



SOFTWARE DEVELOPMENT SYSTEM

SIMULATOR/DEBUGGER



THIS PAGE LEFT INTENTIONALLY BLANK



Table of Contents

CHAPTER 1 Overview	7
Welcome to the Wonderful World of WDCDB	7
CHAPTER 2 Before Running WDCDB	9
C Language Program Debugging	9
Assembly Language Program Debugging	9
Linking for Debugging	9
The WDCDB.INI File	9
CHAPTER 3 Running WDCDB	11
Starting WDCDB	11
Startup Options	11
Loading a Program	11
Loading The Symbol Table	12
Where to Find the Source Files	12
Running Programs	12
Stopping the Program	12
Single Stepping the Program	12
Breakpoints	13
Temporary Breakpoint	13
Watches	13
CHAPTER 4 Menus	15
File Menu	15
Load and Reload Program and Symbols	15
Save and Load Desktop	15
Save Breaks and Watches	15
Exit WDCDB	15
Edit Menu	15
Text Search	16
Code and Data Position	16
Edit Startup Options	16
Edit Runtime Options	16
View Menu	17
View Code Menu Command	17
View Data Menu Command	17
View File Menu Command	17
View Registers Menu Command	17
View Stack Menu Command	17
View Breakpoints Menu Command	17
View Watches Menu Command	17
View Symbols Menu Command	17
View Modules Menu Command	17
View Locals Menu Command	18
Run Menu	18
Go Menu Commands	18
Single Stepping Menu Commands	18
Restart Program Menu Command	18
Memory Menu	18
Inspect Memory Menu Command	19
Fill Memory Menu Command	19



- Search Memory Menu Command 19
- Compare Memory Menu Command 19
- Move Memory Menu Command 20
- Read Memory from File Menu Command 20
- Write Memory to File Menu Command 20
- Empty Cache Menu Command 21
- Breakpoint Menu 21
 - Toggle Breakpoint Menu Command 21
 - Set Breakpoint Menu Command 21
 - Delete Breakpoint Menu Command 21
 - Delete All Breakpoints Menu Command 21
 - ICD Breakpoint 21
- Watch Menu 21
 - Add Watch Menu Command 22
 - Change Watches Menu Command 22
 - Delete Watch Menu Command 22
 - Delete All Watches Menu Command 22
- Windows Menu 22
 - Close Window Menu Command 22
- Help Menu 22
 - Help Contents Menu Command 22
 - About Menu Command 23
 - Status Window Menu Command 23
 - Verbose Status Toggle Menu Command 23
 - Menu Shortcut Keys 23
- CHAPTER 5 Windows 25**
 - Status Window 25
 - Symbols Window 25
 - Data Window 25
 - Code Window 25
 - Breakpoint Window 25
 - Modules Window 26
 - Watch Window 26
 - Locals Window 26
 - Inspector Window 26
 - File Window 26
 - Register Window 26
 - Stack Window 27
- CHAPTER 6 Debug File Format 29**
 - Introduction 29
 - Creating the Symbol File 29
 - Basic Information 29
 - Basic File Structure 29
 - File Header Structure 29
 - Module Information 30
 - Section Information 30
 - Line Record Information 30
 - Symbol Record Information 30
 - Global Symbol Information 30
 - Global String Table 30
 - Auxiliary Record Table 30
 - Source File Record Information 31
 - Auxiliary Record Table Format 33



The Western Design Center, Inc.

September 2005

Simulator/Debugger v3.49

Common Operations	33
CHAPTER 7 Technical Notes	35
Interrupt Debugging Strategy	35
INDEX	37



THIS PAGE LEFT INTENTIONALLY BLANK



CHAPTER 1 Overview

Welcome to the Wonderful World of WDCDB

The WDCDB program is a simulator/debugger for the W65C816S and W65C02 microprocessors. It can operate either as a simulator with a full processor interpreter or through a parallel port as a remote debugger. It is designed to operate with the WDC C and Assembly Language Development Systems and can operate with full source mode display, raw disassembly, or a mixture of the two.

The real power of the **WDCDB** debugger can be most fully realized when used with programs written in the C language. The debugger gives full source level viewing and stepping with visible breakpoints. In addition, three special window types provide detailed information on variables, data structures and program flow.

However, it is hard to describe the power of debugging assembly language programs at the source level. It has to be experienced to be fully appreciated.

This manual has been organized into two main sections. The first section gives a detailed description on how to prepare programs for debugging, how to run and step through programs, set breakpoints and watchpoints and how to examine and modify memory. The second section is a reference and describes each individual menu command and each type of window.



THIS PAGE LEFT INTENTIONALLY BLANK



CHAPTER 2 Before Running WDCDB

Before running WDCDB, the program files must be compiled and/or assembled and linked with the appropriate options to generate a valid symbol file. Secondly, a properly configured .INI file must be created. The following topics address these issues.

C Language Program Debugging

To debug C language programs, they must be compiled and linked. When using the WDC C compiler, **WDCxxCC**, there are two things that need to be done to obtain optimum results. First, the program **must** be compiled with the **-bs** option. This option tells the compiler to generate special source level debugging information that will be passed through the assembler to the linker and eventually to the symbol file.

Secondly, until the program is close to being finished, refrain from using the peephole optimizer. The reason for this is that some of the operations of the optimizer cause the assembly language code to be rearranged and will no longer be synchronized with the source statements. Attempting to debug an optimized program at the source level can be confusing and frustrating. The best course is to debug the program with optimization disabled and then if there are problems after optimizing, run the debugger in the mixed source and assembly mode as this will provide the source context, but will allow detailed examination of the actual program flow.

Note that the information placed in the assembly language output file is parsed by the assembler and placed in the object file. If the assembler is invoked with the **-g** option, it will also place its own source level information in the object file which will conflict with the information derived from the original C source file. So, if you compile to assembly language and then invoke the assembler manually, be sure to **not** use the **-g** option on the assembler. If you let the compiler invoke the assembler automatically, this will not be a problem.

Assembly Language Program Debugging

To debug assembly language programs using the original source assembly language file, all that is needed is to assemble the file with the WDC assembler, **WDCxxAS**, using the **-g** option. This tells the assembler to generate source level information and place it in the object module. The file, *filename.bin*, is created by the **-g** option and needs to be included in the WDCDB>.INI file. The linker will collect this information and place it into the symbol file. Be sure not to use the **-g** option on assembly language files generated by the C compiler since the compiler generates its own source level information.

