

---

# *M62 Development Package Software Manual*

---

---

The M62 Development Package Software Manual was prepared by the technical staff of Innovative Integration, March 1998.

For further assistance contact

Innovative Integration  
5785 Lindero Canyon Road  
Westlake Village, CA 91362  
PHONE:(818) 865-6150  
FAX: (818) 879-1770  
email:techsprt@innovative-dsp.com  
WWW:www.innovative-dsp.com

This document is Copyright 1998 by Innovative Integration. All rights are reserved.

VSS\M62\documents\C\M62sw.fm





---

	List of Tables . . . . .	9
	List of Figures. . . . .	11
<b>CHAPTER 1</b>	<i>Introduction</i> . . . . .	<b>13</b>
	A Note about this Manual. . . . .	14
<b>CHAPTER 2</b>	<i>Installation</i> . . . . .	<b>15</b>
	Host Hardware Requirements. . . . .	15
	Software Installation. . . . .	16
	Configuring the Developer's Package . . . . .	17
	Multiple Board Support . . . . .	17
	Troubleshooting Software Installation Problems . . . . .	19
<b>CHAPTER 3</b>	<i>Integrated Development Environment</i> . . . . .	<b>21</b>
	The Texas Instruments C Compiler Toolset . . . . .	21
	<i>C Compiler Toolset Usage</i> . . . . .	22
	Codewright Editor . . . . .	23
	Code Composer Debugger . . . . .	23
	Support Applets . . . . .	24
	<i>The Terminal Emulator</i> . . . . .	27
	<i>The COFF File Downloader</i> . . . . .	35
	<i>The COFF File Dump Utility</i> . . . . .	39
	<i>The MPO Editor</i> . . . . .	41
	<i>The Viewer Applet</i> . . . . .	43
<b>CHAPTER 4</b>	<i>Developing Target Code</i> . . . . .	<b>59</b>
	Introduction . . . . .	59
	Edit-Compile-Test Cycle using Codewright . . . . .	60
	A Simple Codewright Project. . . . .	61
	<i>Automatic makefile creation</i> . . . . .	65
	<i>Rebuilding a Project</i> . . . . .	68

---

	<i>Running the Target Executable</i> .....	68
	Anatomy of a Target Program .....	69
	<i>Use of Library Code</i> .....	71
	<i>Compiling/Assembling/Linking Outside Codewriter</i> .....	71
	<i>Compiling without a Project</i> .....	72
	<i>Building Libraries</i> .....	73
	The Next Step: Developing Custom Code .....	73
	Edit-Compile-Test Cycle using Code Composer Studio .....	74
	A Simple Studio Project .....	74
	<i>Automatic makefile creation</i> .....	79
	<i>Rebuilding a Project</i> .....	79
	<i>Running the Target Executable</i> .....	79
<b>CHAPTER 5</b>	<i>Developing Host Code</i> .....	<b>83</b>
	Dynamic Link Library .....	84
	<i>Sample Host Programs</i> .....	84
	<i>The XRPT example</i> .....	88
<b>CHAPTER 6</b>	<i>Creating Target Software</i> .....	<b>89</b>
	C Code Development. ....	89
	<i>C Compiler</i> .....	89
	<i>C Library Reference</i> .....	90
	<i>M62 Zuma Toolset Libraries</i> .....	90
	<i>M62 Hardware Interaction</i> .....	94
	<i>Digital Input/Output</i> .....	96
	<i>Timers</i> .....	98
	Example Target Programs for the M62 .....	102
	<i>HELLO</i> .....	102
	<i>TEST</i> .....	103

---

---

<b>CHAPTER 7</b>	<i>Target DSP Peripheral Libraries</i> . . . . .	<b>105</b>
<b>CHAPTER 8</b>	<i>Host DLL Reference</i> . . . . .	<b>113</b>
<b>CHAPTER 9</b>	<i>DOS Environment Requirements</i> . . . . .	<b>119</b>



---

*List of Tables*

TABLE 1.	Generic DLL Function List . . . . .	49
TABLE 2.	Viewer “target” memory selection commands . . . . .	50
TABLE 3.	Viewer “target” memory operators . . . . .	52
TABLE 4.	Target memory display operators . . . . .	53
TABLE 5.	TDUMP mode selector commands . . . . .	53
TABLE 6.	Shorthand memory dump commands . . . . .	54
TABLE 7.	Viewer math and binary operators . . . . .	55
TABLE 8.	Viewer dictionary commands . . . . .	55
TABLE 9.	Viewer system commands . . . . .	56
TABLE 10.	Viewer system commands . . . . .	56
TABLE 11.	Target DLL function shortcuts . . . . .	56
TABLE 12.	Zuma Toolset Source Directories . . . . .	91
TABLE 13.	Zuma Toolset support subdirectories . . . . .	91
TABLE 14.	Texas Instruments Standard Library Functions . . . . .	94
TABLE 15.	M62 External Peripheral Memory Map . . . . .	95
TABLE 16.	Digital I/O Access Memory Location . . . . .	96
TABLE 17.	Digital I/O Latch Configuration . . . . .	97
TABLE 18.	Digital I/O library functions . . . . .	98
TABLE 19.	C Language Timer Functions . . . . .	98
TABLE 20.	STDIO Driver Functions . . . . .	100
TABLE 21.	Generic DLL Function List . . . . .	113
TABLE 22.	Required disk directory structure for II development tools . .	120



---

*List of Figures*

FIGURE 1.	Terminal emulator applet. . . . .	27
FIGURE 2.	Terminal emulator file menu . . . . .	28
FIGURE 3.	Diagnostic received when target DSP is halted. . . . .	29
FIGURE 4.	Terminal emulator plot menu dialog box. . . . .	29
FIGURE 5.	Terminal emulator Window menu. . . . .	32
FIGURE 6.	The Coff File Downloader Applet. . . . .	35
FIGURE 7.	The COFF Dump Utility . . . . .	39
FIGURE 8.	COFF Dump utility output. . . . .	39
FIGURE 9.	The MPO Editor . . . . .	41
FIGURE 10.	MPOEditor "Open" Dialog box. . . . .	42
FIGURE 11.	COFF File list change dialog box. . . . .	42
FIGURE 12.	Viewer main window. . . . .	43
FIGURE 13.	Opening the target DSP. . . . .	44
FIGURE 14.	Variants of Viewers dump command . . . . .	45
FIGURE 15.	Viewers plot window. . . . .	48
FIGURE 16.	Creating a new project in Codewright. . . . .	62
FIGURE 17.	Adding files to a Codewright project . . . . .	63
FIGURE 18.	Codewright Project Window. . . . .	64
FIGURE 19.	Codewright compiler progress in output window . . . . .	65
FIGURE 20.	An example of an auto-generated makefile. . . . .	67
FIGURE 21.	Creating a new project in Studio . . . . .	75
FIGURE 22.	Adding files to a Studio project. . . . .	76
FIGURE 23.	Studio Project Window. . . . .	77
FIGURE 24.	Studio compiler progress in output window . . . . .	79



---

This document describes the Zuma software development environment for Innovative Integration (I.I.) digital signal processor (DSP) cards. The environment comes complete with ANSI compliant C code Compilation, Assembler, Linking, Debugging, and Windows interface software and represents the most complete package available for DSP code creation for Texas Instruments DSP processors.

Each Developer's Package consists of four major features:

- TMS320-based DSP board
- Texas Instruments Floating Point C Compiler/Assembler toolset
- Code Composer JTAG-based hardware-assisted debugger
- Zuma software toolset including:

*Codewright* - integrated code generation environment including world-class editor augmented by a custom DLL to provide makefile script generation and DSP software toolset interface.

*DSP Peripheral Library* - supporting on-board peripherals and DSP functions, with full source code

*Custom 32-bit Windows 95/NT compatible dynamic link library (DLL)* - which utilizes a custom, 32-bit, Ring 0/Kernel-mode device driver for host PC software application development

*Host Support Applets* - for automatic program download, terminal emulation, COFF file dumping and on-board flash programming

*Sample Applications* - showing Host PC as well as target DSP coding techniques

This manual discusses installation issues and includes full documentation on all I.I. software tools (please see the accompanying manuals for specific information on the T.I. toolset or Code Composer software packages). Installation is discussed first, followed by brief introductions to each of the software packages and instructions on their use. General software development issues are presented, and a tutorial on DSP software development, particularly as it relates to the integrated use of the software packages included in this kit, are also discussed. References are given for the peripheral libraries and host DLL packages in the Appendices.

---

### *A Note about this Manual*

Certain typography conventions are used in this manual to indicate user operations, file types, etc., as follows:

- Windows application menu commands are identified and presented as pipe-delimited strings indicating the menu entries which are being discussed. For example, the Load Program menu item under the File menu in the Code Composer package would be named by the following string:

File | Load Program

Computer readable files and keyboard input/output are represented in Courier font, with user input in bold. For example, a program file will be referred to by name as

```
C:\SBC32\TALKER\TALKER.OUT
```

while user input and commands look like

```
ROM MYPROG.OUT
```

---

Installation of the Zuma toolset consists of both hardware and software installation procedures. This document outlines the software installation process, which is detailed in the accompanying *Windows 95 Installation Supplement* or *Windows NT Installation Supplement*, as appropriate. This document details the features of the Innovative Integration software generation tools, applets, utilities and peripheral library functions for the target DSP board. Refer to the target DSP *Hardware Manual* for a discussion of hardware-specific installation and configuration.

This document is intended to augment, not replace, the *Installation Supplement* and the documentation provided with the TI C compiler, Code Composer and other third-party software packages. Refer to the documentation provided with those products for a complete discussion of their features and use.

---

### *Host Hardware Requirements*

The software development tools for the Zuma toolset require an IBM or 100% compatible 486-class or higher machine for proper operation (Pentium-class machines are highly recommended). The host must have at least 16 Mbytes of memory and a CD-ROM (3.5" floppy disks available on request). Windows 95 or NT (referred to

herein simply as *Windows*) is required to run the developer's package software, and is the target operating system for which host software development is supported.

Users wishing to employ innovative DSP boards in systems running under a foreign operating system, such as Unix or OS9 must install the Zuma toolset on a standard Windows platform and use the Code Hammer hardware debugger to umbilical over the the DSP board residing in the foreign system in order effect code development.

Using this method, all of the features and facilities of the Zuma toolset and Code Hammer may be brought to bear in the development of target DSP code and the user need only generate foreign Host-specific code to perform target card communications and COFF object code downloading.

---

### *Software Installation*

Refer to the to the *Development Package Installation Supplement*, supplied with your Developers Package, for specific instruction regarding the installation of the software components of the Zuma Toolset..

In summary, the Zuma Package is installed as a multi-step process, with each component package installed separately using that package's installation instructions. Install the component packages in the following order:

1. DSP Peripheral Library
2. HASP device driver
3. Board-specific device driver (except SBC31/32/54 products)
4. JTAG Device Driver
5. Code Composer debugger software (from GO DSPs disks)
6. TI Code Generation Tools (from TI's CDROM)
7. Codewright Editor (from Premia's CDROM)

After software installation is complete, install the hardware elements below.

8. The JTAG debugger board
9. The target DSP board

Note: the order of installation of these packages is important. Please follow the instructions included in with Code Composer and the TI Compilation Tools for their installation methods. Do not attempt to install the target DSP hardware or the JTAG controller board until the software installation is complete.

---

### *Configuring the Developer's Package*

Once the various tools have been installed, the software packages must be configured for use in the target installation. Code Composer contains a Setup utility which should be used to set up that application: follow the directions in the documentation included with Code Composer to complete its configuration and set up.

Several environment variables must be added/set within your computer's AUTOEXEC.BAT file in order to obtain proper operation of the TI C Compiler/Assembler.

Each of the included support applets creates and uses an application specific initialization file (.INI) to store installation-specific properties. Available switches and configurable settings for each applet are discussed in Section 3, below.

---

### *Multiple Board Support*

Multiple target boards of the same type may be installed in the same system with full development software support (the only exception being the JTAG debugging support under Code Composer for multiple 'C3x targets. Since the modified JTAG standard used on the 'C3x processors does not support multiple processor debugging, Code Composer may be used with only one 'C3x target at a time). Multiple copies of the support applications may be run simultaneously, each communicating with different targets, to provide parallel support for multiple target boards. Follow the instructions below to set up support for more than one target:

1. Install the support software normally per the above instructions.
2. For each target board, make a Windows shortcut icon for each application which must be used simultaneously. For example, if the system has three target boards installed and the user wishes to use the COFF downloader and terminal emulators independently with each board, then make three shortcuts each for the two

applications and label them “COFF Downloader Target 0”, “COFF Downloader Target 1”, etc. To make a shortcut icon, open the “My Computer” desktop icon and open the drive and installation directory where the development tools were installed. Right click on the application for which the shortcut will be made, and select “Create Shortcut”. A new icon will appear in the folder window, labeled “Shortcut to [APPLICATION NAME]”. Rename the icon appropriately by right clicking and selecting the “Rename” menu entry and entering a new board-specific name, such as “COFF Downloader for Board#1”. Optionally, the shortcut may be dragged onto the desktop and the file folder closed to clear display space.

3. Once the shortcut copies have been made for all instances of the application(s) for each target, the shortcuts must be customized to point to their respective target boards. This is accomplished by adding command line switches to the Properties dialog box for each shortcut. Right click on each shortcut and select the “Properties” entry to open the Properties dialog box. Select the “Shortcut” tab and edit the “Target” text box. Add the target number override switch (-t) followed by a space and the target number of the board with which this instance of the program will communicate. To find out each board’s target number, use the FIND utility (described below). For example, if the system has two targets installed, one at target number 0 and one at target number 1, the shortcut for the first board’s COFF downloader would have a “Target” entry of

```
[install directory]\DOWNLOAD.EXE -t 0
```

and the second board’s COFF downloader shortcut would have an entry of

```
[install directory]\DOWNLOAD.EXE -t 1
```

Additional switches may be specified in the “Target” text box to further modify the applications’ individual behaviors. See the support applications’ descriptions below for complete details on the switches available for each application.

4. Note: the command line switches specified in the shortcut properties box act as overrides to the default behavior selected in the configuration utility. Any switches NOT specified in the shortcut properties dialog box will cause the applications to revert to the global configuration selected in the configuration program. For example, if the user selects the Automatic Download feature in the configuration utility and specifies a filename, then all shortcuts created for the COFF downloader will automatically download that file on startup. If one of the shortcuts specifies a -d[FILENAME] switch in its property box, then that

shortcut will download the specified filename on startup, rather than the default application selected in the configuration utility.

---

### *Troubleshooting Software Installation Problems*

If you encounter problems using the development environment, check the following:

- `TERMINAL.INI` and `DOWNLOAD.INI` files. If the support applications do not function correctly, make sure that the configuration has been set up properly for the hardware. Double check the target number override option: if the option is activated, make sure the correct number of target cards have been installed in the host system.
- If the support applications present an error indicating the JTAG interface should be checked, make sure the target is not being held in a non-running state by the Code Composer (or other debugger) software. Select the `Run Free` command in Code Composer, or type `RUNF` if using the Texas Instruments JTAG debugger software.
- If the support applications present an error indicating the target device driver (`.VXD` for '95 or `.SYS` for NT) is not available, make sure that the hardware device drivers have been installed correctly (see the *Hardware Manual* for complete information on installing device drivers).

If components of the Developer's Package still do not operate correctly, contact Innovative Integration for technical support.



# *Integrated Development Environment*

---

The C Developer's Package consists of several software tools, integrated to work together to provide a complete DSP design environment for Innovative Integration DSP boards. This section discusses each of the tools included in the package and gives descriptions of each applets features and use. In the case of the third-party packages, such as the Code Composer, Codewright and Hypersignal programs, a brief introduction is given regarding the program and its use within the Developer's Package and the user is referred to the individual manuals accompanying those products for complete documentation.

---

## *The Texas Instruments C Compiler Toolset*

The C compiler supplied with the Developer's Package is the Texas Instruments (T.I.) Floating Point C Compiler toolset for the TMS320C3x/4x family. The compiler runs under Windows as a cross compiler, generating executable applications for the DSP processor which are then downloaded and executed using the other tools in the Developer's Package. The compiler is ANSI C compatible and sup-

ports nearly all standard C functions. Additional libraries provided with the Developer's System include C standard I/O and peripheral drivers for the A/D, D/A, bit-I/O and timers. Assembly language may also be mixed with C code for higher performance where required.

Typical application programs will consist of one or more C (.C), header (.H), and Assembly language (.ASM) source files, as needed. Additionally, target program generation requires use of a linker command file (.CMD) which specifies the memory map for the target and optionally includes commands defining the libraries to be linked into the final application.

Users of the Codewright editor will also employ make (.MK), make include files (.MKI) and project files (.PJT). The example programs included in the Developer's Package illustrate the use of these files and give example files to use as a basis for custom DSP applications.

Users of the Code Composer Debugger will also employ make (.MAK), workspace (.WSP) and special Code Composer-specific script files (.GEL). The example programs included in the Developer's Package illustrate the use of these files also and give example files to use as a basis for custom DSP applications.

## **C Compiler Toolset Usage**

The C compiler may be run directly from a DOS Prompt window under Windows 95/NT as described in the TI toolset documentation. Also included in the installation directory are batch files useful for manually rebuilding applications programs within the DOS environment. `COMPILE.BAT` and `ASSEMBLE.BAT` are batch files which will recompile/reassemble a C or Assembly source file (respectively) specified as a target parameter to these batch files. The `LINK.BAT` will invoke the TI Linker to link several object modules to create a target executable (.OUT) file, consuming a linker command file (.CMD) as a parameter.

---

## *Codewright Editor*

Codewright is a flexible, high-performance, integrated code generation environment developed by Premia Corporation and bundled into the Innovative Integration Zuma toolset. While Codewright supports code editing, emulating numerous popular editing packages (Brief, EMACS, CUA, etc.), Codewright is much more than simply a Window editor – it supports literally hundreds of extremely useful extensions for project-oriented code development, including syntax highlighting, file differencing, version control interfaces, file greps and much more. Additionally, each developers package includes the Innovative Integration-developed, TIDeps DLL which further extends the capabilities of Codewright to support rapid development of DSP programs via the TI toolset (Compiler, Assembler, Linker and Archiver). For a complete reference to the features of Codewright, refer to the *Codewright Users Guide*. For details on the features of the Innovative Codewright TIDeps DLL extension, refer to the accompanying *CodeWright Support Reference Manual*.

