Your system was not installed from diskette or CD-ROM using INSTALL.BAT, the debug: define was changed, or DEBUG: is not the current directory.
You need to configure a User Logical File name for 'help:'	'help:' is assigned to 'C:\DEBUG\' when the debugger is installed from diskette or CD-ROM. Your system was not installed from diskette or CD-ROM using INSTALL.BAT, the help: define was changed, or DEBUG: is not the current directory.

Appendix H. 4690 Application Debugger Reference Card - February , 2011

KEY	FUNCTION (Source Window)	FUNCTION (Assembler Window)	FUNCTION (Storage Window)
F1	Displays context-sensitive help message.	Displays context-sensitive help message.	Displays context-sensitive help message.
F2	Single step the target program. Step into	Single step the target program (trace) with code breakpoints.	n/a
C_F2	Run to the next CBASIC function call	Run to the next CBASIC function call	n/a
F3	Exit the program and debugger.	End the debug session and the program being debugged.	n/a
F4	Set or reset a breakpoint.	Set or reset a program execution breakpoint on the cursor line.	n/a
A_F4, C_F4	Set or reset a data breakpoint.	n/a	n/a
Shift_F4	Indirectly set or reset a data breakpoint.	n/a	n/a
F5	Run program to the cursor or breakpoint.	Run program to the cursor or breakpoint.	n/a
F6	Run the target program.	Run the target program.	n/a
C_F6	Set CS:IP to this source line.	n/a	n/a
F7	Place the cursor at the instruction to be executed. Find executable line, will switch files. Show Func name.	Switch to the file containing the statement or instruction that is about to be executed and place the cursor on that line.	n/a
A_F7	Redisplay source window that was displayed when you last pressed F2, F5 or F6. Will switch files.	Redisplay the source window that was displayed when you last pressed F2, F5 or F6. Will switch files.	n/a
C_F7	Show the instruction that calls the current function.	Show the instruction that calls the current function.	n/a
F8	Locate a given function name.	Locate a given function name.	Print all global variables
Shift_F8	Locate a local function name.		
F9	n/a	n/a	Change the formatting of the variable at the cursor.
F10	Jump to the action bar.	Jump to the action bar.	n/a
CTRL+BREAK	Stop running program.	Stop running program or application being debugged.	n/a
SPACEBAR	Single step the target program.	Single step the target program (no trace) with code breakpoints enabled.	Type over characters in the formula.
PLUS(+)	Show the variable window.	Show the variable window with the selected variable values.	n/a
MINUS(-)	Hide the variable window.	Remove the variable window.	n/a
EQUAL(=)	Display the variable at the cursor.	n/a	n/a
*	Display the data pointed to by the cursor.	n/a	n/a
&	Display all addresses of variable at the cursor.	n/a	n/a
INS	Place the variable at the cursor into the storage window.	n/a	Toggle insert mode when editing storage.
C_INS	Place the data pointed to by the variable at the cursor into the storage window.	n/a	n/a
DELETE	n/a	n/a	Delete characters in the formula or storage display area.
C_DELETE	n/a	n/a	Delete line from the storage window.
UP ARROW	Move the cursor up one line.	Move the cursor up one line.	Move the cursor up one line.
C_UP ARROW	n/a	n/a	Shrink storage window by one line.
DOWN ARROW	Move the cursor down one line.	Move the cursor down one line.	Move the cursor down one line.
C_DOWN ARROW	n/a	n/a	Expand the storage window to show one more line.
RIGHT ARROW	Move the cursor right one word.	n/a	Edit storage using the formula; move the cursor.
LEFT ARROW	Move the cursor left one word.	n/a	Move the cursor.
TAB	Move the cursor right one word.	n/a	Edit storage using the formula.
S_TAB	Move the cursor left one word.	n/a	n/a
BACKSPACE	Move the cursor left one word.	n/a	Move the cursor (dragging).
PGUP	Scroll to the previous window.	Scroll to the previous window in current view.	Show previous screen in current file.
PGDN	Scroll to the next window.	Scroll to the next window in the current view.	Show next screen in current file.
HOME	Move the cursor to the first line.	Move the cursor to the first line of the current window.	Move the cursor to the first line on the current screen.
C_HOME	Scroll to the first window.	n/a	Display first screen in current file.
END	Move the cursor to the bottom.	Move the cursor to the last line of the current window.	Move the cursor to the last line on the current screen.
C_END	Scroll to the last window in the current view.	n/a	n/a
C_ENTER	n/a	n/a	Add a blank line to the storage window.
CTRL+ALT+TAB	Show the target program's screen.	Toggle to/from the target application's screen	n/a
@	Show the CS:IP and function name of the line at the cursor.	n/a	n/a
C_u, C_d, C_t, C_b	n/a	n/a	Move the line up, down, to top, to bottom