Linking for Debugging

The final step in preparing to debug a program is to generate the final binary image and symbol file. The WDC linker, **WDCLN**, performs this step. Source level debugging information is placed in the object modules by the compiler and the assembler. The linker collects all this information and creates a separate file with a .SYM suffix. To generate the symbol file, the **-g** option must be specified to the linker. In addition, the output format must be the proprietary format. This output format is the default, however it can be explicitly specified by using the **-hz** option.

The WDCDB.INI File

The WDCDB debugger looks for and stores all of its state information in the **WDCDB.INI** file. The debugger looks for the file **only** in the current directory when the debugger is launched. If the debugger was launched from a DOS shell, the search directory is the current working directory of that shell and **not** the directory in which the **WDCDB.EXE** file itself is located. If the debugger is launched from a Windows file manager, the icon associated with the debugger will contain an explicit reference to a working directory. This reference will be used to find the **WDCDB.INI** file.



The Western Design Center, Inc.

September 2005

Simulator/Debugger v3.49

For example, if you are working on two projects in different directories with different **WDCDB.INI** files and starting the debugger from Windows, the best course of action is to create two separate icons and set the working directory of each to one of the two project directories.

Most information in the file is generated and updated automatically by the debugger. The remaining information can be modified by using the **Edit Startup Options** and **Edit Runtime Options** menu command to open the options requester. When starting a new project it is useful to copy the **WDCDB.INI** file from a previous project to the new project's directory. Otherwise, if no **WDCDB.INI** file is found, the debugger will create a new file after gathering the necessary information.



CHAPTER 3 Running WDCDB

WDCDB can be run either from a DOS prompt or from windows. The following topics discuss what happens when WDCDB first starts up and what options are available for altering WDCDB's startup behavior.

Starting WDCDB

When WDCDB first starts, it loads a number of initialization values from the **WDCDB.INI** file. Among other things, these options specify whether simulation is required, the position, size and other attributes of windows that are to be opened, and the source path and file name of the application being debugged. After loading the initialization values, the command line is parsed to check for any startup options that might override the default options. If any of these options require an action, it will be taken.

Startup Options

The following startup options are currently available:

- a This option overrides the option to set a temporary breakpoint at main and begin execution till it is reached. Instead, the Program Counter is set to the program start address.
- c This options sets a new command line.
- d Download only mode. The target file is downloaded, after which the simulator/debugger exits.
- r Download/run mode. The target file is downloaded, after which the target program is started and the debugger exits leaving the program running.
- s Force simulation mode

Loading a Program

There are three ways to load a program into WDCDB. The first method is to specify the name of the program when the program is started either by running the program from the command line or having the program name specified as part of the Windows program manager icon.

For example, to debug the program **tst.bin**, one could simply type:

```
WDCDB tst
```

Since the default extension is **.bin**, the debugger would locate the file **tst.bin** and load it into memory. If the simulator option was set in the WDCDB.INI file, it would be loaded into the simulator memory; otherwise, it would be loaded via the parallel port into the target memory. The second method is to specify the name of the program in the WDCDB.INI file using the **Program=** entry. This entry will be used if no program name was specified when starting the program.

In this case, if the line in the WDCDB.INI file was:

```
Program=tst
```

the, just typing:

```
WDCDB
```

would load the program **tst.bin**.

The last method is to start the debugger with no program specified and then use the menu option **File/Load Program**. This opens up a file requester that allows the user to browse the file system and select the file to be loaded and debugged.



Loading The Symbol Table

When a program is loaded, the debugger automatically searches for a symbol file that has the same root name, but with an extension of **.SYM**. This file contains all the symbolic and line number information used by the debugger. Alternatively, the **File/Load Symbols** command can be executed. This will open a browser window that will allow the user to select a file that will be loaded as the symbol file. Only one symbol file may be loaded. Once a valid symbol file has been loaded, the **File/Load Symbols** menu command is disabled.

Where to Find the Source Files

The final piece that the debugger needs is the path to the source files themselves. By default, the debugger will search the current directory for source files. If the **Source Path=** variable has been initialized in the WDCDB.INI file, the debugger will first search in the locations specified by the path and if not found, will check the current directory.

For example, if the line:

```
Source Path=C:\TEST
```

were in the WDCDB.INI file, then if the debugger were to try and open the file **MAIN.C**, it would first try **C:\TEST\MAIN.C** and if not found, would then try **.\MAIN.C** before giving up.

Running Programs

There are several ways of running a loaded program. The simplest is to simply start execution by using the **Run/Go** command or it's keyboard shortcut **F9**. The program will continue to run until a breakpoint is encountered or the program is interrupted by the PC. If the simulator is being used, the program will also be stopped if an illegal memory access occurs.

Stopping the Program

When a program is executing, it can be interrupted by pressing the escape key. This will always work when simulating. If not simulating, pressing the escape bar will generate an NMI if the appropriate cable connections have been made. The NMI will interrupt the program and return control to the monitor.

Single Stepping the Program

The program can also be executed either one instruction or one source statement at a time. There are two stepping commands, **Run/Step Into** and **Run/Step Over** with their corresponding keyboard shortcuts, **F7** and **F8**. These functions execute either one source statement if the active code window is in source mode otherwise it will execute one machine instruction. The difference occurs on statements that make a call to another function. The **Run/Step Into** command will single step to the first statement of the new function. The **Run/Step Over** command will execute the function being called at full speed and will stop on the statement following the statement making the function call.

There is one special case that occurs when using the **Run/Step Into** command in source mode. If the statement about to be executed calls a function for which there is no source information available, by default, the statement will be stepped over not into. This is the default, since the compiler generates a number of calls to library functions that normally would not be stepped into.

However, the default behavior can be modified by setting the **Step In Library=** flag in the WDCDB.INI file. If this variable is set, the debugger will step into functions for which no source is available. This flag can be changed by using the **Edit/Runtime Options** menu command.



Breakpoints

Breakpoints are used to stop execution at key points in the program. There are currently four types of breakpoints supported by WDCDB: **one-shot**, **enabled**, **disabled**, and **temporary**. Normally a breakpoint is set at a particular instruction address. The debugger saves the actual instruction at that address and replaces it with either the **COP** or **WDM** instruction. The **WDM** instruction is used when running in simulation mode, while the **COP** instruction is used otherwise. Breakpoints appear transparent to the user and are only actually placed in memory immediately before execution of code.

Breakpoints are displayed in two places. The **Breakpoint Window** will display a list of all breakpoints except temporary ones. Each breakpoint will show the address, skip count and status. The **skip count** is used to create a breakpoint that will be skipped some number of times before actually interrupting the program. Breakpoints are also displayed in the **Code Window**. When an assembly language line or source line is displayed with a breakpoint, the line is displayed in a color different from the normal background color. The following table shows the type of breakpoint and the corresponding color.