Custom DSP code creation takes place within the Codewright environment using its project management tools to maintain the relationship between source files, linker command files, and build revisions. The example programs included in the Developer's Package each have a Codewright project file (.PJT) while specifies the project component files and make include file (.MKI) which specifies the TI tool options. The supplied Codewright extension DLL automatically generates make-compatible make files which are used to construct application executables or rebuild libraries while within the Codewright environment. If the user wishes to compile outside of Codewright (or has not purchased the package), these make files may be used from the DOS command line to rebuild individual project files or the entire target file set.

---

## *Code Composer Debugger*

Code Composer is a software program for high-level TIC and Assembly Language debugging which supports high-performance, JTAG or MPSD-based hardware

assisted debugging directly on the target DSP to gain access to the internal register set, peripherals, and bus of the target board in order to load, run, and debug applications. Also integrated into the Code Composer software package is a code management subsystem for editing files as well as creating and compiling DSP projects.

If desired, custom DSP code development can also take place entirely within the Code Composer environment using its project management tools to place source files, libraries and linker command files into project workspaces (.WSP) in order to build executables. The example programs included in the Developer's Package each have a Code Composer project file (.MAK) and workspace file (.WSP) associated with them which may be used to recompile the example.

Since Codewright is a superior code generation environment to Code Composer, most users find that it is efficient to edit and compile DSP projects within Codewright to the point where executable target code has been created and then use Code Composer to load and debug the executable, as required.

For complete documentation on the Code Composer package, see the manuals provided by Go DSP.

---

### *Support Applets*

The Developer's Package includes four support applications supporting general DSP development: the terminal emulator (TERMINAL . EXE), the COFF file downloader (DOWNLOAD . EXE), the COFF file display utility (COFFDUMP . EXE) and the FLASH prom programming facility (BURN . EXE).. This section describes the functionality of each of the applications and their use within the development system.

The functions provided by each of the applications may be configured through menu selections available within each of the applets themselves. Generally, parameters governing the behavior of each applet are stored in program-specific .INI files, located in the directory from which the applet is invoked. See the discussion below for applet-specific parameters.



---

## The Terminal Emulator

The terminal emulator provides a C language-compatible, standard I/O terminal emulation facility for interacting with the `stdio` library running on the DSP processor. Display I/O calls such as `printf()`, `scanf()`, and `getchar()` are routed between the DSP target and the Host terminal emulator applet where ASCII output data is presented to the user via a terminal emulation window and host keyboard input data is transmitted back to the DSP. The terminal emulator works almost identically to console-mode terminals common in DOS and Unix systems, and provides an excellent means of accessing target program data or providing a simple user interface to control target application operation.



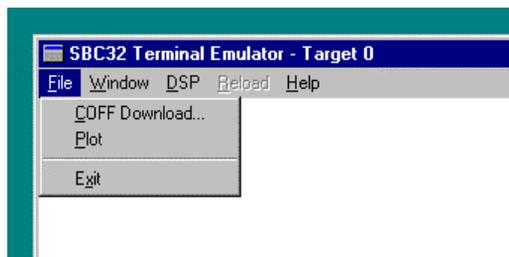
**FIGURE 1. Terminal emulator applet**

The terminal emulator is straightforward to use. The emulator will respond to `stdio` calls automatically from the target DSP card and should be running before the DSP application is executed in order for the program run to proceed normally. DSP program execution will be halted automatically at the first `stdio` library call if the terminal emulator is not executing when the DSP application is run, since standard I/O uses hardware handshaking, except on stand-alone SBC targets. `stdio` output is automatically printed to the current cursor location (with wraparound and scrolling), and console keyboard input will also be displayed as it is echoed back from the target.

The terminal emulator also supports Windows file I/O using the library routines `fopen()`, `fclose()`, `fread()`, `fwrite()`, `fseek()` and `fflush()`. Refer to the Appendix for prototypes and usage of these library functions as their usage is not 100% ANSI compliant.

---

**Terminal Emulator Menu Commands.** The terminal emulator provides several menus of commands for customizing its functionality. The following is a description of each menu entry available in the terminal emulator, and its effects.



**FIGURE 2. Terminal emulator file menu**

**File Menu.** File | COFF Download - provides for COFF program downloads from within the terminal emulator. When selected, a file requester dialog box is opened and the pathname to the COFF filename to be downloaded is selected by the user. Clicking “Open” in the file requester once a filename has been selected will cause the requester to close and the file to be downloaded to the target and executed. Clicking “Cancel” will abort the file selection and close the requester with no download taking place.

**Q62 Users:** Terminal supports downloading of .OUT or multi-processor .MPO files. .MPO files provide a means of downloading separate .OUT files to multiple processors simultaneously, which greatly simplifies the task of synchronizing execution in a multi-processor environment.

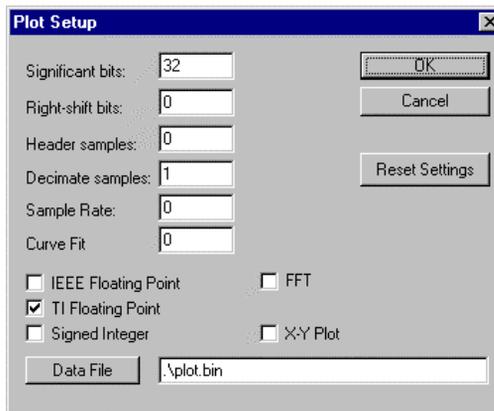
**NOTE:** File | COFF Download physically resets the target DSP (in order to initiate the target Talker program) prior to the download. When using the terminal emulator in conjunction with the Code Composer debugger, use Code Composers File | Load Program facility to download executable code to the target rather than the terminal emulator’s download facility, since the Code Composer mechanism does not physically reset the target during the download and is not reliant on the target Talker to perform the download.



**FIGURE 3. Diagnostic received when target DSP is halted.**

If you attempt to download using the COFF Download menu within the terminal emulator while using Code Composer, you may receive the diagnostic dialog box which indicates that Code Composer has *halted* the target processor via the JTAG hardware link and while in this halted state, the terminal emulator cannot invoke the Talker program on the target DSP in order to perform the software download. To correct this problem, execute the Debug | Run Free menu command from within Code Composer to release the DSP from JTAG control. Afterwards, clear the terminal emulator error message dialogs and retry the terminal emulator COFF Download.

File | Plot – opens the Plot dialog box, similar to the one listed below.



**FIGURE 4. Terminal emulator plot menu dialog box.**

---

The Plot dialog specifies all of the available options for plotting binary data in Host PC files. Binary data files, usually created by target DSP programs using the `fopen()` and `fwrite()` functions, may contain data in a wide variety of formats which may be plotted in a window from within this dialog box.

Each time data is plotted in the plot window, statistics on the plotted graph are calculated. These statistics are reported in the graph window. The statistics include:

*Min* displays the minimum value in the data set.

*Max* displays the maximum value in the data set.

*Delta* displays the difference between the minimum and maximum values in the data set.

*Sdev* displays the standard deviation of the data set.

*Mean* displays the mean value of the data set.

The terminal emulator is capable of plotting files in which binary data has been stored in a wide variety of formats. The default data file format is successive 32-bit (four-byte) values each representing a single TI floating point Y amplitude value. X axis data is not contained in the file and the Y axis amplitude data is plotted against an implied X axis of successively incrementing sample # values, starting at zero.

Each of the available plot options is detailed below.

**Edit Boxes . Significant Bits** specifies the number of significant bits in each data value stored in the data file. The number of bits may range from one to thirty-one. This parameter allows you to plot data gathered from a device at virtually any resolution. For example, if data is accumulated from a 12-bit A/D converter and stored into a binary data file from the target DSP, it would be stored on disk as 16-bit byte-pairs. When plotting this data, with significant bits set to 12, the fallow upper four bits of each 16-bit sample in the data file will be ignored during the data plotting operation.

This parameter indirectly specifies the size of each data sample within the data file, as well. The size of each sample (in bytes) is given by the equation:

Sample size = (significant bits + 7) / 8

---

The sample size is always the truncated integer result of this formula. Use of the term *sample* throughout the rest of this section refers to clusters of bytes within the data file of size *sample size*.

*Shifted* specifies the number of bits to shift each data sample stored in the data file, prior to plotting. The number of bits may range from negative thirty-one to positive thirty-one. This parameter allows you to plot data gathered from a device when the output lines of the device are not mapped onto the low-order lines of the data bus. For example, on some of Innovative's DSP boards, a 12-bit A/D is mapped onto data bus bits 15 through 4 rather than on bits 11 through 0. If this data were plotted without modification, the data would erroneously range from  $-32767$  to  $+32768$  rather than the actual 12-bit A/D range of  $-2047$  to  $+2048$ . By specifying a *Shifted* parameter of 4, each data sample extracted from the data file would be right-shifted four bits prior to plotting to compensate for this effect.

*Decimate* specifies the number of file data points to be skipped between plotted data samples. This option is useful when dealing with a data file containing more than one sample set or in instances where more data is contained in the file than need be plotted. This field must contain a value greater than or equal to one. A value of one specifies that no data should be skipped; a value of two specifies that every other data sample should be discarded, etc.

*Header* specifies the number of file data samples to be skipped at the beginning of the data file before extracting data to be plotted. This option is used to skip irrelevant data appearing at the beginning of a data file.

**Note:** Combinations of *Decimate* and *Header* can be used to view individual, 16-bit channels of data acquired as 32-bit pairs on certain DSP boards. For example, the PC31 features two A/D channels, A and B. The A channel is mapped onto the upper 16-bits of the 32-bit data bus while the B channel is mapped to the lower 16-bits of the bus. If this data were written to a data file as 32-bit data, The *Decimate* parameter could be set to 2 to allow plotting of every other sample in the file (all of the A channel data). Further, the *Header* parameter could be set to 1 in conjunction with the abover *Decimate* setting to allow skipping of the first sample in the file resulting in order to plot of all of the B channel data in the file.

*Fit* specifies that the plotted data should be curvefit to the specified order, ranging from zero to five, using a least-squares regression technique. The curvefit data is plotted atop the actual data in red. The correlation coefficient of the fit and the curvefit equation are displayed in the graph window whenever this parameter is greater than zero.

---

*Data File* indicates that name of the file containing the data to be plotted.

**Radio Buttons.** *IEEE* – When checked, indicates that each sample in the data file is stored in 32-bit IEEE-754 floating-point format. When enabled, the Significant Bits and Shifted fields are ignored.

*TI* – When checked, indicates that each sample in the data file is stored in 32-bit TMS320 TI native floating-point format. When enabled, the Significant Bits and Shifted fields are ignored. This is the default data mode.

*Signed* – When checked, indicates that each data sample in the data file is signed integer data. When enabled, the Significant Bits and Shifted fields are observed.

**NOTE:** When *IEEE*, *TI* and *Signed* are unchecked, the data is assumed to have been stored in the data file as *unsigned* integer data.

*XY* – When checked, indicates that data samples have been stored in the data file as X-Y (distance, amplitude) pairs rather than in the default data format. In the default format, only the Y (amplitude) data is stored in the file and it is plotted against an implied, incrementing “sample number” X. In the XY mode, samples are parsed from the file and plotted in pairs. Therefore in this mode, half as many points are plotted from the data file.

*FFT* – When checked, indicates that a Fast Fourier Transform should be applied to the data in the data file prior to plotting.

File | Exit - exits the terminal emulator program.



**FIGURE 5. Terminal emulator Window menu.**

---

- 
- Window | Clear Screen - clears the terminal emulation screen and resets the current cursor position to the top left hand corner.
  - Window | Reset - causes the terminal emulator to reset all internal stdio processing and clear the screen. If processing is currently halted (via the File | stdio Disabled command), it is reenabled. The Reset command is useful when the terminal emulator needs to be initialized prior to running a new DSP application on the target. This can become necessary because the emulator uses multi-character control codes to implement cursor movement and screen control functionality and it is possible to halt DSP processing (via the JTAG debugger interface) in the middle of a stdio call which is processing a multi-character sequence. If the program is not continued, this causes the terminal emulator to misinterpret subsequent, new stdio activity. Terminal emulation should always be reset, either via this menu entry or by calling the `stdio_reset()` function within the new application, before new stdio activity is attempted.
  - Window | `stdio` Disabled - a toggling command which allows the user to temporarily disable stdio emulation. This will cause the DSP program to halt at the next stdio library call, and remain paused until stdio processing is again reenabled by selecting this menu entry. stdio activity processing is halted while the menu entry is checked.
  - Window | Always On Top - a toggling command which will cause the terminal emulator to float above other windows on the desktop. This is useful when running stdio-based code from within the Code Composer environment, where the terminal needs to be visible at all times. The terminal will remain atop other windows when this entry is checked. Select the entry again to uncheck and allow the terminal emulator window to be obscured by other windows.
  - Window | Quiet Mode – Disables verbose error and diagnostic messages during terminal execution.

### DSP Menu

- DSP | Reset - causes the terminal emulator to momentarily assert the target's physical reset pin, bringing the target board into a cold-start, initialized condition.
- DSP | Interrupt - causes the terminal emulator to trigger a target mailbox interrupt using the test code of 0x80 as the signal value. Helpful during testing of target interrupt handlers.

---

### Reload Menu

- Reload - Causes the terminal emulator to redownload and restart the last COFF application previously selected with the File | COFF Download command.

### Help Menu

- About - presents program copyright and version information plus information pertaining to the use of Host resources by the target DSP board.

**Terminal Emulator Command Line Switches.** The terminal emulator also provides the following command line switches to further modify program behavior. The switches must be supplied via the command line or within Windows shortcut properties (see the Installation section for more information), and will override the default behavior of the applet.

- **-tX** - address selector switch, which allows the user to force the terminal emulator to interact with a specified target. This switch is particularly useful in multi-board installations to create instances of the emulator for targets other than target 0. See the Installation section for more information on multi-board installations. The *X* parameter specifies the logical target number with which to communicate. NOTE: For single-board targets, specify target 0 for boards connected via COM1 and target 1 for boards connected via COM2.
- **-ffilename** - address selector switch, which allows the user to force the terminal emulator to download the specified file to the target DSP board, as soon as the terminal emulator is loaded. This switch is particularly useful in situations where the terminal emulator is “shelled to” from within other Host applications (such as Codewright) to facilitate automatic execution of target applications employing standard I/O.

---

## The COFF File Downloader

The COFF downloader utility provides users with the capability to download and execute COFF files generated by the C compiler or Hypersignal toolsets. This allows users to distribute executable applications independent of the DSP development tools.



**FIGURE 6. The Coff File Downloader Applet**

The COFF downloader is simple to use. Double click on the COFF Downloader icon and the program will start and will open a small window with two menu entries, File and Window. To download an application, click on File | Download. This will present a file requester dialog box containing a list of suitable COFF files (.OUT) which can be downloaded. Select the desired target executable and click OK to proceed. Click Cancel to abort the download command without selecting a filename.

Once a file is selected, the target will be reset-cycled (to restart its talker) and the program will be downloaded and the application launched on the DSP. If any errors are encountered during the download or the download fails to succeed for any reason, an error message box will appear. Typical reasons for failure include improper file selections (a nonexistent or non-COFF format file was selected for download) or errors in hardware or software installation. If repeated errors are noted, proceed to the Installation Troubleshooting section below.

The COFF downloader provides for automated downloads for use in situations where a single application needs to be downloaded and run on the target each time the system is brought up. This can be valuable when placed in the Windows Startup Folder to automatically download a specific DSP program each time Windows is restarted.

**Q62 Users:** Download supports downloading of .OUT or multi-processor .MPO files. .MPO files provide a means of downloading separate .OUT files to multiple processors simultaneously, which greatly simplifies the task of synchronizing execution in a multi-processor environment.

---

The File | Exit menu selection will terminate the download application.

**COFF File Downloader Menu Commands** . The following is a brief description of commands available from the COFF Downloader menus:

### **File Menu**

File | COFF Download - provides for COFF program downloads from within the terminal emulator. When selected, a file requester dialog box is opened and the pathname to the COFF filename to be downloaded is selected by the user. Clicking “Open” in the file requester once a filename has been selected will cause the requester to close and the file to be downloaded to the target and executed. Clicking “Cancel” will abort the file selection and close the requester with no download taking place.

NOTE: File | COFF Download physically resets the target DSP (in order to initiate the target Talker program) prior to the download. When using the terminal emulator in conjunction with the Code Composer debugger, use Code Composers File | Load Program facility to download executable code to the target rather than the terminal emulators download facility, since the Code Composer mechanism does not physically reset the target during the download and is not reliant on the target Talker to perform the download.

- File | DSP Reset - causes the terminal emulator to momentarily assert the target’s physical reset pin, bringing the target board into a cold-start, initialized condition.
- File | Exit - exits the terminal emulator program.

### **Window Menu**

- Window | Quiet Mode – Disables verbose error and diagnostic messages during terminal execution.
- Window | About - presents program copyright and version information.

---

## Reload Menu

- **Reload** - Causes the terminal emulator to redownload and restart the last COFF application previously selected with the File | COFF Download command.

**COFF File Downloader Command Line Switches.** The COFF Downloader also provides the following command line switches to further modify program behavior. These switches must be used in Windows 95/NT shortcut icons (see the Installation section for more information), and will override the same selection made in the configuration utility.

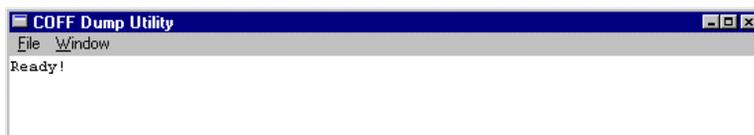
- **-tX** - target number selector switch, which allows the user to force the terminal emulator to interact with the specified target. This switch is particularly useful in multi-board installations. See the Installation section for more information on multi-board installations. The X parameter specifies the logical target number with which to communicate. For single-board targets, specify target 0 (zero) for boards connected via com1 and target 1 (one) for boards connected via com2.
- **-q** - force quiet mode switch, which causes the terminal emulator to omit non-fatal warning messages. Fatal errors are still presented in message boxes.
- **-dpathname** - cause the downloader to automatically download the named file. Complete path and filename must be given (as in `c:\sbc32cc\hello.out`).