Enabled	Red highlighted line in the code window.
Disabled	Grey highlighted line in the code window.
One-shot	Yellow highlighted line in the code window.

Temporary breakpoints are not displayed in the code window.

Standard Enabled Breakpoints: Enabled breakpoints are the standard breakpoint. By default, they have a skip count of zero and will return control to the debugger each time they are encountered.

Disabled Breakpoints: Disabled breakpoints provide a mechanism for keeping a set of breakpoint without having all in the breakpoint table without having all of them active. Disabled breakpoints can be changed to enable or one-shot through the Breakpoint Window.

One Shot Breakpoint: One-shot breakpoints are special breakpoints that when encountered, return control to the debugger and then automatically remove themselves.

Note: When a breakpoint is set inside an IRQ function, one should press F9/RUN before the "RTI" instruction for the Debugger to behave properly. If not, the Debugger will go into a "RUN" mode, but not hit the breakpoint in the IRQ instruction.

Temporary Breakpoint

There is only one temporary breakpoint. The temporary breakpoint is modified in response to the **Run/Go To ...** commands. Once set, the temporary breakpoint remains in effect until either encountered or changed to a different location. For example, consider the case where the command **Run/Go To Here** is executed, but before the Here location is reached, a different breakpoint is encountered. If the **Run/Go** command is executed, the temporary breakpoint will remain in effect until the Here location is reached.

Watches

While the debugger is running, it is often useful to examine the values of important variables and memory locations. The **Watch Window** provides a useful mechanism for doing this. The **Watch Window** contains a set of watch expressions. These expressions are evaluated and the result is displayed. Results can be displayed in a number of different formats and sizes. Expressions based on C typing will automatically use the correct format and size. Watch expressions that do not evaluate to a legal value will display a set of dashed lines. If a watch expression makes reference to a dynamically allocated symbol such as an automatic variable in a C function, it will display a set of dashed lines as well, unless that variable is in the current active scope.



THIS PAGE LEFT INTENTIONALLY BLANK



CHAPTER 4 Menus

Many of the debugger's features and commands are available through the menu system. Note that many of the menus have keyboard shortcuts associated with them. At the end of this set of topics there is a topic that summarizes the keyboard shortcuts associated with the menus.

File Menu

The File Menu contains functions that relate to the loading of programs, symbol tables and the saving and loading of watches and breakpoints.

Load and Reload Program and Symbols

These commands are used to load a program into the debugger. The **Load Program** command opens a file requester that can be used to select the file to load. When a file is loaded using this menu command, the symbol table is automatically searched for and loaded as well. The **Reload Program** command is used when a program has previously been loaded using one of the three methods of loading a program. This menu command will reload the code and data portion of the program being debugged. The symbol table will not be opened since it usually has already been loaded. The **Load Symbols** command opens a file requester to locate a symbol file that will be read to load symbols into the debugger. This function is useful for cases where the debugger is restarted and takes control of a program already loaded and running. If a symbol table has already been read, this function is disabled.

See Also: loading a program

Save and Load Desktop

The **Load Desktop** menu command resets all windows to the positions and sizes specified in the WDCDB.INI file. These positions are either automatically saved when the program is exited or can be specifically saved by using the **Save Desktop** menu command. Note that certain windows including **File Windows** and **Inspector Windows** are not saved.

Save Breaks and Watches

The **Save Breaks** menu command is used to save the current set of breakpoints in the WDCDB.INI file. These breakpoints are loaded automatically when the debugger is started. Similarly, the **Save Watches** command is used to save the current set of watches in the WDCDB.INI file. These watches are also loaded when the debugger is started. To clear either the watches and/or the breakpoints that are automatically loaded, first delete all watches and/or breakpoints and then execute the **Save Watches** and/or **Save Breakpoints** command.

Exit WDCDB

This menu command terminates the debugger. If any of the auto-save options have been specified, the appropriate information is written to the WDCDB.INI file before exiting.

Edit Menu

The **Edit Menu** functions are primarily used for positioning the cursor in **File** and **Code Windows** and to specify the starting position of **Code** and **Data** windows.



Text Search

The **Search** menu command is used to locate text strings in **File Windows** and **Code Windows**. After locating the target string, the window will be scrolled so that the target string is visible and the cursor will be placed at the start of the target string. When the command is executed, a requester is opened asking for the target string. After the string is entered, if a match is found, the cursor will be moved to the target text in the current window. Note that the text search is case independent. Once a search has been performed, additional searches may be performed using the same target string. These searches can be either forward from the current cursor position using the **Search Next** menu command or backward from the current cursor position using the **Search Prev** menu command. Note that searches terminate at the beginning or end of the file and do not wrap.

Code and Data Position

The **Code Position** menu command is used to position the top-most **Code Window**. When executed, a requester is opened asking for an address where the window should be positioned. The requester also allows the code display mode of the window to be specified using a set of three radio buttons. Likewise, the **Data Position** command will position the top-most **Data Window** to the desired address. The requester also allows the data display mode of the window to be specified using a set of radio buttons. Note that the position specified can be any regular expression accepted by the debugger.

Edit Startup Options

The **Startup Options** menu command is used to edit the WDCDB.ini file. The lines that need to be filled in are "Source Path:", "Program:", "Command Line:", "About time out(MS):", and "CPU type:". The three check boxes are "Simulator enabled", "Auto-execute till main", and "Maximize main window", and can be selected or unselected depending on user preference.

"Source Path:" is usually the current directory (i.e., ".") and any higher directory separated by a space (i.e., ". ../higher directory/").

"Program:" is the file name being tested with a ".bin" suffix. This is provided by the assembler or compiler.

"Command Line:", is usually blank.

"About time out(ms):" is enabled when the simulator is off and we are running on a target CPU.

"CPU type:" is either 65C816 or 65C02.

There are "SAVE" and "CANCEL" buttons at the bottom of the **Startup Options** menu. Select save to modify the WDCDB.ini file.

The new settings will take effect the next time the WDCDB.EXE is run.

Edit Runtime Options

The **Runtime Options** menu command is used to edit the WDCDB.ini file. The "Default Radix" box allows the user to select either decimal or hexadecimal representation as the default for variable data displayed. The "Tab Size" box lets the user convert the embedded tab characters in the source file to be assigned the number of spaces that will replace the tab character. The "Options" box has several check boxes that can be selected or unselected, depending on the user preference as indicated below.

"Show addresses with source lines"

"Show line numbers"

"Step into library functions"

"Auto-save desktop on exit"

"Auto-save watches on exit"

"Auto-save breakpoints on exit"

"Update windows during animate"

The new settings will take effect the next time the WDCDB.EXE is run.