---

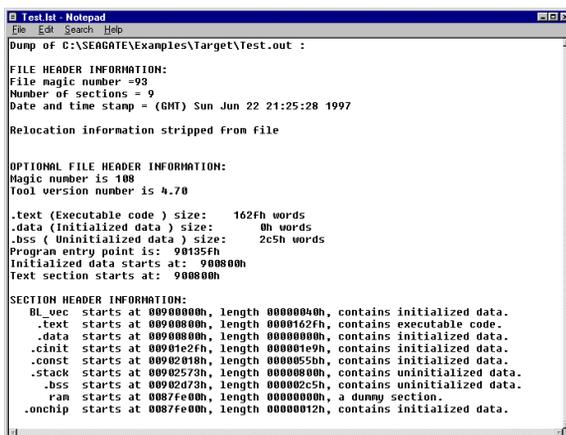
## The COFF File Dump Utility

The COFF downloader utility provides users with the capability to generate a report detailing the memory usage of target DSP programs generated using the TI tool set.



**FIGURE 7. The COFF Dump Utility**

COFFDUMP .EXE parses through COFF files stored in files on the hard disk and ascertains the complete memory consumption by the DSP program. Memory usage for each of the sections defined in the applications command file are tabularized and the results are written to the Windows NotePad scratch buffer. If desired, Not-Pad can then be used to write the data to disk or to a printer.

A screenshot of a Windows Notepad window titled "Test.txt - Notepad". The window shows the output of the COFF Dump utility for a file named "Test.out". The output includes file header information, optional file header information, section sizes, and a detailed section header table.

```
Dump of C:\SEAGATE\Examples\Target\Test.out :  
  
FILE HEADER INFORMATION:  
File magic number = 93  
Number of sections = 9  
Date and time stamp = (GMT) Sun Jun 22 21:25:28 1997  
  
Relocation information stripped from file  
  
OPTIONAL FILE HEADER INFORMATION:  
Magic number is 108  
Tool version number is 4.70  
  
.text (Executable code ) size:    162fh words  
.data (Initialized data ) size:    0h words  
.bss ( Uninitialized data ) size:   2c5h words  
Program entry point is:  90195fh  
Initialized data starts at:  900800h  
Text section starts at:  900800h  
  
SECTION HEADER INFORMATION:  
  .bl_vec starts at 00900000h, length 00000040h, contains initialized data.  
  .text starts at 00900800h, length 0000162fh, contains executable code.  
  .data starts at 00900800h, length 00000000h, contains initialized data.  
  .cinit starts at 00901e2fh, length 000001e9h, contains initialized data.  
  .const starts at 00902010h, length 00000550h, contains initialized data.  
  .stack starts at 00902573h, length 00000800h, contains uninitialized data.  
  .bss starts at 00902d73h, length 000002c5h, contains uninitialized data.  
  .ram starts at 0087fe00h, length 00000000h, a dummy section.  
  .onchip starts at 0087fe00h, length 00000012h, contains initialized data.
```

**FIGURE 8. COFF Dump utility output.**

---

---

**COFF Dump Utility Menu Commands .** The following is a brief description of commands available from the COFF Downloader menus:

**File Menu**

- File | Dump – Involes the standard Windows file selector window for COFF output files (.OUT). Parses through selected file and writes diagnostic dump of contents of executable image to NotePad scratch buffer.
- File | Exit - exits the dump utility program.

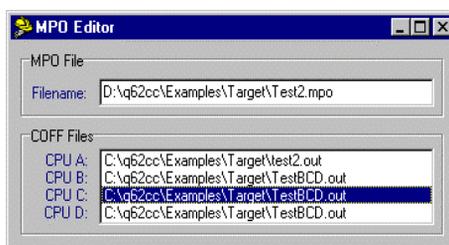
**Window Menu**

- Window | About - presents program copyright and version information.

---

## The MPO Editor

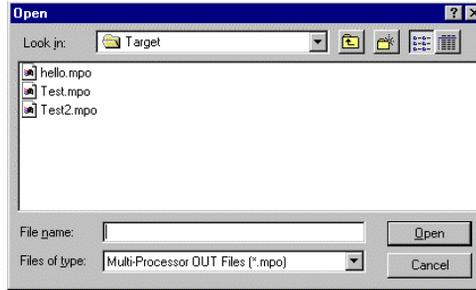
The MPO Editor provides a means of editing the special configuration files used on the Q62 to allow downloading of multiple COFF object files simultaneously. The Terminal and Download applets for the Q62 understand the MPO file format and are able to consume .MPO files as well as .OUT files as download arguments. Attempting to download an MPO file from within Terminal or Download will cause new code to be loaded onto and executed by all processors. This is in contrast to the downloading a standard COFF .OUT file, which simply downloads and executes code on processor A only.



**FIGURE 9. The MPO Editor**

The MPO Editor is simple to use. Double click on the MPO Editor icon and the program will start, presenting a single window similar to the figure above. The Filename edit box contains the name of the MPO file currently being edited. The COFF Files list box contains the filenames of the executable .OUT files to be downloaded to each of the processors on the DSP board.

**Opening an Existing MPO File.** To open an existing MPO file, double-click on the Filename edit box. This will open a dialog box in which you may browse to the directory containing the MPO file. Selecting an MPO file from within this dialog will dismiss the dialog and will make the selected MPO file current.



**FIGURE 10. MPOEditor "Open" Dialog box.**

**Creating a new MPO File .** To create a new MPO file, simply type the complete path specification to the new MPO file you wish to create into the Filename edit box, then press Enter. Alternately, you may double-click the Filename edit box and browse to the directory in which you wish to create the new MPO file and then type the name of the MPO file to create into the File name edit box of the Open dialog and finally press the Open button.

**Changing the COFF File List.** To change any entry in the list of COFF files to be downloaded, double-click on the line in the COFF File list box corresponding to the CPU which is to execute the new .OUT file. This will open a dialog box in which you may browse to the directory containing the .OUT file to be downloaded to the selected CPU.



**FIGURE 11. COFF File list change dialog box.**

---

Double clicking on a file in the list box within the Open dialog will replace the selected line within the MPO editor COFF File list box with the selected file.

**Saving MPO Editor Changes.** Closing the MPO editor applet automatically saves the current MPO Filename in the registry so that it is persistent from run to run of the applet.

If the specified MPO file does not exist, it is created. Then, the names of the .OUT files contained in the COFF Files list box are written into current MPO file. Any previous contents of the MPO file are overwritten.

## The Viewer Applet

**Introduction.** Viewer is a software debugging utility supplied with Innovative Integration DSP boards. Viewer supports interactive execution of each of the DLL functions supplied in the Zuma toolset, interactive display of all DLL allocated and addressable memory structures in a variety of formats and other forms of low-level DSP board control.

Viewer is useful during the DSP code development cycle, before a Host application program has been written and debugged to deal with data flow between the DSP and the Host PC.

Viewer is based on a public domain Windows Forth package, Win32For. Viewer supports the full extensibility of Forth scripts may be written in the Forth language to assist in the Host/DSP debugging effort.

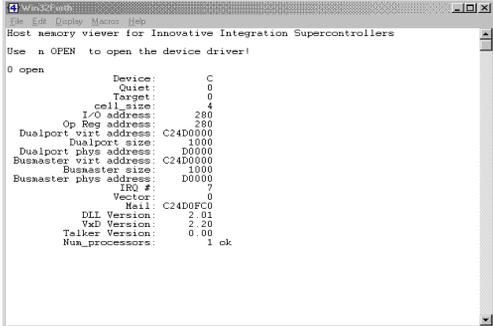
**Starting the Program .** Viewer may be executed by changing into the II\_BOARD directory and executing the VIEWER.EXE program file. When invoked, the program will open a single window, shown below.



**FIGURE 12. Viewer main window**

---

**Opening the Target.** Before attempting to communicate with the target processor, you must first “open” the target device and driver. This is accomplished using the `open` command.



```
Host memory viewer for Innovative Integration Supercontrollers
Use n OPEN to open the device driver!

0 open
   Device:      C
   Quiet:       0
   Target:      0
   cell_size:   4
   I/O address: 280
   Op Reg address: 280
   Dualport virt address: C24D0000
   Dualport size: 3000
   Dualport phys address: D0000
   Busmaster virt address: C24D0000
   Busmaster size: 3000
   Busmaster phys address: D0000
   IRQ #:       7
   Vector:      0
   Mail:        C24D0FC0
   DLL Version: 2.03
   VxD Version: 2.20
   Talker Version: 0.00
   Num_processors: 1 ok
```

**FIGURE 13.** Opening the target DSP

When the target is opened, strategic entries in the DLL `cardinfo` structure are read and displayed. While Viewer provides access to all of the `cardinfo` elements, only the most common ones are displayed during `open`.

**Accessing Shared Memory.** All of Innovatives bus-based DSP products provide some sort of “shared memory”. On ISA bus boards, this is a fixed block of dual ported memory, accessible by both the Host PC processor and the DSP processor.

On newer, PCI-based boards, the shared memory is actually Host PC memory allocated by the board’s device driver as contiguous, page-locked memory suitable for use as a “bus-mastering” target or source.

Regardless of the type of shared memory, Viewer provides a means of accessing it. This can be very helpful during the development process, before your custom Host program has been developed. Ultimately you must generate custom code for both the Host and the target to provide the umbilical communications layer between the DSP program running on the target board and your Host application program. But in the interim, Viewer can be used to provide basic access to the shared memory pool and limited diagnostics capabilities.



---

dpdump      To display a range of dual ported memory

dump        To display a range of host memory (Viewer application local space)

Viewer may also be used to view resources located in the I/O space of the PC. The variants are:

idump        To display a range of the DSP board's I/O space

odump        To display a range of the DSP board's operations register space  
(PCI only)

iodump       To display a range of host I/O space (Win95 only)

If the target Talker is running (after a Reset), Viewer allows you to view target memory without moving its contents to the Host first. The commands are:

pdump        To display target DSP program memory (Talker monitor must be running)

ddump        To display target DSP data memory (Talker monitor must be running)

Each of these commands is *modal* and selects the default memory region to be accessed in all subsequent dump and plot commands. These spaces may be explicitly made active using the space mode commands:

program-space    Selects target program memory region as "default" memory space

data-space        Selects target data memory region as "default" memory space

bm-space         Selects bus master memory region as "default" memory space

dp-space         Selects dual port memory region as "default" memory space

memory-space    Selects host PC memory region as "default" memory space

space

i-space          Selects DSP board I/O block as "default" memory space

op-space         Selects target operation registers region as "default" memory space

---

---

io-space	Selects Host PC I/O region as “default” memory space
hpi-space gets)	Selects Host PC I/O region as “default” memory space (C6x tar-

Dumping and plotting commands respect the current display format type, set by one of the format mode commands below.

signed	Sets display format to signed integer
unsigned	Sets the display format to unsigned integer
floating	Sets the display format to TI 32-bit floating-point format
ieee	Sets the display format to IEEE-754 32-bit floating point format.

These mode commands remain in effect until explicitly changed. The default mode is unsigned.

**Modifying Shared Memory.** The currently active memory region may be modified using a number of Viewer commands, listed below. These commands support clearing or filling a region of memory or altering a single cell of memory.

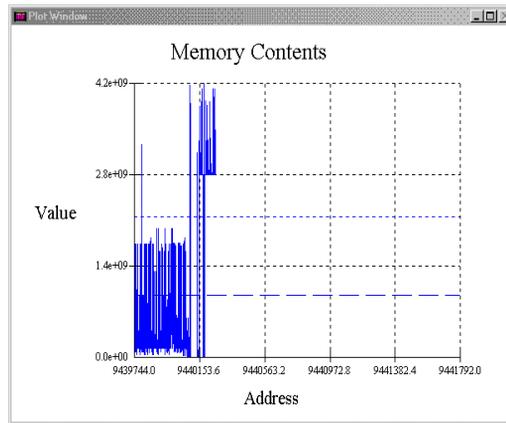
t!	Stores a value into specified target memory cell
t@	Retreives a value from specified target memory cell
tf!	Stores a floating point number into specified target memory cell.
tf@	Retrieves a floating point number from specified target memory cell.
tfill	Fills a region of target memory with a seed value
terase	Clears a region of target memory to zeros.
tdump	To display a region of memory in the “current” memory space.

Each of the se target memory operators deals with the current default *memory space* and *data type*.

---

**Plotting Shared Memory.** Data may be plotted rather than dumped in numeric format. Plotting is also modal and is controlled by the same display formatting commands listed above. To plot a data range, use the plot command, as shown below.

0 100 plot      To plot the data range starting at offset 0 in the current memory region using the current display format .



**FIGURE 15. Viewers plot window**

Plots, like dumps are automatically performed out of the currently selected memory region using the current data type.

**Generic View DLL functions.** The functions tabularized below may be executed interactively within Viewer. The parameters to each function must be pushed onto the stack in the order shown prior to invoking the function. The parameters column below lists the required parameters for each function. The dash in the parameter description denotes where the function name should be substituted when executing the command.

**TABLE 1. Generic DLL Function List**

Function	Parameters	Description
target_open	target - f	Opens driver for specified target DSP board. Returns boolean.
target_close	target - f	Closes driver for specified target DSP board. Returns boolean
target_cardinfo	target - a	Returns address of cardinfo structure for target.
nicoffld	string target handle - f	Loads a COFF executable file onto target DSP
host_interrupt_enable	target - f	Enables a previously installed virtual interrupt handler.
host_interrupt_disable	target - f	Disables a previously enabled virtual interrupt handler
host_interrupt_install	target fcn -	Installs a virtual interrupt handler
host_interrupt_deinstall	target -	Removes a virtual interrupt handler.
target_reset	target -	Physically asserts reset on the target DSP board.
target_run	target -	Deasserts reset on the target DSP board
target_outport	target port value -	Outputs a value to specified DSP board I/O port address
target_inport	target port - n	Inputs a value from specified DSP board I/O port
target_opreg_outport	target port value -	Outputs a value to specified DSP board operation port address
target_opreg_inport	target port - value	Inputs a value from specified DSP board operation port
target_control	target bit state -	Modifies a bit in the control register of the target DSP board
read_mailbox	target box - value	Reads the specified mailbox of the target DSP board
write_mailbox	target box value -	Writes to the specified mailbox of the target DSP board.
check_outbox	target box - f	Interrogates the specified output mailbox status
check_inbox	target box - f	Interrogates the specified input mailbox status
read_mb_terminate	target box key mode - f	Reads the specified input mailbox, if full
write_mb_terminate	target box value mode - f	Writes to the specified output mailbox, if empty
clear_mailboxes	target	Clears all mailboxes to empty state
mailbox_interrupt	target value -	Interrupts the target DSP after writing value to special mailbox
mailbox_interrupt_ack	target -	Acknowledges target to Host interrupt, returns special mailbox contents
target_key	target - key	Reads terminal mailbox, returns an 8-bit contents
target_emit	target value -	Writes 8-bit value to terminal mailbox
target_Tx	target value -	Writes 32-bit value to terminal mailbox
target_Rx	target - key	Reads 32-bit value from terminal mailbox
target_get_semaphore	semaphore target -	Gains ownership of specified target semaphore

target_interrupt	target -	Interrupts target DSP board
target_request_semaphore	target semaphore -	Requests ownership of specified target semaphore
target_own_semaphore	target semaphore -	Interrogates ownership status of specified semaphore
target_release_semaphore	target semaphore -	Relinquishes control of specified semaphore
target_check	target - f	Interrogates for Talker running on target
start_app	target -	Starts a previously downloaded target application program
start_talker	target - f	Starts the target Talker executing.
target_revision	target - f	Returns the revision of the target Talker
talker_fetch	target addr - n	Uses the Talker to fetch contents of specified target memory address
talker_store	target addr value -	Uses Talker to store value to specified target memory address
talker_read_memory	target page addr - n	Uses the Talker to fetch contents of specified target memory address
talker_store	target page addr value -	Uses Talker to store value to specified target memory address
talker_download	target addr cnt -	Downloads a block of data or code to target DSP
talker_launch	target addr -	Launches downloaded application at boot vector address
talker_resume	target -	Resumes execution after suspended by Talker (not available all targets)
talker_registers	target -	Returns Talker register save address on target
target_slow	target -	Changes bus control to permit safe FLASH ROM access
target_fast	target -	Changes bus control to support fast target code execution
talker_flash_sector_erase	target sector -	Erases specified sector in FLASH ROM on target
talker_flash_init	target -	Initializes FLASH ROM on target.
talker_flash_offset	target offset -	Specifies memory offset of base of FLASH ROM on target

**Viewer Command Reference.** Viewer is capable of operating on a variety of memory regions. The commands below may be used to enable display or modification of specific memory regions using the target memory operator commands. These commands are modal and remain in effect until explicitly changed.

**TABLE 2. Viewer “target” memory selection commands**

Function	Parameters	Description
io-space	--	Makes host I/O space the “current” target memory space

memory-space	--	Makes host memory space the “current” target memory space. All addresses are relative to the base of Viewers executable image in memory
bm-space	--	Makes host bus master memory space the “current” target memory space. All addresses are relative to the base of the page-locked bus master memory block.
dpram-space	--	Makes shared DSP/Host dual-port memory the “current” target memory space. All addresses are relative to the base of the shared memory pool.
o-space	--	Makes DSP card operations register space the “current” target memory space. All target memory addresses are specified relative to the beginning of this region.
i-space	--	Makes DSP card I/O space the “current” target memory space. All target memory addresses are specified relative to the beginning of this region.
program-space	--	Makes DSP card program memory the “current” target memory space. All target memory addresses are specified relative to the beginning of this region.
data-space	--	Makes DSP card data memory space the “current” target memory space. All target memory addresses are specified relative to the beginning of this region.
i-space	--	Makes DSP card I/O space the “current” target memory space. All target memory addresses are specified relative to the beginning of this region.
hpi-space	--	Makes DSP card Host Port Interface space the “current” target memory space. All target memory addresses are specified relative to the beginning of this region (C6x targets only)

Viewer supports storing and fetching from the “currently-selected” target memory region. The commands below may be used to modify the currently memory region. These commands are subject to the current target memory region mode, selected above.

**TABLE 3. Viewer “target” memory operators**

Function	Parameters	Description
t!	n a --	Pronounced “t store”. Stores integer n into address a in target memory. For example 0x100 0x1000 t! stores 100h into target memory address 1000h.
t@	a - n	Fetches integer n from address a in target memory. For example 0x1000 t@ returns contents of target memory address 1000h onto stack.
tf!	a - r -- (fp stack)	Stores floating pt r into address a in target memory. For example 1.23 0x1000 tf! stores floating point 1.23 into target memory address 1000h
tf@	a - -- r (fp stack)	Fetches floating pt r from address a in target memory. For example 0x1000 tf@ returns floating pt contents of target memory address 1000h.
tdump	a n --	Dumps n cells of the current memory region starting at address a according to current dump mode (ie signed). For example unsigned 0 100 ddump shows 100 cells of target memory starting at 0000h as unsigned integers.
tfill	a n c --	Fills n cells of target memory starting at a with integer c. For example: 0 100 0x1234 tfill fills 100 target memory cells starting at 0000h with value 1234h.
terase	a n --	Zeros n cells of target memory starting at a. For example 0 1000 derase erases 1000 cells of target memory starting at offset n in the current memory space.

Viewer supports dumping ranges of target memory in text form and graphically. The commands below are used to display ranges of target memory.

---

---

**TABLE 4. Target memory display operators**

Function	Parameters	Description
tdump	a n --	Dumps n cells of target memory starting at offset a interpreted according to current dump mode. For example 0 100 DEDUMP shows 100 16-bit cells of dual port memory starting at D000:0000h
plot	a n --	Plots n cells of memory starting at address a according to current dump mode (ie signed) and active memory space. Requires EasyPlot in working directory. For example SIGNED 0 100 PLOT graphs 100 cells of memory starting at 0000h as unsigned integers.

The dump and plot commands operate on the current target memory region. The commands below modify how data is interpreted during the data display operation.

**TABLE 5. TDUMP mode selector commands**

Function	Parameters	Description
signed	--	Subsequent DUMPs/PLOTs show signed values.
unsigned	--	Subsequent DUMPs/PLOTs show unsigned values.
floating	--	Subsequent DUMPs/PLOTs show TI floating point values.
ieee	--	Subsequent DUMPs/PLOTs show IEEE floating point values.

The commands below are shorthand convenience forms of the dump and target memory access commands. They automatically select a target memory region and perform a target memory accesses using a single, short-form command. Note: All addresses are specified as *offsets* into the selected memory region.

**TABLE 6. Shorthand memory dump commands**

Function	Parameters	Description
iodump	a n --	Makes host I/O space current and dumps the specified range.
idump	a n --	Makes DSP card I/O space current and dumps the specified range.
odump	a n --	Makes DSP card operations space current and dumps the specified range.
bmdump	a n --	Makes host bus master memory space current and dumps the specified range.
dpdump	a n --	Makes dual port memory space current and dumps the specified range.
pdump	a n --	Makes target program memory space current and dumps the specified range.
ddump	a n --	Makes target data memory space current and dumps the specified range.
hpidump	a n --	Makes target HPI memory space current and dumps the specified range.
p@	a - n	Makes DSP program memory current and fetches from it.
p!	n a --	Makes DSP program memory current and stores into it.
d@	a - n	Makes DSP data memory current and fetches from it.
d!	n a --	Makes DSP data memory current and stores into it.
i@	a - n	Makes DSP board I/O space current and fetches from it.
i!	n a --	Makes DSP board I/O space current and stores into it.
o@	a - n	Makes DSP board operations register space current and fetches from it.
o!	n a --	Makes DSP board operations register space current and stores into it.
bm@	a - n	Makes bus master memory space current and fetches from it.
bm!	n a --	Makes bus master memory space current and stores into it.
dp@	a - n	Makes dual port memory space current and fetches from it.
dp!	n a --	Makes dual port memory space current and stores into it.
io@	a - n	Makes Host I/O space current and fetches from it. 32-bit form
io!	n a --	Makes Host I/O space current and stores into it.
ioh@	a - n	Makes Host I/O space current and fetches from it. 16-bit form
ioh!	n a --	Makes Host I/O space current and stores into it.
ioc@	a - n	Makes Host I/O space current and fetches from it. 8-bit form
ioc!	n a --	Makes Host I/O space current and stores into it.
hpi@	a - n	Makes Target HPI space current and fetches from it.
hpi!	n a --	Makes Target HPI space current and stores into it.

Viewer maintains two independent user-accessible stacks onto which parameters are placed for consumption by Viewer commands.

The *parameter stack* is a 32-bit wide stack used to contain addresses and integer parameters to and results from functions.

The *floating point stack* is used to hold used to hold floating point parameters to and results from Viewer functions. Floating point arithmetic takes place directly on the 8087 numeric stack. Viewer interprets numbers as reals when an 'e' is embedded in a literal number. Parameters on the fp stack are denoted below the parameter stack notation in the tables below.

**TABLE 7. Viewer math and binary operators**

Function	Parameters	Description
+	n1 n2 - n	Adds n2 to n1 leaving the result n. For example: 10 20 + . adds 10 and 20 and prints the result.
-	n1 n2 - n	Subtracts n2 from n1 leaving the result n. For example: 20 10 - . subtracts 10 from 20 and prints the result.
*	n1 n2 - n	Multiplies n1 by n2 leaving the result n.
/	n1 n2 - n	Divides n1 by n2 leaving the result n.
f+	-- r1 r2 - r	Adds r2 to r1 leaving the floating point result r. For example: 10.0 20.0 f+ f. adds 10 and 20 and prints the result.
f-	-- r1 r2 - n	Subtracts n2 from n1 leaving the result n. For example: 20 10 - . subtracts 10 from 20 and prints the result.
f*	-- r1 r2 - n	Multiplies r1 by r2 leaving the result r.
f/	-- r1 r2 - n	Divides r1 by r2 leaving the result r.
and	n1 n2 - n	Bitwise ANDs n1 and n2 leaving result n.
or	n1 n2 - n	Bitwise ORs n1 and n2 leaving result n.
xor	n1 n2 - n	Bitwise XORs n1 and n2 leaving result n

The commands below support Viewer dictionary display and modification.

**TABLE 8. Viewer dictionary commands**

Function	Parameters	Description
words	--	Displays the names of all available Viewer commands.
empty	--	Empties the Viewer dictionary of all user-defined commands.

:	-- wordname	Begins definition of a new Viewer command (called a <i>word</i> ).
;	--	Terminates definition of a new word.

The commands below affect global Viewer operation.

**TABLE 9. Viewer system commands**

Function	Parameters	Description
bye	--	Terminates Viewer, returns to the operating system.
z	-- filename	Invokes Codewright editor on specified filename
dir	-- dirspec	Displays the specified directory
chdir	-- dirspec	Changes to the specified directory

**TABLE 10. Viewer system commands**

Function	Parameters	Description
.s	--	Non-destructively prints entire parameter stack contents.
f.s	--	Non-destructively prints entire floating point stack contents.
.	n --	Prints the integer on top of the parameter stack.
f.	--	Prints the real number on top of the floating point stack.
	r --	
decimal	--	Changes default I/O conversion radix to decimal.
hex	--	Changes default I/O conversion radix to hexadecimal. Numeric literals prefixed with 0x are interpreted in hexadecimal, regardless of current radix.
bye	--	Terminates Viewer, returns to the operating system.
z	-- filename	Invokes Codewright editor on specified filename

The following commands are convenient Viewer command shortcuts to common DLL functions.

**TABLE 11. Target DLL function shortcuts**

Function	Parameters	Description
+reset	--	Places the target in the reset state
-reset	--	Removes the target from the reset state
reset	--	Reset-cycles the DSP board
run	-- filename	Downloads and runs specified COFF .OUT file
break	--	Fires a target interrupt
open	n --	Opens the specified target DSP device driver

---



---

close	--	Closes the currently open DSP device driver
inbox?	slot - f	Reports status of specified input mailbox
outbox?	slot - f	Reports status of specified output mailbox
?mailbox@	slot - n f	Reads input mailbox, if full. Returns value read and status.
?mailbox!	n slot --	Writes output mailbox, if empty. Returns status.
.boxes	--	Destructively dumps all input and output mailboxes



---

*Introduction*

The Innovative Integration Zuma Toolset allows users of II DSP processor boards to develop complete executable applications suitable for use on the target platform. The environment suite consists of the TI Optimizing C Compiler, Assembler, and Linker, the Code Composer debugger as well as II's custom Windows applets (such as the `TERMINAL.EXE` terminal emulator) plus the Codewright code authoring environment.

Codewright is the default package is used to automate executable build operations within Innovatives Zuma Toolsets, simplifying the edit-compile-test cycle. Source is edited, compiled, and built within Codewright, then downloaded to the target and tested within either the Code Composer debugger or via the Zuma terminal emulator.

On C6x platforms, such as Innovatives M6x, SBC6x and Quatr6x, Code Composer Studio may be used instead of Codewright for both code authoring and code debugging. Details of constructing projects for use on Innovative DSP platforms using Studio are provided in this chapter.

Do not confuse the creation of target applications (code running on the target DSP processor) with the creation of host applications (code running on the host platform). The TI tools generate code for the TI DSP processors, and are a separate toolset from that needed to create applications for the host platform (which would consist of some native compiler for the host processor, such as Microsoft's Visual C++ or Borland Builder C++ for IBM compatibles). To create a completely turn-key application with custom target and host software, *two* programs must be written for *two* separate compilers. While II supports the use of Microsoft C/C++ for generation of host applications under Windows with sample applications and libraries, we do not supply the host tools as part of the Development Environment. For more information on creating host applications, see the section in this manual on host code development.

This section supplies information on the use of the development environment in creating custom or semicustom target DSP software. It is not intended as a primer on the C language. For information on C language basics, consult one of the C primer books available at your local bookstore. The definitive reference to the C language is The C Programming Language, by B. Kernighan and D. Ritchie (Prentice Hall. Englewood Cliffs, NJ. 1988).

#### Components of Target Code (.c, .asm, .cmd)

In general, DSP applications written in TI C require at least two files: a .c file (or "source" file) containing the C source code for the application, and a .cmd file (or "linker command" file) which contains the target-specific build data needed by the linker. There may also be one or more .asm assembler source files, if the user has coded any portions of the application in assembly language.

---

### *Edit-Compile-Test Cycle using Codewright*

Nearly every computer programming effort can be broken down into a three step cycle commonly known as the edit-compile-test cycle. Each iteration of the cycle involves editing the source (either to create the original code or modify existing code), followed by compiling (which compiles the source and creates, or builds, the executable object file), and finally downloading and testing the result to see if it functions in the desired fashion. In the TI development system, these stages are accomplished within the Codewright editor and Code Composer debugger programs. Codewright supports the editing and compilation stages, and Code Composer allows the executable result to be downloaded and tested on the target hardware.

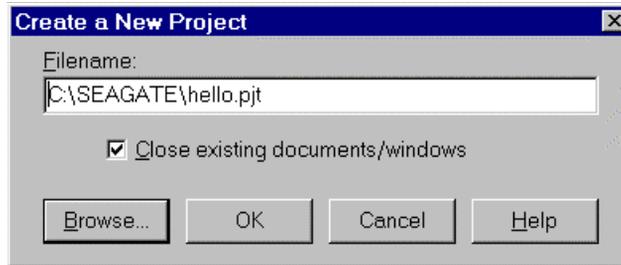
Codewright is a full-featured programmer's editor whose functionality has been extended using a custom DLL to allow the TI compiler, assembler, and linker to be called from within the editor, making the environment more user-friendly than the basic command line interface which comes standard with the TI tools.

---

### *A Simple Codewright Project*

The following sequence illustrates the creation of a project to build the `Hello World!` program from within Codewright.

First, start Codewright. Select Project | New from the Project menu and you will see the following dialog:

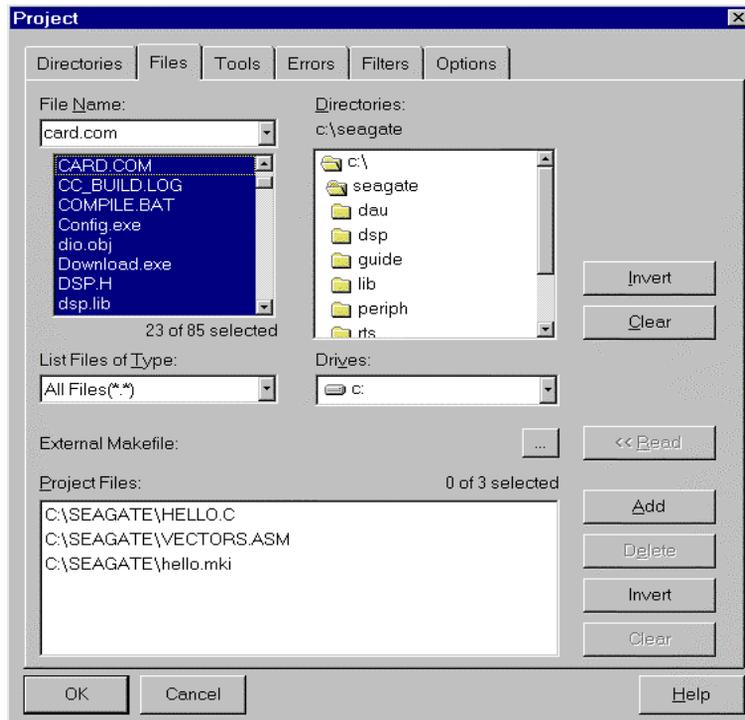


**FIGURE 16.** Creating a new project in Codewright

---

Browse to the directory in which you would like to create the new project (your working directory) and then type the name of the new project. In this example, the working directory is `c:\seagate` and the project name is `hello.pjt`. In the standard developers package, you could browse into the `%II_BOARD%\EXAMPLES\TARGET` directory (Note: Substitute actual Zuma root directory for `%II_BOARD%`).

Next Codewright will open the Add Files dialog. Add the `HELLO.C` and the automatically-created `HELLO.MKI` files to the project.



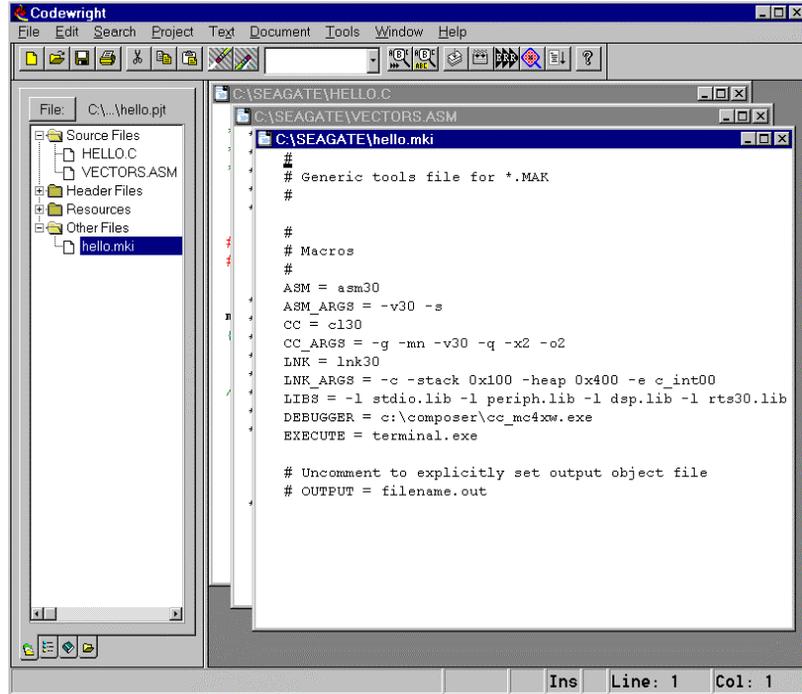
**FIGURE 17. Adding files to a Codewright project**

---

If you do not add a command file to the project, the `generic.cmd` file located in the `II_BOARD` directory will be assumed. That is, the memory map for the target DSP specified in the `generic.cmd` file will be used to link the project output file. If you do include a command file in the project, it will override the settings in the `generic.cmd` file in the `II_BOARD` directory.

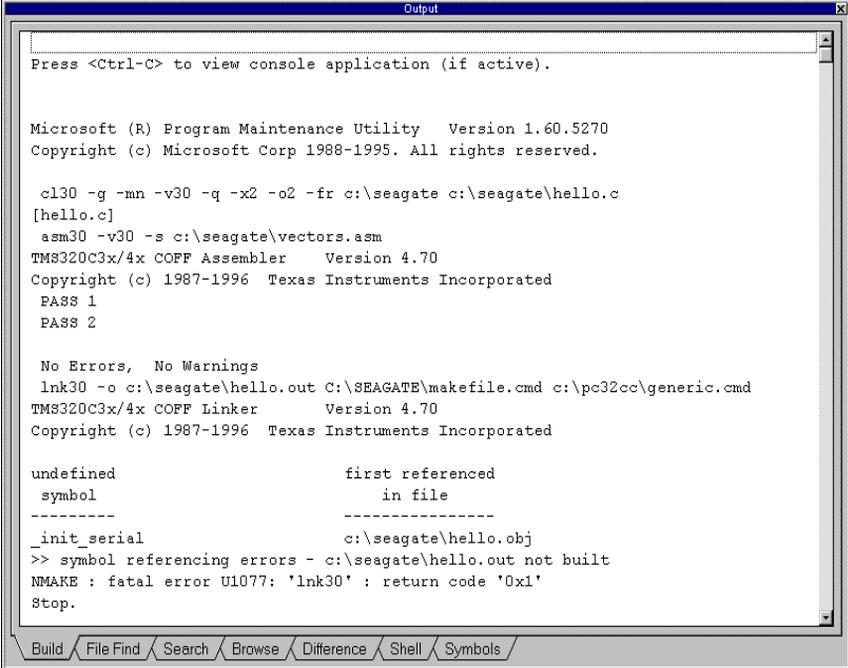
When you have finished adding files, click **OK**.

Next, you may optionally open the files in the project by double-clicking on their names within the Project window.



**FIGURE 18. Codewright Project Window.**

Now, to build the hello.out file, simply click on the Build icon on the Codewright toolbar. Compiler progress is shown in the Output window:



The screenshot shows a window titled "Output" with a scrollable text area. The text inside the window is as follows:

```
Press <Ctrl-C> to view console application (if active).

Microsoft (R) Program Maintenance Utility  Version 1.60.5270
Copyright (c) Microsoft Corp 1988-1995. All rights reserved.

cl30 -g -mn -v30 -q -x2 -o2 -fr c:\seagate c:\seagate\hello.c
[hello.c]
asm30 -v30 -s c:\seagate\vectors.asm
TMS320C3x/4x COFF Assembler  Version 4.70
Copyright (c) 1987-1996  Texas Instruments Incorporated
PASS 1
PASS 2

No Errors, No Warnings
lnk30 -o c:\seagate\hello.out C:\SEAGATE\makefile.cmd c:\pc32cc\generic.cmd
TMS320C3x/4x COFF Linker  Version 4.70
Copyright (c) 1987-1996  Texas Instruments Incorporated

undefined          first referenced
symbol             in file
-----
_init_serial       c:\seagate\hello.obj
>> symbol referencing errors - c:\seagate\hello.out not built
NMAKE : fatal error U1077: 'lnk30' : return code '0x1'
Stop.
```

At the bottom of the window, there is a toolbar with buttons for "Build", "File Find", "Search", "Browse", "Difference", "Shell", and "Symbols".