View Menu

The **View Menu** is used to open different types of windows to view information about the program being debugged. The types of windows are listed below:

View Code Menu Command

This command will open a new **Code Window**. First a requester will be opened that allows the initial position and code display mode to be specified. Then the window will be opened using the specified position and display mode.

View Data Menu Command

This command will open a new **Data Window**. First a requester will be opened that allows the initial position and data display mode to be specified. Then the window will be opened using the specified position and display mode.

View File Menu Command

This command provides a general purpose text file viewer. When executed, a file requester will be opened that allows the file to be selected. More than one **File Window** can be opened.

View Registers Menu Command

This command opens the **Register Window** if not already opened. If already open, the window is made the active window. Only a single **Register Window** can be opened.

View Stack Menu Command

This command opens the **Stack Window** if not already opened. If already open, the window is made the active window. Only a single **Stack Window** can be opened.

View Breakpoints Menu Command

This command opens the **Breakpoint Window** if not already opened. If already open, the window is made the active window. Only a single **Breakpoint Window** can be opened.

View Watches Menu Command

This command opens the **Watch Window** if not already opened. If already open, the window is made the active window. Only a single **Watch Window** can be opened.

View Symbols Menu Command

This command opens the **Symbols Window** if not already opened. If already open, the window is made the active window. Only a single **Symbols Window** can be opened.

View Modules Menu Command

This command opens the **Modules Window** if not already opened. If already open, the window is made the active window. Only a single **Modules Window** can be opened.



View Locals Menu Command

This command opens the **Locals Window** if not already opened. If already open, the window is made the active window. Only a single **Locals Window** can be opened.

Run Menu

The **Run Menu** is used to control execution of the program being debugged. The operation of each menu item is discussed.

Go Menu Commands

The **Go** menu commands all cause the processor to begin full speed execution. The different commands provide options to control where the processor begins execution and where it ends execution. The unqualified **Go** menu command causes the processor to begin execution at the current program counter address and continue till stopped by a breakpoint or an interruption from the debugger. The **Go To Here** menu command sets a temporary breakpoint on the line currently under the cursor. Thus, to execute to a specific line without setting a permanent breakpoint, simply position the cursor on that line and execute the **Go To Here** menu command. The **Go To ...** menu command is similar to the **Go To Here** menu command except that a requester is opened and a temporary breakpoint is set at the address specified in the requester. The **Go From Here** menu command changes the program counter to match the address of the line currently under the cursor. Then, execution begins from that address. The **Go From ...** menu command opens a requester and allows the starting address of the processor to be explicitly specified by typing an expression into the requester. Processor execution begins from the specified address.

Single Stepping Menu Commands

There are several different single stepping commands. These commands cause the processor to execute a single line of code. The **Step** commands cause the processor to execute a single line of code and then return control to the debugger. There are two basic forms of this command called **Step Into** and **Step Over**. A **Step Into** command executes the current line of code and 'steps into' any functions that are called by the line being executed. The **Step Over** command executes the current line of code and 'steps over' any functions that are called by the line being executed. The functions are executed at full speed and single stepping continues after the function returns. There is one exception to the **Step Into** command. If there is no source file or source information for the function being called, it is assumed that the function is a library function and an implicit **Step Over** is performed. There are two ways to change this behavior. First, the code display mode can be switched into mixed or raw assembly mode and the **Step Into** will operate as expected. Second, the **Step Into Library** variable in the WDCDB.INI file can be set that will cause all **Step Into** functions to always step into the function. There are two additional commands **Animate Into** and **Animate Over** that will continuously execute **Step Into** and **Step Over** commands respectively. The animated stepping will continue until a breakpoint is reached or the animation is stopped by pressing the space bar. The last command, **Skip**, also operates on single lines of code. However, this command simply changes the program counter of the processor to point to the next line of code without executing it.

Restart Program Menu Command

This command resets the program counter to the start address of the program and resets any other settings. The processor is not started. Note that any initialized data not copied from ROM during startup may contain corrupted data.

Memory Menu

The **Memory Menu** contains some general utility commands to inspect and manipulate memory. The operation of each menu item is discussed.



Inspect Memory Menu Command

The Inspect Memory menu command is used to examine C typed variables especially, structures, unions and arrays. When executed, a requester is opened that allows an expression to be entered. The expression is parsed and the result is displayed in an Inspector Window.

Fill Memory Menu Command

The **Memory Fill** menu command is used to fill blocks of memory with constant values. When executed, a requester is opened. The user then fills in the start address, end address and a sequence of up to 10 bytes, words, or long words with which to fill memory. The size of the sequence items is specified using the radio buttons in the requester.

For example, to fill a block of memory from \$100 to \$200 with alternating long words of \$ffffff and \$00000000:

```
Start: $100
      End: $200
Fill with: $ffffff $00000000
Size:
  o Byte
  o Word
  x Long
```

When the **Okay** button is pressed, the memory block is filled.

Search Memory Menu Command

The **Memory Search** menu command is used to search a range of memory for a specific sequence. When executed, a requester is opened that allows the user to specify the start and end address of the range of memory to be searched. There is a set of radio buttons that control how the search target is interpreted, either as a sequence of bytes, words or long words. Finally, the target sequence itself can be entered with as many as ten entries.

For example, to search for the word sequence \$1234 \$5678 \$9000 in the memory range from \$100 to \$200, the following setup would be used:

```
Start Address: $100
End Address: $200
Search: $1234 $5678 $9000
Size: o Byte
      x Word
      o Long
```

When the **Okay** button is pressed, the memory is searched. If a match is found, an info requester giving the address will be displayed. If no match is found, an error requester will be displayed.

Compare Memory Menu Command

The **Memory Compare** menu command is used to compare two blocks of memory. When executed, a requester is opened that allows the user to specify the start and end address of the first block and the start address of the second block. The length of the second block is automatically set to the length of the first block.



For example, to compare the two blocks of memory at \$100 and \$800 both of size \$80, the following setup would be used:

```
Start: $100
      End: $180
      Compare with: $800
```

When the **Okay** button is pressed, the two memory blocks are compared. If no difference is found, an info box is displayed to that effect. If a difference is found, an info box giving the address of the difference is displayed.

Move Memory Menu Command

The **Memory Move** menu command is used to copy a block of memory from one location to another. When the command is executed, a requester is opened that allows the user to specify the start and end address of the block to be moved and the destination address.

For example, to take the block of memory from \$100 to \$200 and copy it to address \$800, the following setup would be used:

```
Start: $100
      End: $200
      Move to: $800
```

When the **Okay** button is pressed, the first block of memory will be copied to the address specified.

Read Memory from File Menu Command

The **Memory Read** menu command reads data from a disk file and stores it into memory. When the command is executed, a requester is opened that asks for the name of the file and the address to store the data in the file.

For example, to load the file "tstdata.bin" at address \$100, the following setup would be used:

```
Start: $100
      File: tstdata.bin
      Browse
```

When the **Okay** button is pressed, the specified file will be opened and the binary contents of the file will be loaded starting at the specified address. The **Browse** can be pressed to open a file requester that can be used to find the desired file.

Write Memory to File Menu Command

The **Memory Write** menu command will write a range of memory as a binary image to a disk file. When executed, the command will open a requester asking for the start and end address of the block of memory to be written and the name of a file to write the data to.

For example, to write the block of memory from \$100 to \$200, the following setup would be used:

```
Start: $100
      End: $200
      File: data.bin
```

When the **Okay** button is pressed, the file will be opened, the specified range of memory will be written to the file and the file will be closed.



Empty Cache Menu Command

The **Memory Empty Cache** menu command is used to clear the memory cache. This forces the debugger to invalidate all cache buffers and fetch all memory data from the target. The cache is normally only used on hardware systems with areas that are marked as ROM,

Breakpoint Menu

This menu controls the setting of breakpoints. The commands are:

Toggle Breakpoint Menu Command

The **Breakpoint/Toggle** menu command toggles the type of breakpoint currently under the cursor in **Code Windows** and **Breakpoint Windows**. The breakpoints toggle between enabled (red bar), disabled (grey bar), oneshot (yellow bar) and no breakpoint.

Set Breakpoint Menu Command

The **Breakpoint/Set** menu command will set a permanent breakpoint at the current cursor position in an active **Code Window**. In a **Breakpoint Window**, it will open a requester allowing an expression to be entered that will be evaluated to an address. There is also a set of radio buttons that allows the type of breakpoint to be specified. (Enable, Disable, and Oneshot) Finally, a **skip count** can be specified that indicates how many times the breakpoint should be skipped before returning control to the debugger.

Delete Breakpoint Menu Command

The **Breakpoint/Delete** menu command is used to delete a breakpoint. The command has two different modes of execution depending on the current active window and cursor position. If the active window is a **Code Window**, the command will delete the breakpoint under the cursor. If there is no breakpoint on the current cursor line, no action will be taken. If the active window is a **Breakpoint Window**, the command will delete the currently selected breakpoint. If any other window is active, no action is taken.

Delete All Breakpoints Menu Command

The **Breakpoint/Delete All** menu command will clear all breakpoints of all types. Note that breakpoints saved in the WDCDB.INI file will not be cleared unless the **Save Breakpoints** menu command is executed after clearing all breakpoints.

ICD Breakpoint

The ICD breakpoint allows the debugging of a target system. This allows a breakpoint on a READ or WRITE with matching or un-matching data comparison. The ICD is useful in debugging “memory leaks” and unexpected variables being modified other than by a direct “store” instruction. It also helps find pointers or indices that have gone astray.

Watch Menu

The **Watch Menu** implements the creation, deletion and control of the **Watch Window**. Entries in the watch window are used to monitor the status of key variables and memory locations.



Add Watch Menu Command

The **Watch/Add** menu command is used to create a new watch entry in the **Watch Window**. When executed, the command will open a requester that asks for an expression to use for the watch, the size and the type of display desired. The choices for display **size** are Byte, Word and Long. These only affect the Binary, Decimal and Hexadecimal display types. The choices for display **type** are Ascii, Binary, Decimal, Hex, Float and C Type.

If the **Watch Window** is active when the command is given, the new watch will be inserted before the currently selected watch. Otherwise, the new watch will be added to the end of the list of watches.

Change Watches Menu Command

The **Watch/Change** menu command is used to modify watches. It has two different modes of operation. If the current window is the **Watch Window**, then the currently selected watch will be modified. If any other window is active, a requester is opened that asks for either the expression of the watch to change or the number of the watch as listed in the **Watch Window**. Once the watch is selected, a requester is opened that allows either the watch expression or the watch value to be modified.

Delete Watch Menu Command

The **Watch/Delete** menu command is used to delete watches. It has two different modes of operation. If the current window is the **Watch Window**, then the currently selected watch is deleted. If any other window is active, a requester is opened that asks for either the expression of the watch to delete or the number of the watch as listed in the **Watch Window**.

Delete All Watches Menu Command

The **Watch/Delete All** menu command will delete all watches currently in use. Note that watches saved in the WDCDB.INI file will not be cleared unless the **File/Save Watches** menu command is executed after clearing all watches.

Windows Menu

The **Windows Menu** controls the sub-windows of the debugger. At the moment, only one command is active, but there will be more added in future versions.

Close Window Menu Command

The **Window/Close** menu command is used to close the current sub-window of the debugger. This command cannot be used on the main debugger window. Use the **File/Exit** menu command to close the main window.

Help Menu

The **Help Menu** gives access to this help file and gives some information about this program's version.

Help Contents Menu Command

The **Help/Contents** menu command opens the debugger help file and shows the contents page. If this command displays an error, make sure that the **WDCDB.HLP** file is located in the same directory that the debugger itself was executed from.



About Menu Command

The **Help/About** menu command displays a requester that gives the current version of the debugger along with some additional information. Pressing any key or the mouse will close the requester.

Status Window Menu Command

The **Help/Status Window** menu command opens the **Status Window** if closed. Otherwise it will make it the active window. This displays the status of the WINIO.DLL, as well as the download of the “bin” file blocks to the developer board, if attached. It also displays internal testing information.

Notes: You can only have one (1) **Status Window** open at a time!
You can only have one (1) **Symbol Window** open at a time!
You can only have one (1) **Stack Window** open at a time!
You can only have one (1) **Register Window** open at a time!
You can only have one (1) **Breakpoint Window** open at a time!
You can only have one (1) **Modules Window** open at a time!
You can only have one (1) **Watch Window** open at a time!
You can only have one (1) **Locals Window** open at a time!

Verbose Status Toggle Menu Command

The **Help/Verbose Status Toggle** menu command is primarily used for internal testing purposes. It causes additional information to be printed to the status window during certain debugger operations and can be ignored by the user.

Menu Shortcut Keys

The following is a summary of the menu shortcut keys.