**FIGURE 19. Codewright compiler progress in output window**

If errors are encountered in one or more source files, they are listed in the output window and you may view and repair each error by either clicking on each error in the Output window or by clicking the Err icon on the Codewright toolbar.

### Automatic makefile creation

When a project is created, opened, modified, built or rebuilt, the dependency DLL automatically generates a project makefile (named *<project file>.mk*, located in the project directory) which is capable of rebuilding the project's output file from its components. An example makefile is shown below.

```
#
# Innovative Integration, Inc
```

```
# Auto-generated makefile
#

#
# Clear suffix list & use a new one
#
.SUFFIXES:
.SUFFIXES: .out .obj .c .asm .cmd .lib

#
# Macros
#
RESPONSE = Log.rsp
CMD = ..\..\generic.cmd

#
# Generic tools file for *.MAK
#

#
# Macros
#
ASM = asm30
ASM_ARGS = -v30 -s
CC = cl30
CC_ARGS = -g -mn -v30 -q -x2 -o2
LNK = lnk30
LNK_ARGS = -c -stack 0x800 -heap 0x400 -e c_int00
AR = ar30
AR_ARGS = -r
LIBS = -l stdio.lib -l periph.lib -l dsp.lib -l rts30.lib
DEBUGGER = c:\composer\cc_mc4xw.exe
# EXECUTE = terminal.exe

# Uncomment to explicitly set output object file
# OUTPUT = filename.out

OUTPUT_BASE = Log
OUTPUT = Log.out
#
# Files for project
#
MAKEDEPSRC_ASM = ..\..\vectors.asm
```

```
MAKEDEPSRC = log.c scale.c scan.c solve.c

#
# Targets and dependencies
#
build: $(OUTPUT)
log.obj: log.c ..\command.h dau.h log.mki $(FRC)
scale.obj: scale.c dau.h log.mki $(FRC)
scan.obj: scan.c ..\..\INCLUDE\TARGET\periph.h
..\..\INCLUDE\TARGET\stdio.h\
dau.h log.mki $(FRC)
solve.obj: solve.c ..\..\INCLUDE\TARGET\dsp.h
..\..\INCLUDE\TARGET\periph.h\
..\..\INCLUDE\TARGET\stdio.h dau.h log.mki $(FRC)
..\..\vectors.obj: ..\..\vectors.asm log.mki $(FRC)
rebuild:
$(MAKE) build FRC=force_rebuild -f Log.mk
force_rebuild:

#
# Build rules
#
$(OUTPUT_BASE).out : $(MAKEDEPSRC:.c=.obj)
$(MAKEDEPSRC_ASM:.asm=.obj) \
$(CMD)
$(LNK) -o @$ $(RESPONSE)
$(OUTPUT_BASE).lib : $(MAKEDEPSRC:.c=.obj)
$(MAKEDEPSRC_ASM:.asm=.obj)
!$(AR) $(AR_ARGS) @$ $?

#
# Inference rules...
#
.c.obj:
$(CC) $(CC_ARGS) -fr $(<D) $<

.asm.obj:
$(ASM) $(ASM_ARGS) $<
```

**FIGURE 20. An example of an auto-generated makefile**

---

This file is automatically submitted to the make facility whenever you click on build or rebuild within Codewright. The make facility automatically constructs the output file by recompiling the out-of-date source files including the dependencies contained within those source files.

## **Rebuilding a Project**

It is sometimes necessary to force a complete rebuild of an output file manually, such as when you change optimization levels within a project's mki file. To force a project rebuild, select Project | Rebuild from Codewright menu bar.

## **Running the Target Executable**

The `hello` program is very simple, only printing the single line "Hello, World" to the terminal emulator before waiting for a key and exiting. Scroll down the source file by using cursor down until you reach the call to `printf()`, which looks like the following:

```
printf("Hello, World\n");
```

Change the output string to read "Hello, Brave New World\n". You can now compile the new version by executing Build from the Project menu (or by clicking on its toolbar icon). This causes Codewright to start the compiler, which produces an assembly language output. The compiler then automatically starts the assembler, which produces a `.obj` output file (`hello.obj`). Codewright then invokes the TI Linker using the `generic.cmd` file, which is located in the root board directory. This rebuilds the executable file using the newly revised `hello.obj`. If no errors were encountered, this process creates the downloadable COFF file `hello.out`, which can be run on the target. At this point, the program may be run using the Terminal Emulator applet, which may be automatically invoked from within Codewright using the Project | Execute menu item. The program runs and outputs the message "Hello, Brave New World" to the terminal emulator window.

If errors are encountered in the process, Codewright detects them and places them in the Output window. If the error occurred in the compiler or assembler (as in a C

syntax error), the cursor may be moved to the offending line by simply double-clicking on the error line within the output window, and the error message will be displayed in the Codewright status bar. If the linker returns a build error, the output window shows the error file. From this information, the linker failure can be determined and corrected. For example, if a function name in a call is misspelled, the linker will fail to resolve the reference during link time and will error out. This error will be displayed on the screen in the Output window.

This outlines the basics of how to recompile the existing sample programs. The following section explains why the program is structured the way it is and what function each component is performing.

---

### *Anatomy of a Target Program*

While not providing much in the way of functionality, the `hello` program does demonstrate the code sequence necessary to properly initialize the target. The exact coding, however, is very specific to the IIC Development Environment and target boards and is explained in this section in order to acquaint developers with the basic syntax of a typical application program.

Here we examine the SBC31 version of the `hello` program. Although the source is not necessarily identical to that of `hello` for the other targets, it is typical of the overall structure of the typical application program designed under the development environment.

```
/*
    HELLO.C
    Classic K&R Hello program.
*/
#include "stdio.h"
#include "periph.h"
main()
{
    int key;
    enable_cache();
    enable_monitor();
    enable_interrupts();
}
```

```
        clrscr();
        printf("Hello World!\n");
        do
        {
            key = getchar();
            putchar(key);
        }
        while(key != ESC);

        monitor();
    }
```

The first two lines of the program are `#include` statements which include the header files for the peripheral and standard I/O libraries. These include prototypes for all the library routines as well as variable definitions and `#define` statements for the peripheral memory-mapping addresses. These `#defines` are especially important for those who wish to perform direct peripheral access, rather than using the peripheral libraries.

The call to `enable_cache()` enables the internal CPU cache. Though HELLO.C is certainly not a performance application, the next line sets the `MHZ` global variable to the appropriate processor speed for this card. Properly setting this variable is necessary since the library routines which handle the timers depend on having the correct processor speed available so that the timer period registers can be set correctly. Although the timers are not used in this application, it is good form to include this line in all PC44 programs.

Next, global interrupts are enabled with a call to `enable_interrupts()`. This routine merely sets the global interrupt enable bit in the 'C44 status register, which allows any interrupts not locally masked to become pending on the CPU.

Once interrupts are globally enabled, the monitor interrupt is unmasked using `enable_monitor()`. This routine globally enables interrupts, permitting serial port interrupts to propagate through to the DSP chip.

The next two lines perform the standard I/O function of the program, clearing the terminal emulation screen and printing "Hello, World". These two lines are where custom code should be inserted.

The following `getchar()` call simply echoes keys typed at the terminal emulator back to the terminal display. This routine is also part of the standard I/O library. The program effectively terminates here, except that interrupts are still active and interrupt handlers (if they had been installed) would still execute properly.

As is shown, the `hello` program is very simple, but it exhibits the basic functionality needed to properly start on the CPU, as well as the initialization needed to interact with `Code Composer` and the terminal emulator properly in the development environment.

### Use of Library Code

Library routines can be compiled and linked into your custom software simply by making the appropriate call in the source and adding the appropriate library to the linker command file. Refer to the library reference in this manual for library location information on each function.

In general, user software needs to `#include` the relevant library header file in source code. The header files define prototypes for all library functions as well as definitions for various data structures used by the library functions. The file `stdio.h` should be included by programs using the standard I/O library, and the file `periph.h` should be included if a program uses functions in the peripheral library. The function definitions in the peripheral library reference note which library a particular function lives in, as well as the header file which should be included for that function.

### Compiling/Assembling/Linking Outside Codewright

Under certain circumstances, it may not be possible to use Codewright's macro definitions to compile inside the editor. If the user has a large number of TSR programs or device drivers loaded under DOS, there may not be enough DOS memory available for Codewright to shell off to the compiler, assembler, or linker. If this situation comes up, the batch programs normally used by Codewright to perform compilation, assembly, and linking may be executed directly from DOS. `COMPILE.BAT`, `ASSEMBLE.BAT`, and `LINK.BAT` are provided in the `%II_BOARD%` directory and may be executed by typing their names followed by the source file on which they are to operate. For example, the file `mycode.c` can be compiled by typing

**compile mycode**

at the DOS prompt. This causes the `COMPILE.BAT` script to start, which runs the compiler and generates the file `mycode.obj`, assuming no errors occurred. The `COMPILE.BAT` script also searches for the file `mycode.cmd` in the current directory. If the linker command file is found, then the linker is automatically run and the entire executable linked. If the command file is not found, processing stops with the generation of `mycode.obj`.

Assembly source (`mycode.asm`) may be assembled by typing

**assemble mycode**

where the assembler is called and an object file generated.

Linking can also be performed. In this case the input file is not source code, but a linker command file (`mycode.cmd`):

**link mycode**

This line causes the linker to build the executable `mycode.out`, again assuming no errors have occurred during the process. Also note that the `COMPILE.BAT` script will automatically link the executable if a linker command file of the same name exists.

In all cases, if any errors occur, an error file (`mycode.err`) is generated by the tools. It contains the full console output of each of the tools, and any error generated by any tool will be recorded in this file.

**Compiling without a Project**

Occasionally, during program development it is useful to generate and compile code without constructing a project – just to verify proper syntax, etc. When compiling or assembling programs in Codewright without an open project, the default compiler/assembler command lines and arguments are derived from the `generic.mki` file in the `II_BOARD` directory. Therefore, to compile or assemble a single file, simply open the file and click on the Compile button on the toolbar. It will be built using the switches specified in `generic.mki`.

## Building Libraries

The makefile generated by the dependency DLL is capable of building either executable targets or linker-compatible libraries. By default, the .mki file for a project does *not* specify a name for the target output file. The default target output file name is, therefore, constructed from the base of the project name plus the .out extender. For example, if a new project called EXAMPLE.PJT is created, the default name for the output file would be EXAMPLE.OUT. This can be overridden in the .mki file by un-commenting the line containing OUTPUT = macro and specifying the desired target filename on that line.

If the name of the newly-specified output file has an .out extender, the target is assumed to be an executable and the makefile will attempt to link the executable during a build operation. However, if the target file is a library file with the .lib extension, the makefile will use the archiver tool to add rebuilt components of the project to the library file specified by the OUTPUT = macro. This property can be very helpful when building and maintaining libraries.

Whenever changing the default OUTPUT = macro within an mki file, be sure to reconstruct the project makefile by executing Project | Makefile within Codewright.

---

## *The Next Step: Developing Custom Code*

In building custom code for an application, II recommends that you begin with one of the sample programs as an example, extending it to serve the exact needs of the particular job. Since each of the example programs illustrates a basic data acquisition or DSP task integrated into the target hardware, it should be fairly straightforward to find an example which roughly approximates the basic operation of the application. For example, if the task calls for digital recording of analog information, look to `echo` or `loopback`. If the task involves filtering, `fir` would be a good starting point. And if the application needs to pass data to/from the host processor, `dualport` should be studied carefully. Familiarize yourself with the sam-

ple programs: they should provide the skeleton for the fully custom application, and ease a lot of the target integration work by providing hooks into the peripheral libraries and devices themselves.

---

### *Edit-Compile-Test Cycle using Code Composer Studio*

Nearly every computer programming effort can be broken down into a three step cycle commonly known as the edit-compile-test cycle. Each iteration of the cycle involves editing the source (either to create the original code or modify existing code), followed by compiling (which compiles the source and creates, or builds, the executable object file), and finally downloading and testing the result to see if it functions in the desired fashion.

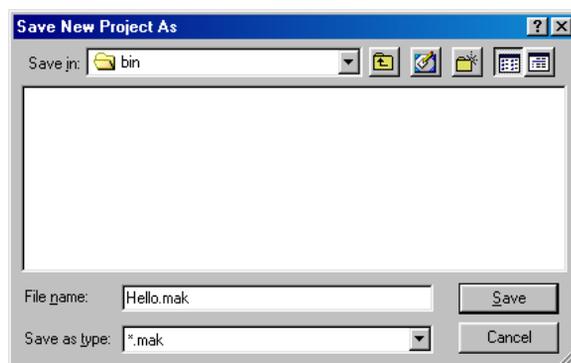
When using Studio, these stages are accomplished entirely within the Studio integrated. The project features of Studio support the project and component file editing and compilation stages, and Code Composer allows the executable result to be downloaded and tested on the target hardware.

---

### *A Simple Studio Project*

The following sequence illustrates the creation of a project to build the `Hello World!` program from within Studio.

First, start Studio. Select `Project | New` from the Project menu and you will see the following dialog:

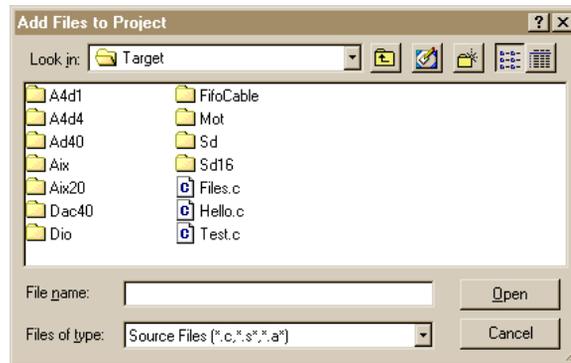


**FIGURE 21.** Creating a new project in Studio

---

Browse to the directory in which you would like to create the new project (your working directory) and then type the name of the new project. In this example, the working directory is `c:\ti\bin` and the project name is `hello.mak`. In the standard developers package, you could browse into the `%II_BOARD%\EXAMPLES\TARGET` directory.

Next open the Project | Add Files to Project dialog. Add the `HELLO.C` and the `GENERIC.COMD` files to the project.



**FIGURE 22.** Adding files to a Studio project

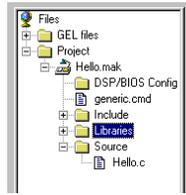
---

It is imperative that you add an appropriate command file to the Studio project. The generic.cmd command file describes the memory map of the target hardware, without which the linker will be unable to place executable sections into appropriate memory regions for debugging. That is, the memory map for the target DSP specified in the generic.cmd file will be used to link the project output file. If you wish, you may copy the contents of the generic.cmd file (located in the root of the Zuma toolset) into your working directory, rename it appropriately and add the modified cmd file to your project instead.

Do not add any library files directly into the project. Rather, manually type the desired libraries needed to link the project into the Project | Options | Linker tab when instructed to do so later within this chapter.

When you have finished adding files, click **OK**.

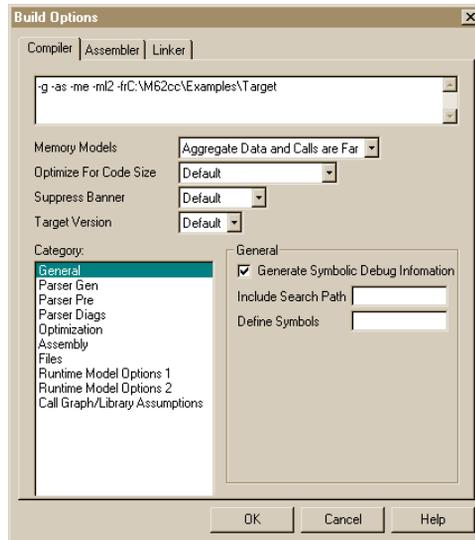
Next, you may optionally open the files in the project by double-clicking on their names within the Project window.



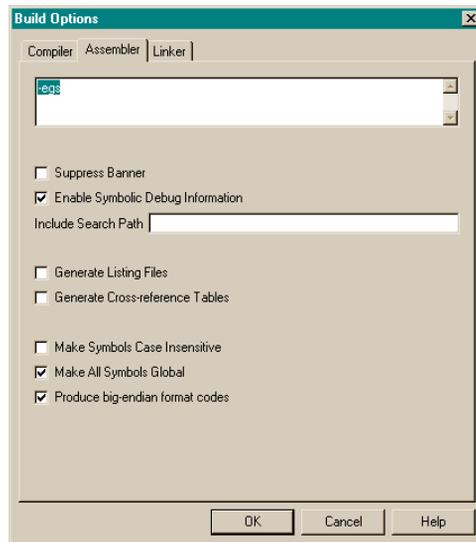
**FIGURE 23. Studio Project Window.**

Next, you must configure the project compiler settings so that when `Hello.c` is compiled, the appropriate memory model and switches are used. Click on Project | Options to open the Build Options dialog then click on the Compiler Tab to show the current compiler options. Configure the compiler to use the following settings:

Aggregate Data and Calls are far (-ml2 memory model)

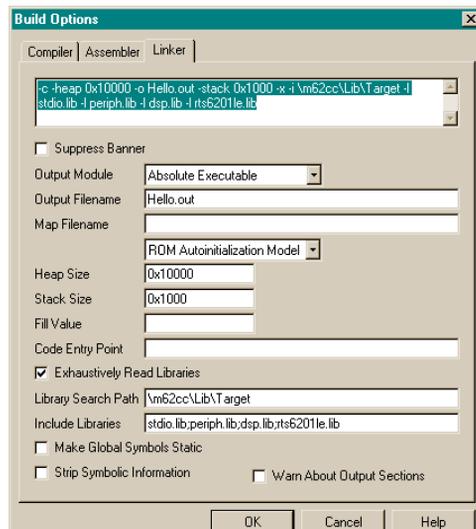


Next, click on the Assembler Tab.