Alt+X	File/Exit	Ctl+I	Memory/Inspect
Sh+F3	Edit/Copy	Ctl+F3	Memory/Empty Cache
Sh+F4	Edit/Paste	F2	Breakpoint/Toggle
Ctl+F	Edit/Search	Alt+F2	Breakpoint/Set
Ctl+N	Edit/Search Next	Sh+F2	Breakpoint/Delete
Ctl+U	Edit/Search Prev	Ctl+F2	Breakpoint/Delete All
Alt+C	Edit/Code Position	F1	Watch/Add
Alt+D	Edit/Data Position	Alt+F1	Watch/Change
F9	Run/Go	Sh+F1	Watch/Delete
F4	Run/Go To Here	Ctl+F1	Watch/Delete All
F7	Run/Step Into	Alt+F3	Windows/Close Current
F8	Run/Step Over	Alt+Y	Help/Verbose Status Toggle
Alt+S	Run/Skip		



THIS PAGE LEFT INTENTIONALLY BLANK



CHAPTER 5 Windows

The display is maintained as separate windows within the main WDCDB window. All windows can be opened, closed and moved. Most can be resized as well.

Status Window

The **Status Window** contains information relating to the current state of the debugger. Messages are displayed as programs are loaded and when the verbose status mode is enabled, during any reads or writes of the memory.

Symbols Window

The **Symbols Window** displays a list of all global symbols loaded from the program's symbol file. Each symbol is displayed on a separate line with the address or value of the symbol displayed as well. The symbol table can be displayed in either alphabetical or numerical order. To switch between the two, use the **F3** key. Hitting the enter key on a code symbol will position the topmost **Code Window** to the corresponding address. If the symbol is a data symbol, the topmost **Data Window** will be positioned to the corresponding address.

Data Window

The **Data Window** is used to examine the memory of the target processor. The memory can be displayed in a number of different formats: byte, word, long word, float, and double. In the non-floating point modes, the bytes are also displayed as ASCII characters with non-printing characters replaced by a period. The window can be switched between modes by using the **F3** function key. The window can be positioned by scrolling or by using either the generic **Alt+P** position command or the **Data Window** specific **Alt+D** position command. These commands will open a requester and allow an expression to be entered that will be evaluated and the resulting value will be used as the new window starting position. The requester also has a set of radio buttons for setting the window display mode.

The requester also has a check box that when set will cause the window to be **anchored**. An anchored data window will always reposition to the location specified by the anchor expression. For example to keep track of values placed on the stack, a data window can be anchored to the expression `+1` that will force the window to always be positioned to show the top value on the stack. Any valid expression can be used to anchor a **Data Window**. Memory in a **Data Window** can be modified simply by positioning the cursor to the memory to be changed and typing the new values. If the window is in one of the floating point modes, positioning the cursor and pressing any key will open a requestor allowing the new value to be entered. There can be as many **Data Windows** open as needed.

Code Window

The **Code Window** displays the program code being debugged. The program can be displayed in source mode, mixed assembly and source, or just plain assembly code. Breakpoints are displayed as a different color background. The current program counter is shown by changing the color of the line. If a line is marked by both the program counter and a breakpoint, the program counter indicator occurs on the first part of the line, while the breakpoint indicator marks the remainder of the line.

Breakpoint Window

The **Breakpoint Window** shows the current set of breakpoints. Each breakpoint is displayed on its own line and shows the address of the breakpoint, the current skip count, the type of breakpoint and a symbol name if one is associated with the address. Within the window, breakpoints can be added and removed by using the **Insert** and **Delete** keys or by using the menu functions in the **Breakpoint Menu**. In addition, hitting the **Return** key will position the **Code Window** to the address of the breakpoint.



Modules Window

The **Modules Window** displays a list of all files that were used when creating the target program. Selecting a file name and hitting return or double clicking will position the top-most code window to the beginning of that file.

Watch Window

The **Watch Window** shows the current value of any expressions that have been placed in the **Watch Window**. These values are updated each time the target processor stops executing and control returns to the debugger.

Locals Window

The **Locals Window** shows the names, types and values of local variables and arguments to the current C function. Each line shows the variable name, the address, the value in hex and decimal and finally, the type. If the **Return** key is pressed, the selected variable will become the target of an **Inspect Window**. If the **Space** key is pressed, a requester will be opened showing the current value of the local variable and allowing the value to be changed.

In the Locals window, the:

1st column is the variable name

2nd column is the address of the variable

3rd column is the data in hex for this variable or a pointer address

4th column is the data in decimal

5th column is the typedef of the variable

Inspector Window

The top of the window shows the expression and the address associated with the expression. The remainder of the window shows the value of the expression. If the expression represents a scalar quantity, a single line will display the type and value. Hitting return on a member of an array or structure will open a new inspect window using the current symbol to determine the new expression.

File Window

The **File Window** is used to browse through text files, usually other than source code files. Files are displayed and can be scrolled and searched through.

Register Window

The register window displays the current state of the CPU registers, the current cycle counter and the source and destination of the current active instruction.

Definitions are as follows:

A:	Accum
X:	X Register
Y:	Y Register
PB:	Program Bank Register
PC:	Program Counter
DB:	Data Bank
DP:	Direct Page



SP: Stack Pointer
SR: Status Register
ENVMXDIZC: (If the letter is capital, then the bit is a "1")
S:
D:
C: Current Cycle Counter

Stack Window

The Stack Window displays the calling sequence used to reach the current function if that calling sequence consists of C functions. Normally, when a C function is called, it saves information on the stack from the previous function. This information allows the debugger to examine and parse the information on the stack and create a list of functions that have been called in the order that they were called. In addition, each function is displayed with a list of the parameters associated with that function.

To open a **Stack Window**, use the menu command **View/Stack**. The **Stack Window** will show the calling sequence with the current function at the top of the window. It is possible for no functions to be displayed in the window. This can be caused by several conditions. The two most common causes are that the current function is not a C function or that the function preamble code has not yet been executed. Besides showing a trace of how the current function was reached, the **Stack** window also allows more detailed viewing of the calling sequence. Selecting a particular function from the list and either pressing the **Return** key or double clicking with the mouse will position the code window to the line of code that made the call associated with that entry.



THIS PAGE LEFT INTENTIONALLY BLANK



CHAPTER 6 Debug File Format

Introduction

This document describes the WDC symbolic debugging format. The information is provided as a service and WDC reserves the right to make changes as are deemed necessary.

Creating the Symbol File

The symbol file is created by the linker and is used during debugging to properly locate and interpret the program. The symbol file can contain both symbolic and source-level debugging information. Specifying the ``-g'` and ``-sz'` options to the linker generates the information that is placed in a file with a `.sym` extension.