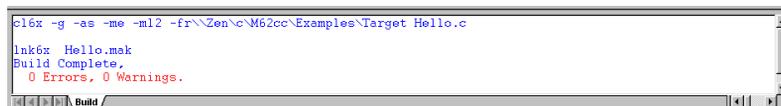


Configure the Assembler to generate Big Endian target code, to make all symbols global and to enable Symbolic Debug Information

Finally, click on the Linker Tab



Add `stdio.lib`; `periph.lib`; `dsp.lib`; and `rts62011e.lib` into the Include Libraries edit box (in that order). Enable Exhaustively Read Libraries. Set the Heap size to 0x10000 bytes and the stack size to 0x1000 bytes.



```
E16x -g -as -me -ml2 -fr\\Zen\\c\\M62cc\\Examples\\Target Hello.c
lnk6x Hello.mak
Build Complete,
0 Errors, 0 Warnings.
```

**FIGURE 24. Studio compiler progress in output window**

---

If errors are encountered in one or more source files, they are listed in the output window and you may visit and repair each error by either clicking on each error in the Output window.

### Automatic makefile creation

When a project is created, opened, modified, built or rebuilt, the Studio dependency generator automatically generates a project makefile (named *<project file>.mak*, located in the project directory) which is capable of rebuilding the project's output file from its components.

This file is automatically submitted to the internal make facility whenever you click on build or rebuild within Studio. The make facility automatically constructs the output file by recompiling the out-of-date source files including the dependencies contained within those source files.

### Rebuilding a Project

It is sometimes necessary to force a complete rebuild of an output file manually, such as when you change optimization levels within a project. To force a project rebuild, select Project | Build All from the Studio menu bar.

### Running the Target Executable

The `hello` program is very simple, only printing the single line “Hello, World” to the terminal emulator before waiting for a key and exiting. Scroll down the source

file by using cursor down until you reach the call to `printf()`, which looks like the following:

```
printf("Hello, World\n");
```

Change the output string to read “Hello, Brave New World\n”. You can now compile the new version by executing **Build** from the Project menu (or by clicking on its toolbar icon). This causes Studio to start the compiler, which produces an assembly language output. The compiler then automatically starts the assembler, which produces a `.obj` output file (`hello.obj`). Studio then invokes the TI Linker using the `generic.cmd` file, which is located in the root board directory. This rebuilds the executable file using the newly revised `hello.obj`. If no errors were encountered, this process creates the downloadable COFF file `hello.out`, which can be run on the target. At this point, the program may be run using the Terminal Emulator applet, which may be invoked using the Terminal shortcut located within the program group created during the Zuma Libraries installation process. The program runs and outputs the message “Hello, Brave New World” to the terminal emulator window.

If errors are encountered in the process, Studio detects them and places them in the Output window. If the error occurred in the compiler or assembler (as in a C syntax error), the cursor may be moved to the offending line by simply double-clicking on the error line within the output window, and the error message will be displayed in the Studio status bar. If the linker returns a build error, the output window shows the error file. From this information, the linker failure can be determined and corrected. For example, if a function name in a call is misspelled, the linker will fail to resolve the reference during link time and will error out. This error will be displayed on the screen in the Output window.

**Note:** Be sure to start the terminal emulator **BEFORE** starting Studio, to avoid resetting the DSP target in the midst of the debugging session. If Terminal is not yet running and you wish to run the Hello object file, perform the following steps.

1. Execute **Debug | Run Free** to logically disconnect the DSP from the debugger software
2. Terminate the Studio application
3. Invoke the Terminal Application

**4. Restart the Studio application**

This outlines the basics of how to recompile the existing sample programs within the Studio environment.



---

This section describes the Innovative Integration Windows host software development environment. The environment provides complete support for generating 32-bit Windows-compatible software which is capable of controlling and communicating with I.I.'s DSP coprocessor and data acquisition cards. Virtual device drivers (Windows 95 VxD or NT Kernel Mode Driver) and dynamic link libraries (DLL) are included to provide an easy-to-use, portable low-level interface for the target hardware, while sample applications show how to call the DLL functionality and present basic interface examples and guidelines on processor card control requirements and data movement.

Host software development is directly supported under the Microsoft Visual C/C++ 4.0 environment for generating 32-bit Windows applications. All example application programs included in the development package are supplied with Visual C workspace files, making program modification and regeneration as simple as possible.

Please Note: only Windows application development is currently supported by the Developer's Package. Foreign operating systems, such as Unix and OS9 are not currently supported.

---

## *Dynamic Link Library*

All target interaction takes place through calls to the supplied dynamic link library (DLL). This library supplies low-level functions for basic target board control, including processor reset/run state, message passing via the board-specific mailbox registers, application downloading, and bus master memory locking and access control.

The function calls available under the DLL are documented in the appendices. Sample applications

(described below) provide working examples on how to interact with the card via host software.

### **Sample Host Programs**

The DLL is capable of interacting with up to four target DSP boards simultaneously by default (contact II if more than four targets are required). The DLL maintains a board-specific structure of information for each target, known as the `cardinfo` structure. An prototype of the `cardinfo` structure is located in the `\INCLUDE\HOST\` subdirectory in the `CARDINFO.H` file. An example is shown below.

```
//
//   cardinfo.h   --   definition of CARDINFO structure
//

#ifndef __CARDINFO_H__
#define __CARDINFO_H__

#include "ii_iostr.h"           // Common IO Driver/DLL Structures
#include "mailbox.h"           // Definition of MAILBOX structures

//
//   BoardInfo structure
//
typedef struct _BoardInfo
{
    ULONG           ProcessorCount;
};
```

```
        ULONG        DLL_Version;        // Version ID numbers
        ULONG        DrvVersion;
        ULONG        TalkerVersion;
        ULONG        CellSize;          // Targ memory cell size, in bytes
        ULONG        CtlReg;            // Shadow of control register
        ULONG        FlashSectorSize;   // Size of flash sectors, in bytes
        ULONG        FlashDeviceId;     // Flash device ID
        ULONG        QuietMode;         // Don't Display Messages if true
    } BoardInfo;

//
// InterruptInfo structure
//
typedef struct _InterruptInfo
{
    ULONG        IRQ;                // IRQ of attached interrupt
    HANDLE       Ring0Event;         // Ring 0 event handle
    HANDLE       Ring3Event;         // Ring 3 event handle
    void         (*Vector)(void *); // Virtual ISR function pointer
    void *       Context;           // Virtual ISR context pointer
} InterruptInfo;

//
// SerialInfo structure
//
typedef struct _SerialInfo
{
    LONG         In;                // Buffer for last character received
    LONG         ReadFlag;          // True when character received
    LONG         MbValue;           // Multi-byte value
    LONG         MbCtr;             // Multi-byte read state
    ULONG        RTS_state;         // Current state of the RTS output
    LONG         Bcr;               // Bus control register value for Flash access
    LONG         Reading;           // TRUE if currently reading a character
    OVERLAPPED   RxOverlap;         // Info used in asynch input
    OVERLAPPED   TxOverlap;         // Info used in asynch output
    COMMTIMEOUTS Timeouts;         // Info for set/query time-out parameters
    DCB          Dcb;               // Device control block
} SerialInfo;

//
// CARDINFO structure
//
typedef struct _cardinfo
```

```
{
    ULONG           Target;           // Number of current target
    HANDLE          Device;           // Handle to Driver for device
    BoardInfo       Info;             // Board Info
    MAILBOX *       Mail;             // Talker Mailbox Array
    IoPortBlock     Port;             // Primary Port Block Information
    IoPortBlock     OpReg;            // Secondary Port Block Information
    MemoryBlock     DualPort;         // Shared Memory Area Information
    MemoryBlock     BusMaster;        // BusMaster Memory Information
    nterruptInfo    Interrupt;        // Interrupt Information
    SerialInfo      Serial;           // Serial Port I/O (SBC's)
} CARDINFO;

#endif
```

The `cardinfo` structure is accessed within Host application programs in order to gain access to board-specific parameters which are maintained by the DLL. For example, in order to ascertain the size of the shared memory area on a specific target card a host program could use:

```
/* send bus mastering physical address to target processor */

dsp = (CARDINFO*)target_cardinfo(target);

size = dsp->Dualport.Size;
```

### Sample Host Programs

Each Zuma Toolset is supplied with one or more example programs which illustrates control of the DSP board via the supplied DLL. For bus-based boards, the example is `SCOPE.C`, which emulates a simple oscilloscope. For stand-alone boards, the `XRPT.C` example is provided, which illustrates advanced serial communications. The `SCOPE` example

`SCOPE.C` is a small, working example written in Visual C v4.0 showing how to use bus-based DSP boards to move data between the target and Host memory spaces. The host application works in concert with a small DSP program running on the target to mimic the operation of a simple oscilloscope.

The `SCOPE` application is included in the `\EXAMPLES\HOST\SCOPE` subdirectory of `%II_BOARD%`. Its executable is located in the `\EXAMPLES\HOST\SCOPE\RELEASE` subdirectory. The DSP support code for this application is located in the `\EXAMPLES\HOST\SCOPE\DSP` subdirectory.

SCOPE.C is a multi-threaded application example with three threads. The primary thread performs Window management, including the Windows message handler. A second thread, `EnqueueData()` handles data movement from the target DSP to the Host using shared memory (dual port memory on ISA bus cards and bus master memory on PCI cards). The third thread, `PlotData()`, plots the enqueued data received from the target within the window.

This program illustrates many of the elements of a typical Host application which communicates with a target DSP application. In this example, this Host program communicates closely with the `SCOPE.C` DSP application located in the `\EXAMPLES\HOST\SCOPE\DSP` directory. `SCOPETRG.C` is the code which runs on the target DSP and which is responsible for feeding information to the Host via shared memory.

When the Host program starts, it invokes the COFF downloader to download the object image of the target DSP application (`SCOPETRG.OUT`) within the `download()` procedure. This procedure makes calls on the DLL in order to effect the download. Following the download, the application is started running using the `start_app()` function. The DSP application immediately begins generating mock analog data in order to emulate acquiring data from the analog subsection on the target, enqueueing the acquired data. As soon as a packet-full of data is available, the data is dequeued by the target, moved into shared memory and the Host program is signalled, using the `host_interrupt()` target procedure.

The Host device driver handles the target interrupt signal and issues an special EVENT message to the ring three DLL which performs a callback on the user-installed `Host EnqueueData()` function. When this occurs, the offset into dual port memory containing the new packet of analog samples is read from the shared memory. This address is used to enqueue data from shared memory area into a Host-maintained data queue of real-time analog samples.

The Host `PlotData()` thread draws an oscilloscope-like grid on the display window, then polls continuously for the availability of analog samples in the Host queue. When a screenfull of data is available in the queue, it is dequeued and plotted.

The primary thread is responsible for handling window messages only. The most typical window messages are invoked when the users drags or resizes the oscilloscope window. When this occurs, the `WM_SIZE` message handler sets the global variable `refresh` TRUE, which indicates to the `PlotData()` thread that a complete window update is needed. The `PlotData()` function temporarily drops out

of the data plotting loop in order to redraw the oscilloscope display. Then, it resume the plotting function again, until the refresh variable is modified again.

### **The XRPT example**

`XRPT.C` is a small, working example written in Visual C v4.0 showing how to use serial-based DSP boards to move data between the target and Host memory spaces. The host application works in concert with a small DSP program running on the target to tally the number of target-to-host interrupts signalled by the DSP during application execution. Like the `SCOPE` example for bus-based DSPs, `XRPT` illustrates installation of a Host interrupt handler using DLL calls. This interrupt handler is invoked by the target DSP via the `host_interrupt()` function call, which in the case of single-board targets initiates a delta-CTS interrupt to the Windows device driver, which signals and event to the `II` DLL which calls back the user-installed interrupt function.

---

Software is created for the target DSP by using one or more of the tools included in the Developer's Package, alone or in concert with each other, to generate a downloadable executable COFF format file which can be run on the target DSP board with the aid of the utilities included in the package.

This section of the *Developer's Package Software Manual* details the use of the individual tools in the package to create executables for the target, and gives step-by-step instructions on how to use the C compiler and Code Composer to write, compile, test, and debug custom C applications on the target. Sample C applications are also discussed

---

## *C Code Development*

### **C Compiler**

The Texas Instruments C compiler is an ANSI C compatible compiler which produces optimized assembly code for the TMS320C4x family of processors. A complete set of manuals is included with the M62 Developers Package.

In addition to the excellent manuals from TI, refer to the Kernighan and Ritchie C Handbook (available at cost from I.I.) for generic C questions and syntax. The TI manuals primarily describe the use of the compiler with the TMS320C4x family and are not intended as C primers for the beginner.

## **C Library Reference**

Complete source code to the entire suite of ANSI C libraries is provided with the C system to aid in code development. Refer to pages 5-10 through 5-18 of the *TMS320 Floating Point DSP Optimizing C Compiler Manual* for a complete list of TI C functions.

The I.I. M62 Developer's System also includes extensive high-level libraries useful in interacting with the various peripherals on the M62 board. The following sections describe by peripheral type the functions provided in the peripheral library. For a complete alphabetical listing of all peripheral functions, see Appendix I.

## **M62 Zuma Toolset Libraries**

The Zuma toolset provides both target peripheral libraries and Host DLLs and numerous example programs to illustrate usage.

The peripheral libraries for the M62 provide support for the on-board peripherals and terminal I/O functions. The libraries are provided in three linkable .LIB files: `PERIPH.LIB`, `STDIO.LIB` and `DSP.LIB`. `STDIO.LIB` holds all the console terminal emulation and communications routines listed in the following section, while `PERIPH.LIB` contains all other peripheral driver routines. `DSP.LIB` contains commonly requested C-callable digital signal processing functions, plus common math and queue management extensions. Source code for the routines is also provided, arranged by function in the `\PERIPH`, `\STDIO` and `\DSP` subdirectories of the root `II_BOARD` directory, as follows:

Directory	Library Source
\DSP	Standard Digital Signal Processing Routines
\PERIPH\ANALOG	Drivers for the M62 A4D4 instrumentation-grade analog I/O module and the complementary TERM mux module.
\PERIPHERAL\BUS	Drivers for the SD high-performance audio module
\PERIPH\DIGITAL	Drivers for V360 bus-mastering PCI interface
	Digital I/O, PIT Timer control, module FLASH ROMs, etc.
	Drivers for DIO module
	Drivers for MOT motion control module
\PERIPHERAL\DIO	DIO module DUART and digital I/O drivers
\PERIPH\MISC	Miscellaneous processor control and data conversion functions
\PERIPH\RTS	Modified boot-up routines for the M62 baseboard.
\STDIO	Console and terminal emulation functions
\TALKER	Startup umbilical $\tau$ C6201 software

**TABLE 12. Zuma Toolset Source Directories**

The toolset also contains various support files arranged as described below.

Directory	Library Source
\EXAMPLES\HOST	Example programs illustrating use of the DLL to control the DSP board from within MS Visual C programs.
\EXAMPLES\TARGET	Example programs illustrating use of the target peripheral libraries to perform common DSP tasks.
\INCLUDE\HOST	Header files used by Host Visual C programs
\INCLUDE\TARGET	Header files used by target Texas Instruments C programs
\LIB\HOST	Linkable library files for Host Visual C and C++ programs
\LIB\TARGET	Linkable library files for target TI C and assembler programs
\SRC	Useful public domain source files for the C6201 processor.

**TABLE 13. Zuma Toolset support subdirectories**

**STDIO Console Terminal Driver.** The Developer's Package contains a full-featured terminal emulator application (terminal.exe), suitable for both user interface purposes as well as debugging use. The peripheral library provides a complete set of standard I/O routines which can communicate directly with this terminal emulator. The source for the standard I/O routines is given in the \STDIO subdirectory under the installation directory. In general, the standard I/O library functionality is identical to that of the K&R standard I/O library. However, some M62-specific functions are provided to allow higher level functionality such as cursor positioning, text attribute control and graphical data plotting. The following target programming section gives details on how to use the standard I/O peripheral library to interact with the terminal emulator.

**Digital Peripheral Drivers.** The digital peripheral drivers control the 'C6201 internal timers and the digital I/O lines, allowing high-level access to timebase control functions and digital I/O activity without doing direct hardware programming. The following target programming section gives details on how to use the digital peripheral library to program the digital peripherals. Source code for the functions is given in the `\PERIPH\DIGITAL` directory.

**BUS Peripheral Drivers.** The BUS peripheral drivers provide control functions for the onboard V360 PCI bus interface. The available routines support very-high speed bus-mastering transfers between the 512 Kbyte, external async SRAM of the M62 and host PC memory. This driver also includes hardware mailbox support routines which are used extensively by the standard I/O library in order to support terminal emulation. Additionally, these mailbox routines provide a means of performing interrupt-driven communications with the Host PC. The target programming section gives details on how to use the bus peripheral library. Source code for the functions is given in the `\PERIPH\BUS` directory.

**Miscellaneous Peripheral Drivers.** The MISC directory contains code to support high-level access to the internal registers, byte packing and unpacking, interrupt vector support, and other functions. Source code for the functions is given in the `\PERIPH\MISC` directory.

**RTS Peripheral Drivers.** The RTS peripheral drivers provide board-specific versions of the functions called by the TI C Compiler during coldstart initialization of the C runtime engine. These files have been modified as necessary in order to provide a complete initialization of the M62 onboard hardware immediately prior to calling `main()` within application code. Additionally, the RTS functions include a modified version of the millisecond timer function required to support the TI C timekeeping functions (listed in `time.h`). Source code for the functions is given in the `\PERIPH\BUS` directory.

**Digital Peripheral Drivers.** The digital drivers support access to all baseboard and add-on digital I/O functions.

The DIO peripheral drivers provide control functions for the optional DIO plug-in module. The functions provide high-level C access to the DIO module's 32 additional digital I/O lines plus either interrupt-driven or polled use of the DIO's onboard DUART (Dual-channel Universal Asynchronous Receiver Transmitter). The target programming section gives details on how to use the digital peripheral library to program the digital peripherals. Source code for the functions is given in the `\PERIPH\DIGITAL\DIO` directory.

The MOT peripheral drivers provide control functions for the optional MOT plug-in module. The functions provide high-level C access to the MOT module's four, precision motion-control axes. Each of the axes features independent encoder inputs and either digital or 16-bit analog output. Digital output may be either pulse and direction positive and negative pulse to support stepper motor amplifier inputs. The target programming section gives details on how to use the MOT peripheral library to program these peripherals. Source code for the functions is given in the `\PERIPH\DIGITAL\MOT` directory.

**Analog Peripheral Drivers.** The Analog peripheral drivers provide control functions for the optional analog plug-in modules: The A4D4, AIX and SD modules. The functions provide high-level C access to the A4D4's analog input and output channels and their associated gain amplifiers. Additionally, the driver supports control of the optional TERM break-out panel, a companion to the A4D4 module, in order to support muxing of each of the A4D4 modules 8:1 to allow input from up to 32 simultaneous channels per A4D4 module. Source code for the functions is given in the `\PERIPH\ANALOG\A4D4` directory.

The AIX peripheral drivers provide control functions for the optional AIX plug-in module. The functions provide high-level C access to the AIX module's four, 2.5 MHz, 16-bit analog input channels. Source code for the functions is given in the `\PERIPH\ANALOG\AIX` directory.

The SD peripheral drivers provide control functions for the optional SD plug-in module. The functions provide high-level C access to the A4D4 module's four, audio-grade, 24-bit analog input and 20-bit output channels. Source code for the functions is given in the `\PERIPH\ANALOG\SD` directory.

The target programming section gives details on how to use the analog peripheral library to program these analog peripherals.

**Digital Signal Processing Library.** The DSP directory contains code to support high-level access to the common signal processing functions such as FFT's, filters and compression. Additional routines are provided for common functions such as matrix manipulation, curve fitting and general purpose queue management. Source code for the functions is given in the `\DSP` directory.

**Texas Instruments C Libraries.** Several libraries are included with the system that provide support for floating point and extended math functions, DSP oriented procedures and initialization examples. Chapter 5 in the *TMS320 Floating Point DSP Optimizing C Compiler User's Guide* describes the libraries.

The following libraries are available:

<b>Library</b>	<b>Operation</b>
ASSERT.H	Defines the assert macro for runtime error message reporting.
CTYPE.H	Declares functions that test and convert characters.
LIMITS.H	Defines range limits for characters and variable types.
FLOAT.H	Defines floating point range limits
MATH.H	Defines trigonometric, exponential and hyperbolic math functions
ERRNO.H	Defines errno variable for catching range errors in function calls
STDARG.H	Defines macros to aid in variable argument functions
STDDEF.H	Defines two new types and macros used within runtime functions
STDLIB.H	Declares many common library functions such as string conversion, sorting and searching functions, program exit functions and some integer-arithmetic that is not a standard part of C
STRING.H	Declares functions for string manipulations
TIME.H	Declares macros and types useful for time manipulations

**TABLE 14. Texas Instruments Standard Library Functions**

## **M62 Hardware Interaction**

All peripherals are memory mapped into the 'C6201 address space, using the locations given in the following table. The table also lists the wait states applied to accesses to each peripheral.

The development system provides routines to access all integrated M62 peripherals. This section describes how to program the peripherals using the supplied library functions under C or via direct memory accesses to the supplied peripheral register map. In general, direct memory access delivers higher performance than using the C function library since it avoids the overhead of the function calls necessary to access the library. However, the libraries have been crafted to utilize inline code where possible to mitigate this effect. In the peripheral descriptions that follow, each device's access methods are called out for both high level and direct memory access. In the case of C functions, the function names and argument variables are called out. In the case of direct memory access operations, the relevant addresses are listed along with the functions they perform and accompanying `Periph` structure elements which may be used from C to simplify access. These elements are defined in the header file `periph.h`.

**TABLE 15. M62 External Peripheral Memory Map**

Function	Address	C Language Mneumonic	Mem Space
FIFO Port	0x0400000	Periph->Fifo	CE0
V360 Registers	0x1400000	Periph->PciRegs	CE1
FIFO Port Reset	0x1410000	Periph->FifoReset	
AD9850 Reset	0x1470000	Periph->DDS.Reset	
AD9850 Frequency Update	0x1480000	Periph->DDS.Update	
AD9850 Write Clock	0x1490000	Periph->DDS.Clock	
Digital I/O Data Register	0x14A0000	Periph->Dio.Data	
Digital I/O Direction Control	0x14B0000	Periph->Dio.Direction	
Digital I/O Input Latch Clock Control Register	0x14C0000	Periph->Dio.LatchControl	
External Mux Control 0	0x14D0000	Periph->Mux[0]	
External Mux Control 1	0x14E0000	Periph->Mux[1]	
16 bit External Timer	0x14F0000	Periph->Timer	
External Interrupt Input 4 Select	0x1500000	Periph->EI[4]	
External Interrupt Input 5 Select	0x1510000	Periph->EI[5]	
External Interrupt Input 6 Select	0x1520000	Periph->EI[6]	
External Interrupt Input 7 Select	0x1530000	Periph->EI[7]	
I/O Module Strobe 0	0x1540000	Periph->Module[0]	
I/O Module Strobe 1	0x1550000	Periph->Module[1]	
I/O Module Strobe 2	0x1560000	Periph->Module[2]	
I/O Module Strobe 3	0x1570000	Periph->Module[3]	
I/O Module Strobe 4	0x1580000	Periph->Module[4]	
I/O Module Strobe 5	0x1590000	Periph->Module[5]	
I/O Module Strobe 6	0x15A0000	Periph->Module[6]	
I/O Module Strobe 7	0x15B0000	Periph->Module[7]	
I/O Module Strobe 8 (cM62 only)	0x15C0000	Periph->Module[8]	
I/O Module Strobe 9 (cM62 only)	0x15D0000	Periph->Module[9]	
I/O Module Strobe 10 (cM62 only)	0x15E0000	Periph->Module[10]	
I/O Module Strobe 11 (cM62 only)	0x15F0000	Periph->Module[11]	
Async SRAM (128Kx32)	0x1600000	Periph->ASRam[0..0x80000]	
SDRAM (16Mbyte) (optional)	0x2000000	Periph->SDRam[0..0x1000000]	CE2
SBSRAM (1Mbyte) (optional)	0x3000000	Periph->SBRam[0..0x100000]	CE3

This document does not describe peripheral hardware specifications and other hardware issues. Refer to the *M62 Hardware Manual* for hardware information.

## Digital Input/Output

The digital input/output (I/O) buffers provide a means for generating 32 bits of direct digital input or output to and from external hardware. This I/O can be clocked from either the 'C6201 processor or from external TTL sources, allowing external devices to automatically latch data into the I/O buffers for the 'C6201 to read.

Input/output direction for either half of the 32-bit port may be programmed on the fly using on-board logic. The port may be configured in software for input or output in groups of eight bits.

**Memory Mapped Digital I/O Access.** The following table shows the memory locations used to interact with the digital I/O buffers. Three C language routines are supplied to interact with the digital I/O port.

Function	C Language Mnemonic
Digital I/O Data Register	<code>Periph-&gt;Dio.Data</code>
Digital I/O Direction Control (4 bytes)	<code>Periph-&gt;Dio.Direction</code>
Digital I/O Latch Control	<code>Periph-&gt;Dio.LatchControl</code>

**TABLE 16. Digital I/O Access Memory Location**

The `Periph->Dio.Data` location is used to access the data lines of the digital I/O port. Results of read and write accesses depend on the I/O direction of the port (see below for information on setting the port direction). If the port is configured for input, a read access latches new read data from the external pins and the new data is read into the 'C6201. If the port is configured for output, the most recently latched output data is read into the 'C6201 (output data does not change). Write accesses to an input port cause no change to the port status, while write accesses to an output port cause the new data to be latched and output to the external I/O pins.

The `Periph->Dio.Direction` location controls the direction of each byte of the digital I/O port. The four least significant bits of this register are used to configure each of the bytes of the digital I/O port for either input or output, as follows:

Dio.Direction-Register Bit #	Value	Direction
0	0	DX[0..7] output (default)
	1	DX[0..7] input
1	0	DX[8..15] output (default)
	1	DX[8..15] input
2	0	DX[16..23] output (default)
	1	DX[16..23] input
3	0	DX[24..31] output (default)
	1	DX[24..31] input

**Table 17: Digital I/O Direction Configuration**

The `Periph->Dio.LatchControl` location controls the method of latching data into each byte of the digital I/O port. The four least significant bits of this register are used to configure the latch method as either internal (triggered by CPU accesses) or external (triggered by an external TTL pulse), as follows:

Dio.LatchControl Register Bit #	Value	Bits Affected	Clock Source
0	0	0..7	Internal (CPU-based)
	1		External
1	0	8..15	Internal (CPU-based)
	1		External
2	0	16..23	Internal (CPU-based)
	1		External
3	0	24..31	Internal (CPU-based)
	1		External

**TABLE 17. Digital I/O Latch Configuration**

**C Language Digital I/O Functions.** Data may be read or written to the digital I/O port using the following routines in the DIGITAL support library.

Function Name	Description
DIO_dir()	Sets the direction of all four bytes of the onboard 32-bit digital I/O port.
DIO_read()	Returns current state of all 32-bits of digital I/O port.
DIO_write()	Sets current state of all 32-bits of digital output port currently configured for output.
DIO_latchcontrol()	Sets the latch method of all four bytes of the onboard 32-bit digital I/O port.

**TABLE 18. Digital I/O library functions**

## Timers

The timers provide the capability to generate hardware timebases which can be used to trigger processor interrupts, analog signal conversions, or as direct outputs to external hardware. There are a total of six timebase sources built in to the M62: two 32-bit timers internal to the 'C6201 processor, and three 16-bit channels implemented with custom logic within the FPGA plus one AD9850 direct digital synthesizer. The supplied library functions initialize the timers to a free-running, pulse generation mode suitable for generating convert pulses to the analog hardware.

The timers are initialized by code in the `timebase()` routine each time it is called. Normally, no other function calls are necessary to use the timers. However, when supplying an external TTL signal to the 'C6201 TCLK0/1 inputs in order to provide an external timebase to analog circuitry, it will be necessary to create and use a custom version of `timebase()` which tristates the TCLK output driver to avoid contention with external sources. Please note that certain hardware setups might be required depending on the application. See the *M62 Hardware Manual* for more details on how to set up the M62 board.

**C Language Timer Functions.** The following functions give high-level access to the timer hardware. See the appendices for complete information on the functions.

Function Name	Operation
<code>timebase()</code>	Configures a specified timer channel (0..5) for periodic counting at a specified frequency using a specified source clock rate..

**TABLE 19. C Language Timer Functions**

`timebase()` can be used to set a particular timebase to a particular frequency. For example, the following call sets PIT timer channel 1 to generate a 1000 Hz out-

put pulse stream, assuming the the hardware default 1 MHz input clock to the FPGA logic:

```
timer(1, 1000.0, 1.0);
```

**Memory Mapped Timer Access** It is possible to directly access to the internal timer hardware controls via memory mapped registers at specific addresses. It may be necessary to use these addresses to set the timers to a custom mode. In general, unless custom functionality is required of the timers, it is recommended that the user exclusively access the timers via the `timer()` routine rather than programming the control and period registers manually.

For information about the 'C6201 internal timers, please see the *TMS320C6x User's Guide*. For information about the custom PIT counter/timer device, contact Innovative Integration. For an example of direct timer channel control, refer to the source code for the `timebase()` function, located in the `PERIPH\DIGITAL` subdirectory.

**STDIO Communication.** C stdio terminal emulation is provided in the Peripheral Library. The stdio library communicates with the host `TERMINAL.EXE` program via the V360 PCI interface mailbox registers to provide stdio support to DSP applications running on the M62. The stdio interface may be used for real-time, non-intrusive software debugging or to create a basic user interface for OEM applications.

The following list shows the available Peripheral Library calls and their operation. See Appendix I for complete information on the functions.

Function Name	Operation
<code>putchar()</code>	Emits an 8-bit character to the terminal emulator
<code>getchar()</code>	Gets an 8-bit character from the terminal emulator's keyboard buffer
<code>gets()</code>	Inputs a string into a target buffer
<code>puts()</code>	Displays a string from a target buffer
<code>sprintf()</code>	Formats a string into a memory buffer pointed to by buffer
<code>printf()</code>	Prints a formatted string to the terminal
<code>scanf()</code>	Inputs a formatted string from the terminal into a buffer
<code>sscanf()</code>	Converts a formatted string in memory into a buffer..
<code>stdio_reset()</code>	Resets the terminal emulator display
<code>fopen()</code>	Opens a file on the Host PC, returning the file handle
<code>fclose()</code>	Closes a previously opened Host PC file.
<code>fread()</code>	Reads file contents into a target buffer
<code>fwrite()</code>	Writes a target buffer into a Host PC file
<code>fseek()</code>	Repositions the Host PC file pointer
<code>ferase()</code>	Erases the specified Host PC file

<code>kbd_hit()</code>	Returns a nonzero value if characters are currently available in the monitor keyboard buffer
<code>kbd_key()</code>	Returns 16-bit IBM scancode for pending keystroke from the terminal emulator's keyboard buffer.
<code>gotoxy()</code>	Moves the terminal cursor
<code>wherexy()</code>	Returns the terminal cursor position
<code>clreol()</code>	Clears to end of current line
<code>clrscr()</code>	Clears the terminal screen
<code>type()</code>	Types formatted, null terminated string to console
<code>bold()</code>	Enables bold text attribute in terminal emulator
<code>normal()</code>	Enables standard text attribute within terminal emulator
<code>get_attribute()</code>	Returns the current character display attributes
<code>set_attribute()</code>	Sets the current character display attributes
<code>cursor()</code>	Enables/disables the cursor
<code>get_busmaster_addr()</code>	Obtains the base of the host busmaster memory from the terminal emulator.
<code>plot()</code>	Plots a Host PC file as a graph.
<code>view()</code>	Plots a target buffer as a graph.

**TABLE 20. STDIO Driver Functions**

**Using Interrupts.** The M62 supports four external and numerous internal hardware interrupts. These include EI0, EI1, EI2, EI3 plus TINT0, TINT1 (internal timer/counters), internal comm port transmit and receive and DMA.

Interrupts on the TMS320C6201 may be handled by writing either high-level C or assembly language procedures within your application files which employ the following interrupt-specific function names:

```
void c_intNN()      for C handlers or
_c_intNN           for assembly language
```

where NN is numbered 0 through 99 for each of the interrupts. For each interrupt, a procedure must be coded which will be executed upon acknowledgment of interrupt NN by the 'C6201. This is described on page 4-27 of the C Compiler Users Manual.

Consider the following code example:

```
/*
 * EXAMPLE.C
 */
#define TINT0 14
```

```
main()
{
    enable_interrupts();      /* Enable unmasked xrpts */
    timebase(1, 1000.0, 1.0); /* Internal timer 1 at 1kHz */
    /* install interrupt handler on TINT1 */
    install_int_vector(c_int02, TINT0);
    enable_interrupt(TINT0);
    .
    /* Bulk of application */
    .
    disable_interrupt(TINT0); /* Disable TINT0 xrpt */
}
/*
 * ISR for timer 0 - Tally a variable
 */
int milliseconds
void c_int14()
{
    milliseconds++; /* Internal timer 0 is used to */
}
/* synthesize a timebase */
```

In this code, the internal timer 0 is configured to output a pulse every millisecond which drives TINT0 on the 'C6201. The vector is installed into the jump table with a call to `install_int_vector()` and the bit associated with TINT0 in the

interrupt enable register, is enabled. Finally, `main()` calls `enable_interrupts()` which sets the global interrupt enable bit so that all unmasked interrupts can be processed.

Each time the counter expires, the routine `c_int14()` executes. In this example, the variable `milliseconds` is incremented during each interrupt service cycle.

Each DSP application should include a copy of the default interrupt vector table which is defined in `vectors.asm`. This assembly file is located in the `PERIPH\RTS` directory and when compiled into a `.obj` file and linked into the application, will cause all entries in the vector table to be initialized with a default handler (the one exception being the break interrupt vector, which is filled with the pointer to the talker program). If an application needs to make use of interrupts, those vectors which are affected need to be changed with `install_int_vector()` at run time.

See the target example programs provided on your distribution disks for further examples of the use of interrupts.

---

### *Example Target Programs for the M62*

The following section details the example target software included with the Developer's Package. These programs are provided as models for custom user software, and it is highly recommended that the user examine these examples before beginning a first development effort for the target. Full source code is provided for user inspection and reuse in modified or custom applications.

These examples will run on a standard M62 card with no additional hardware required.

#### **HELLO**

HELLO is a very simple introduction to basic program components and use of the C `stdio` library for the target card. When run with the host terminal emulator active, the program simply initializes the target hardware and `stdio` interface and prints the message "Hello, world" via the `stdio` library to the terminal emulator screen. The program then drops into an infinite dwell loop.

HELLO may be rebuilt from within the Codewright environment by loading the HELLO.PJT project from the \target\examples directory, modifying the source file HELLO.C, and rebuilding the project (see the Codewright documentation for more information on the application's project management and make facilities).

For correct program functionality, it is necessary to run the HELLO application via the host terminal emulator program. If the terminal emulator is not active and communicating with the target M62 card on which HELLO is running, the application will appear to hang at the first instance of a stdio function call (usually a `get - int ()` or `put int ()` call). This is due to the fact that all stdio calls use the M62 bus mailbox interface and are handshaken with the host terminal emulator application, and any such calls will hang if the terminal emulator is not active to complete the communication link.

## **TEST**

TEST is board level hardware test program, capable of exercising the major peripherals on the M62 to double-check proper hardware functionality. As such, it contains routines for exercising each of the peripherals on the M62, including:

1. Digital I/O
2. Internal timers
3. External timers
4. Communications Ports

Since the TEST program aims to be all-encompassing in that it tries to test as much of the board-level functionality as possible, it serves as a poor example for complicated operations such as A/D multi-channel sampling and display. However, since the code included for TEST is broken down into functional pieces which are called separately for each subsystem to be tested, it is possible to factor out individual tests for use in other programs.