Basic Information

There are two files that provide additional information. The header file `'obj816.h'` (located in `WDC_SDS\ZFORMAT`) contains definitions of all the structures and constants. The source file `'zsym.c'` (located in `WDC_SDS\ZFORMAT`) is a program that will display the information in a symbol file. It can be used as an example of how to read and parse a symbol file. All words and long words are written with the low byte first.

To view the `.SYM` file, run the program `WDCSYM.EXE` located in `WDC_SDS_BIN`. Please refer to Chapter 10 of the Assembler/Linker manual for more information on how to use `WDCSYM.EXE`.

Basic File Structure

The basic structure of the file is outlined as follows:

- File Header
- Module 1 Information
 - Section 1 Information
 - ...
 - Section N Information
- Line Record Information
- Symbol Record Information
- Module 2 Information
- ...
- Module N Information
- ...
- Global Symbol Records
- String Table
- Auxiliary Record Table
- Source File Information
- End of file

File Header Structure

The symbol file begins with a header structure that contains

Signature word	\$2345	Number of auxiliary table entries	word
Version	word	String table offset	long word
Number of sections	word	Size of string table	long word
Symbol records offset	long word	Source files offset	long word
Number of symbol records	word	Number of source files	word
Auxiliary table offset	long word	Number of modules	word



Module Information

Each module record contains a long word string index giving the offset to the module name and a word specifying the number of sections.

Section Information

Each section record within a module record contains a single word specifying the section number and a long word specifying the starting address of the section. This is followed by a long word specifying the size of the section and a word specifying the number of line records in this module.

Line Record Information

Line records are only included if code is generated on that line. Each line record consists of:

- A long word specifying the address of the current line.
- A word specifying the source line number.
- A word specifying the number of bytes associated with this line.
- A byte specifying the source file number of this line record.
- A byte specifying the LONGA/LONGI status at the beginning of the current line.

Symbol Record Information

Each module contains a number of symbol records. The number is specified by a word that immediately follows the Line Record Information. The definitions of the fields in a symbol record vary depending on the symbol class. See below.

The structure for each symbol record is:

- A long word index.
- A long word value.
- A byte class.
- A byte flags.
- A word auxiliary index.

Global Symbol Information

Following the last module is the global information for the program. For each global symbol, there is a symbol record as defined above.

Global String Table

All modules use the global string table. The linker collects all strings and concatenates them into one large table. Multiple occurrences of a string are eliminated to save space.

Auxiliary Record Table

Following the string table, is the auxiliary record table. This table contains information collected by the linker from all modules. The auxiliary information includes C language typing information.



Source File Record Information

Then for each source file there is a long word string index to the file name and a word specifying the number of physical lines in the source file.

The following is a description by class type.

strindex ---- C_FILE ---- index of next file

This record indicates a source file. The strindex is the name of the file and the aux index is the index within this symbol record table of the next C_FILE record type. The last file record will have an aux index of 0.

linNum address C_BLOCK ---- index of eblock

This record indicates the start of a code block. Usually this is generated in C by a '{' character. The linNum is the source file line number. The address is the code address associated with the first instruction of the block. The aux index is the index of the end of block symbol record.

linNum address C_EBLOCK ---- index of start block

Paired with C_BLOCK, this record indicates the end of a nested block. The aux index points back to the associated start of block.

strindex size C_STRTAG ---- serial

This record defines the name, size and serial number associated with a particular structure. The strindex is an offset into the global string table with the name of the structure definition. If the struct is unnamed, a name of the form 'fakeN_' is automatically generated. The size is the size of the structure in bits. The serial number is an internal value used by the linker to prevent placing information on unreferenced structures into the symbol file.

strindex offset C_MOS ---- typIndex

This record defines the attributes of a member of a structure. The strindex is the offset of the name of the member in the global string table. The offset is the offset in bits of the member. The typIndex is the index into the global auxiliary records table that defines the type of the member.

strindex offset C_FIELD fldlen typIndex

This record defines the attributes of a bit field member of a structure or union. The strindex is the offset of the name of the bit field in the global string table. The offset is the offset in bits of the bit field. The typIndex is the index into the global auxiliary records table that defines the type of the bit field. The flags field defines the field length.

---- ---- C_EOS ---- index of next tag

This record defines the end of a structure, union or enum. The aux index is the index of the next structure, union or enum.

strindex size C_UNTAG ---- serial

This record defines the name, size and serial number associated with a particular union. The strindex is an offset into the global string table with the name of the union definition. If the union is unnamed, a name of the form 'fakeN_' is automatically generated. The size is the size of the union in bits. The serial number is an internal value used by the linker to prevent placing information on unreferenced unions into the symbol file.



strindex	offset	C_MOU	----	typIndex
----------	--------	-------	------	----------

This record defines the attributes of a member of a union. The strindex is the offset of the name of the member in the global string table. The typIndex is the index into the global auxiliary records table that defines the type of the member.

strindex	size	C_ENTAG	----	serial
----------	------	---------	------	--------

This record defines the name, size and serial number associated with a particular enum. The strindex is an offset into the global string table with the name of the enum definition. If the enum is unnamed, a name of the form 'fakeN_' is automatically generated. The size is the size of the enum in bits. The serial number is an internal value used by the linker to prevent placing information on unreferenced enums into the symbol file.

linNum	addr	C_FUNC	----	index of efunc
--------	------	--------	------	----------------

This record defines the beginning of a function. The linNum is the physical source line associated with the function. The addr is the program counter location associated with the function. The aux index is the index of the end of function symbol record.

localOffset	argOffset	C_FRAME	----	----
-------------	-----------	---------	------	------

This record always follows a C_FUNC record and describes the offset from the current DP setting to the start of the locals and arg variable areas. This will vary depending on the memory model being used and the number of temporary variables used by the compiler.

linNum	addr	C_EFUNC	----	index of next func
--------	------	---------	------	--------------------

This record defines the end of a function. The linNum is the physical source line associated with the end of the function. The addr is the address of the next byte after the last function code byte. The aux index is the index of the next function start record or zero if there are no more.

strindex	value	C_EXT	funcflg	typIndex
----------	-------	-------	---------	----------

This record defines an external symbol and is found in the global symbol record table. The strindex is the offset of the symbol name in the global string table. The value is the address of the symbol. The funcflg is a 1 if the symbol denotes a function, otherwise it is zero to denote a data label. The typIndex is an offset into the global auxiliary record table that gives information on the type of the symbol.

strindex	localOffset	C_AUTO	----	typindex
----------	-------------	--------	------	----------

strindex	argOffset	C_ARG	----	typIndex
----------	-----------	-------	------	----------

These records define local and argument variables. The strindex is the offset into the global string table of the symbol name. The typIndex is the index into the auxiliary record table to the type definition for the variable. The localOffset and argOffset are the offset of the variable from the beginning of the local and arg area respectively. The local and arg areas are defined by the C_FRAME record associated with the current function. Thus the address of a local variable will be the current DP register value + the C_FRAME localOffset value + the C_AUTO localOffset value.



Auxiliary Record Table Format

The auxiliary record table is a table of long words. Each long word is interpreted as follows:

The low four bits of each type indicates the base type as defined in the enum:

```
enum { T_NULL, T_VOID, T_SCHAR, T_CHAR, T_SHORT, T_INT16, T_INT32,
       T_LONG, T_FLOAT, T_DOUBLE, T_STRUCT, T_UNION,
       T_ENUM, T_LDOUBLE, T_UCHAR, T_USHORT, T_UINT16, T_UINT32,
       T_ULONG };
```

The remaining 28 bits specify derived types of the base type. The bits are arranged as nine sets of three bits each. Each three-bit set defines one of the following derived types:

```
enum { DT_NON, DT_PTR, DT_FCN, DT_ARY, DT_FPTR, DT_FFCN };
```

A derived type of DT_NON signals the end of the derived types.

For example if the base type is T_SHORT and the first derived type is DT_PTR and the second derived type is DT_ARY, then the value would be:

```
DT_SHORT | ( DT_PTR << 4 ) | ( DT_ARY << ( 4+3 ) )
4 | ( 1 << 4 ) | ( 3 << 7 )
0x194
```

Certain base and derived types make use of additional following records.

The T_STRUCT, T_UNION and T_ENUM records are followed by a record that contains the module number of the module which first defined the type in the low 16 bits and the record number within the module's record symbol table in the high 16 bits. This allows the definition to occur only once regardless of how many modules might include the same declaration. Similarly, a record containing the dimension for the array follows the DT_ARY derived type. As many dimension records are generated as there are DT_ARY derived types. In the case where the type is an array of structures, the struct record is always first followed by any dimension records.

Common Operations

Finding a function by name:

- Search the global symbol table for C_EXT records with the funcFlag set to 1.
- Do a string compare on the names.

Finding a function by address:

- For each module, scan each section to see if it contains the address.
- Once a module is found, scan the symbol records for the first C_FUNC record.
- Then, for each C_FUNC record:
 - Compare the target address against the start address and the address defined by the C_EFUNC record.
 - The C_EFUNC record can be easily found by using the AUX field of the C_FUNC record.



If the target address falls within the range, the function has been found. To determine the name of the function, scan the global symbol table for a C_EXT record with the funcFlag set and the same VALUE field as the C_FUNC record's VALUE field.

If not in the range, skip to the next C_FUNC record using the AUX field of the C_EFUNC record.

Continue till the AUX field of the C_EFUNC record is zero.

Finding the source line associated with an address:

First, find the function as defined above.

Scan backwards through the symbol record table until a C_FILE record is found.

Then, scan the line record table for this module until a line containing the address is found.



CHAPTER 7 Technical Notes

Interrupt Debugging Strategy

Using the “signal” and the “raise” functions to debug an Interrupt while using the simulator.

Many times we need to test software while the hardware is being designed. It is very difficult to debug Interrupt code without the actual hardware and even with the hardware it can be confusing. When we actually take the interrupt, it needs to run without being slowed down. We can accomplish simple testing of the actual execution in non-real time by using the “signal” and “raise” function calls.

The “signal” function defines the type interrupt we are to assign the interrupt. The “raise” function calls the interrupt type. (The type of interrupt allowed is defined in <signal.h>)

The “signal” and the “raise” functions are also good for trapping “Floating Point Errors” (FPE).

Example code:

```
// command line: WDC816CC -ms -bs -lt -sop.c
```

```
typedef unsigned char uchar;
typedef unsigned char U8;
typedef unsigned short U16;
typedef unsigned long U32;
```

```
#include <signal.h>
```

```
//Gloable variable
int timer_tic = 0;
```

```
void *timer_code(int sig)
{
```

```
    //if (sig == SIGINT)
        timer_tic++;
} // End of the Function Timer_code()
```

```
void main(void)
{
    int error_flag = 0;
    int i;
    int result =0;
```

```
    if (signal(SIGINT, timer_code) != SIG_ERR)
        error_flag = 1;        //Error condition

    for (i=0; i<3; i++) {
        if (raise(SIGINT) ==0)
            result = timer_tic;
        else
            result = timer_tic;
    }
} // End of main()
```



THIS PAGE LEFT INTENTIONALLY BLANK



INDEX

- anchored**, 25
- Assembly Language Program Debugging, 9
- Breakpoint Menu, 21, 25
- Breakpoint Windows**, 21
- bs** option, 9
- C Language Program Debugging, 9
- Code Windows**, 16, 21
- Debug File Format, 29
- Edit Menu**, 15
- Edit Runtime Options**, 10, 16
- Edit Startup Options**, 10, 16
- File Windows**, 16
- File/Load Program**, 11
- File/Load Symbols**, 12
- g** option, 9
- Help Menu**, 22
- ICD Breakpoint, 21
- Memory Menu**, 18
- Menu Shortcut Keys, 23
 - Alt+C Edit/Code Position, 23
 - Alt+D Edit/Data Position, 23
 - Alt+F1 Watch/Change, 23
 - Alt+F2 Breakpoint/Set, 23
 - Alt+F3 Windows/Close Current, 23
 - Alt+X File/Exit, 23
 - Alt+Y Help/Verbose Status Toggle, 23
- Ctl+F Edit/Search, 23
- Ctl+F1 Watch/Delete All, 23
- Ctl+F2 Breakpoint/Delete All, 23
- Ctl+F3 Memory/Empty Cache, 23
- Ctl+I Memory/Inspect, 23
- Ctl+N Edit/Search Next, 23
- Ctl+P Edit/Search Prev, 23
- F1 Watch/Add, 23
- F2 Breakpoint/Toggle, 23
- F4 Run/Go To Here, 23
- F7 Run/Step Into, 23
- F8 Run/Step Over, 23
- F9 Run/Go, 23
- Sh+F1 Watch/Delete, 23
- Sh+F2 Breakpoint/Delete, 23
- Sh+F3 Edit/Copy, 23
- Sh+F4 Edit/Paste, 23
- Run Menu**, 18
- tst.bin**, 11
- View Menu**, 17
- View/Stack.**, 27
- Watch Menu**, 21
- WDCDEBUG**, 7
- WDCDEBUG.INI**, 9
- WDCxxAS**, 9
- WDCxxCC**, 9
- Windows Menu**, 22