# Target DSP Peripheral Libraries

## Target Functions by Category

Category	Name	Description	
Board Initialization & System Functions	baud	Set baud rate on current serial port	
	cpu	Set CPU number and mailbox	
	cpu_num	Get CPU number	
	cpu_number	Get CPU number (inline)	
	detect_cpu_speed	Derive DSP clock speed	
	dma_done	Wait for DMA completion	
	dpram_addr	Return start address of Dualport RAM on PC31	
	dpram_type	Detects 16 or 32 bit Dualport RAM on PC31	
	init_serial	Initialize the serial I/O system	
	InitIP	Initialize Industry Pack access structure	
	mem_size	Detect size of memory space	
	test_mem	PC31 memory check	
	Busmaster Transfer Functions	bm_init	Busmaster transfer initialization
		bm_transfer	General busmaster transfer
fifo_init		Busmaster initialization	
transfer_complete		Wait for Busmaster transfer to complete	
USB Bulk Transport Interface Functions	InitBulkTransport	Initialize the Bulk Transport Interface	

	StopBulkTransport	Shut down the Bulk Transport Interface
	IsBulkTransportReady	Returns true if system can send data
	OpenBulkTransport	Opens a channel of the Bulk Transport System
	CloseBulkTransport	Shuts down an open channel of the Bulk Transport System
	ReadBulk	Read a block from a Bulk Transport channel
	WriteBulk	Writes a block to a Bulk Transport channel
	BulkDataAvailable	Returns the amount of data available for reading on a channel
	BulkSpaceAvailable	Returns the room for new data available on a channel
	FlushBulk	Forces the transmission of all data in a channel
Digital I/O Functions	C31_dig_dir	Program the direction of PC31/SBC31 PIA Digital I/O bytes
	C31_read_dig	Read PC31/SBC31 PIA Digital I/O lines
	C31_write_dig	Write to PC31/SBC31 PIA Digital I/O lines
	C31_write_dig_bit	Update a single bit on PC31/SBC31 PIA Digital I/O
	dig_dir	Program the direction of Digital I/O bytes
	read_abits	Read state of ABITS output lines
	read_abits_bit	Read state of a single ABITS output bit
	read_dig	Read Digital I/O lines
	read_dig_bit	Read state of a single digital bit
	write_abits	Write to ABITS digital output
	write_abits_bit	Update a single ABITS digital output bit
	write_dig	Write to digital output
	write_dig_bit	Update a single digital output bit
Analog I/O Control Functions	enable_analog	Initialize analog subsystem
	trigger_adc	Set triggering mode for an ADC
	trigger_adc_pair	Set triggering mode for an ADC pair
	trigger_dac	Set triggering mode for an DAC
	trigger_dac_pair	Set triggering mode for an DAC pair
	write_analog_interrupt_mask	Set which analog conversions fire interrupts
Analog Input Functions	correct_adc	Adjust ADC reading to proper range
	correct_adc_pair	Adjust a pair of ADC readings to proper range
	convert_adc	Manually trigger an ADC conversion
	convert_adc_pair	Manually trigger an ADC conversion on an ADC pair
	read_adc	Read data from ADC
	read_adc_pair	Read data from a pair of ADCs
	read_adc_automux	Read data from ADC, and switch multiplexer

	read_adc_pair_automux	Read data from a pair of ADCs, and switch mux
Analog Output Functions	correct_dac	Adjust DAC reading to proper range
	correct_dac_pair	Adjust a pair of DAC readings to proper range
	convert_dac	Manually trigger a DAC conversion
	convert_dac_pair	Manually trigger a DAC conversion on a DAC pair
	convert_dacs	Manually trigger DAC conversions using a bit mask
	read_dac	Read last value loaded into a DAC
	read_dac_pair	Read last value loaded into a DAC pair
	update_dac	Write DAC value and automatically trigger conversion
	update_dac_pair	Write DAC pair and automatically trigger conversion
	write_dac	Write value to DAC
	write_dac_pair	Write value pair to a DAC pair
Programmable Gain Functions	gain_to_mode	Convert Gain into equivalent Gain Mode number
	mode_to_gain	Convert gain mode to actual gain value
	read_gain	Read last Gain setting
	write_gain	Update gain setting for a channel
	write_gains	Update gain setting for all channels
Mux Control Functions	auto_mux	Configure automatic multiplexing feature
	read_mux	Read last setting of a particular mux
	write_mux	Update multiplexer setting for a channel
	write_muxes	Update multiplexer setting for all channels
Mailbox and Semaphore Functions	check_inbox	Check incoming mailbox for new data
	check_outbox	Check outgoing mailbox for new data
	clear_mailboxes	Clear mailboxes
	get_semaphore	Get hardware semaphore
	read_mailbox	Read from incoming mailbox
	read_mb_terminate	Read from incoming mailbox if data available
	release_semaphore	Release hardware semaphore
	write_mailbox	Write to outgoing mailbox
	write_mb_terminate	Write to outgoing mailbox if box is ready
Interrupt Support Functions	deinstall_int_vector	Remove vector from vector table
	disable_interrupt	Disable specific interrupt
	enable_interrupt	Enable specific interrupt
	host_interrupt	Target to host interrupt
	install_int_vector	Install vector into vector table
	mailbox_interrupt	Post a mailbox interrupt to the host
	mailbox_interrupt_ack	Acknowledge a mailbox interrupt
	mailbox_interrupt_deinstall	Unload the handler for mailbox interrupts
	mailbox_interrupt_disable	Disable mailbox interrupts

	mailbox_interrupt_enable	Enable mailbox interrupts
	mailbox_interrupt_install	Load a handler for mailbox interrupts
	suspend	Idle until interrupts arrive
	interrupt_cpu	Interrupt specified multiprocessor target CPU
	cpu_int_src	Return source code # for specified multiprocessor CPU
	cpu_xrpt_bit	Return register index to specified multiprocessor CPU
Timer Functions	disable_clock	Disable system millisecond timebase
	enable_clock	Initialize system millisecond timebase
	ms	Dwell milliseconds
	read_timer	Read value from a hardware timer
	timebase	Set hardware timer frequency
	timer	Set hardware timer frequency
	uclock	Get system millisecond timer value
	us	Dwell microseconds
Memory Movement Functions	copy_mem	Fast on-chip memory copy
	fill_mem	Fast on-chip memory fill
	mem_to_port	Fast on-chip transfer of data to a port
	port_to_mem	Fast on-chip transfer of data to a port
	dma_copy_mem	Fast DMA memory copy
	dma_fill_mem	Fast DMA memory fill
	dma_mem_to_port	Fast DMA transfer of data to a port
	dma_port_to_mem	Fast DMA transfer of data to a port
Conversion Functions	from_ieee	Convert from IEEE-754 floating point format
	packb	Pack byte value into int
	packh	Pack half word value into int
	to_ieee	Convert to IEEE-754 floating point format
	unpackb	Unpack byte values from int
	unpackh	Unpack half word values from int
Flash Memory Programming	fast	Restore PBCR to original value after Flash access
	flash_erase	Erase entire Flash memory
	flash_init	Initialize Flash for programming
	flash_rd	Read Flash byte
	flash_read	Read 32-bit word from Flash
	flash_sector_erase	Erase a Flash sector
	flash_wr	Write a byte to Flash memory
	flash_write	Write 32-bit word to Flash
	slow	Reduce speed of I/O accesses to access Flash memory
CPU Register I/O	clear_interrupt_flag	Disable interrupt enable bit
	get_DIE	Retrieve 320C4x DIE register
	get_IE	Retrieve 320C3x IE register
	get_IIE	Retrieve 320C4x IIE register
	get_IF	Retrieve 320C3x IF register
	get_IIF	Retrieve 320C4x IIF register
	get_IOF	Retrieve 320C3x IOF register
	get_ST	Retrieve 320C3x/4x Status register
	set_DIE	Set 320C4x DIE register
	set_IE	Set 320C3x IE register
	set_IF	Set 320C3x IF register
	set_IIE	Set 320C4x IIE register
	set_IIF	Set 320C4x IIF register
	set_IOF	Set 320C3x IOF register

	set_interrupt_flag	Set C3x Interrupt Flag Bit
	set_PC	Set processor program counter
	set_ST	Set processor status register

## FIFO Library Functions

FIFO Link Support	set_fifo_link_AF_levels	Set almost-full threshold levels
	fifo_link_emit	Send a character to link using handshake
	fifo_link_key	Get a character from link using handshake
	fifo_link_spit	Send a character to link without using handshake
	fifo_link_eat	Get a character from link without using handshake
	bleed_fifo_link	Drain FIFO into memory buffer
	fill_fifo_link	Fill FIFO from memory buffer
	reset_fifo_link	Initialize a link to empty state
	get_fifo_link_status	Obtain fullness state information
	login()	Query subordinate processors for login sequence
	sub_login	Send login sequence to master processor
	fifo_link	Return register index to FIFO link for specified CPU
FIFO Port Support	set_fifo_port_AF_levels	Set almost-full threshold levels
	fifo_port_emit	Send a character to link using handshake
	fifo_port_key	Get a character from link using handshake
	fifo_port_spit	Send a character to link without using handshake
	fifo_port_eat	Get a character from link without using handshake
	bleed_fifo_port	Drain FIFO into memory buffer
	fill_fifo_port	Fill FIFO from memory buffer
	reset_fifo_port	Initialize a link to empty state
	get_fifo_port_status	Obtain fullness state information

## Standard I/O Library Functions

Category	Name	Description
Console Terminal Control Functions	bold	Set console text bold attribute
	clreol	Clear console to end of line
	clrscr	Clear console screen
	cursor	Enable/disable console cursor
	get_attribute	Get current console text attribute type
	gotoxy	Set cursor position
	normal	Set console text normal attribute
	set_attribute	Set current console text attribute type
	wherexy	Get cursor position
Low Level I/O	emit	Send a character to the terminal emulator
	getchar	ANSI get character from console
	kbd_hit	Install vector into vector table
	kbd_key	Get a key from the terminal emulator
	key	Get a character from the standard mailbox
C Standard I/O Library Emulation Functions	putchar	ANSI put character to console
	fclose	Close a host disk file
	fclose	Close a host disk file
	fclose	Close a host disk file
	fremove	Delete a host disk file by name
	fflush	Commits an open file I/O stream to disk
	fopen	Open a host disk file for read
	fread	Read from host disk file into target memory
	fseek	Moves the file pointer to a specified location
	fwrite	Write to host disk file from target memory
	gets	ANSI gets from console
	printf	ANSI printf to console
	puts	ANSI puts to console
	scanf	ANSI scanf from console
	sprintf	ANSI sprintf
sscanf	ANSI sscanf	
Terminal Applet Extensions	type	Send a character string to the terminal emulator
	get_busmaster_addr	Retrieve host busmaster address from Terminal
	plot	Transfer data buffer to host for plotting
	stdio_reset	Reset the Terminal program
	stdio_terminate	Send the termination code to Terminal

## DSP Library Functions

Category	Name	Description	
Signal Processing Functions	bartlett	Bartlett window generation	
	bitrev	Bit reversal function	
	blackman	Blackman window generation	
	buffer_statistics	Calculate statistics on a data buffer	
	fft_r1	Forward Fast Fourier Transform - Real	
	fft_r2	Forward Fast Fourier Transform - Complex	
	fir	Finite Impulse Response Filter	
	hamming	Hamming window generation	
	hanning	Hanning window generation	
	harris	Harris window generation	
	ifft_r1	Inverse Fast Fourier Transform - Real	
	ifft_r2	Inverse Fast Fourier Transform - Complex	
	Matrix Functions	vmul	Multiply two vectors into a third vector
		matrix_add	Add two matrices and return a sum MATRIX
		matrix_allocate	Allocate a matrix and return its MATRIX pointer
matrix_crop		Form sub-matrix from a larger matrix	
matrix_det		Return the determinant of a square matrix	
matrix_free		Free matrix area and MATRIX structure	
matrix_invert		Invert a square matrix, return inverse MATRIX	
matrix_mult		Multiply two matrices, return new MATRIX	
matrix_mult_pwise		Multiply two matrices element by element	
matrix_print		Print the elements of a matrix to stdout	
matrix_scale		Scale all of a matrix by a constant	
matrix_sub		Subtract two matrices and return a difference MATRIX	
Queue Support Functions	matrix_transpose	Transpose a matrix, return pointer to new MATRIX	
	dequeue_ptr	Remove data from a queue and adjust pointer	
	enqueue_ptr	Load data into Queue and update pointers	
BERR Sequence Generation Functions	enqueued	Return count of data elements in a Queue	
	queue_init	Initialize memory Queue structure	
	berr_decode	Tests a value in a BERR sequence	
Data Compression Functions	berr_encode	Generate the next value in a BERR sequence	
	berr_initialize	Set up a BERR sequence generator	
	a_compress	A-Law data compression	
	a_expand	A-Law data expansion	
	mu_compress	Mu-Law data compression	
	mu_expand	Mu-Law data expansion	



## DLL Functions Grouped by Function

The functions tabularized below may be used in any Host program written in a language which supports access to a Dynamic Link Library. The prototypes for these functions are listed in the PERIPH\INCLUDE\LIB\TARGET.H file. These names of these functions are aliai of the actual board-specific library function names which are prototyped in PERIPH\LIB\HOST\ALIAS.H.

**TABLE 21. Generic DLL Function List**

Category	Function Prototype	Function Description
General	BOOL target_open(int target)	Opens driver for specified target DSP board. Returns boolean.
	BOOL target_close(int target)	Closes driver for specified target DSP board. Returns boolean
	LPVOID target_cardinfo(int target);	Returns address of cardinfo structure for target.
	int icoffld(char *, int target, HWND hParent);	Loads a COFF executable file onto target DSP
Interrupt Functions	BOOL host_interrupt_enable(int target);	Enables a previously installed virtual interrupt handler.
	BOOL host_interrupt_disable(int target);	Disables a previously enabled virtual interrupt handler

	void host_interrupt_install(int target, void (*virtual_isr)(void *), void * context);	Installs a virtual interrupt handler
	void target_interrupt(int target);	Interrupts target DSP board
	void host_interrupt_deinstall(int target);	Removes a virtual interrupt handler.
	void mailbox_interrupt(int target, unsigned int value);	Interrupts the target DSP after writing value to special mailbox
	unsigned int mailbox_interrupt_ack(int target);	Acknowledges target to Host interrupt, returns special mailbox contents
Control Functions	void target_reset(int target);	Physically asserts reset on the target DSP board.
	void target_run(int target);	Deasserts reset on the target DSP board
	void target_outport(int target, int port, int value);	Outputs a value to specified DSP board I/O port address
	int target_inport(int target, int port);	Inputs a value from specified DSP board I/O port
	void target_opreg_outport(int target, int port, int value);	Outputs a value to specified DSP board operation port address
	int target_opreg_inport(int target, int port);	Inputs a value from specified DSP board operation port
Mailbox Functions	void target_control(int target, int bit, int state);	Modifies a bit in the control register of the target DSP board
	int read_mailbox(int target, int);	Reads the specified mailbox of the target DSP board
	void write_mailbox(int target, int, int);	Writes to the specified mailbox of the target DSP board.
	BOOL check_outbox(int target, int);	Interrogates the specified output mailbox status
	BOOL check_inbox(int target, int);	Interrogates the specified input mailbox status
	int read_mb_terminate(int target, int, int *, int wide);	Reads the specified input mailbox, if full
	int write_mb_terminate(int target, int box_number, int value, int wide);	Writes to the specified output mailbox, if empty
	void clear_mailboxes(int target);	Clears all mailboxes to empty state
	int target_key(int target);	Reads terminal mailbox, returns an 8-bit contents
	void target_emit(int target, int value);	Writes 8-bit value to terminal mailbox
	void target_Tx(int target, int value);	Writes 32-bit value to terminal mailbox
	Bulk Transport Interface Functions	int target_Rx(int target);
int BULK_GetNumDevices();		Returns the number of SBC62 USB devices detected
BOOL BULK_OpenDevice(int iDevice, HANDLE *phDevice)		Opens a device for BULK transport access.
BOOL BULK_CloseDevice(IN HANDLE hDevice)		Closes a device for BULK transport access
BOOL BULK_OpenChannel(int iDevice, WORD wChannel, BOOL fOverlapped, BULK_HANDLE *pHandle);		Opens a data channel in BULK mode

	BOOL BULK_CloseChannel(BULK_HANDLE Handle)	Closes a data channel opened with BULK_OpenChannel()
	BOOL BULK_Read(BULK_HANDLE Handle, LPVOID lpBuffer, DWORD dwNumberOfBytesToRead, LPDWORD lpNumberOfBytesRead, LPOVERLAPPED lpOverlapped);	Reads a block of data in BULK mode.
	BOOL BULK_Write(BULK_HANDLE Handle, LPCVOID lpBuffer, DWORD dwNumberOfBytesToWrite, LPDWORD lpNumberOfBytesWritten, LPOVERLAPPED lpOverlapped);	Writes a block of data in BULK mode
	BOOL BULK_GetOverlappedReadResult(BULK_HANDLE Handle, LPOVERLAPPED lpOverlapped, LPDWORD lpNumberOfBytesTransferred, BOOL bWait )	Gets the WIN32 Overlapped Result for the Read portion of the data channel.
	BOOL BULK_GetOverlappedWriteResult(BULK_HANDLE Handle, LPOVERLAPPED lpOverlapped, LPDWORD lpNumberOfBytesTransferred, BOOL bWait );	Gets the WIN32 Overlapped Result for the Write portion of the data channel.
	BOOL BULK_CancelIo(BULK_HANDLE Handle )	Cancels all pending I/O on the device
	BOOL EXPORT STREAM_Open(int iDevice, WORD wChannel, WORD wBufferSize, WORD wBlockSize, BULK_HANDLE *pHandle)	Opens s data channel in STREAM node.
	BOOL STREAM_Close(BULK_HANDLE handle )	Closes a STREAM data channel
	WORD STREAM_WriteAvailable(BULK_HANDLE handle )	Returns the amount of space available for Write data
	WORD STREAM_ReadAvailable(BULK_HANDLE handle )	Returns the amount of data available on the STREAM channel
	WORD STREAM_Write(BULK_HANDLE handle, INT32 *pBuffer, WORD wElementCount )	Writes a block of data to the STREAM channel
	void STREAM_Read(BULK_HANDLE handle, INT32 *pBuffer, WORD wElementCount )	Reads a block of data from the STREAM channel
	void STREAM_Flush(BULK_HANDLE handle )	Writes all the output data to the target
Semaphore Functions	void get_semaphore(int target, int semaphore);	Gains ownership of specified target semaphore
	void request_semaphore(int target, int semaphore);	Requests ownership of specified target semaphore

	BOOL own_semaphore(int target, int semaphore);	Interrogates ownership status of specified semaphore
	void release_semaphore(int target, int semaphore);	Relinquishes control of specified semaphore
Talker Functions	int target_check(int target);	Interrogates for Talker running on target
	void start_app(int target);	Starts a previously downloaded target application program
	int start_talker(int target);	Starts the target Talker executing.
	int target_revision(int target);	Returns the revision of the target Talker





# DOS Environment Requirements

Innovative Integration Developers Packages, including the TI C Compiler, make use of environment variables in order to locate header files monitor script files, etc. Be sure to set the following environment variables when installing either the C compiler or II libraries. Note that a number of these environment variables may be automatically set when running the SETUP program on the distribution disks. However, when upgrading from previous versions or when mixing development components from II or other sources, problems can arise.

Use the table below to insure that you specify all needed environment variables.

Environment Variable Name	Products Affected	Suggested settings
DSP_COMPILE R	All TTC Compilers	<b>set DSP_COMPILER</b> =<compiler dir> ie set DSP_COMPILER=c:\c6xtools
II_BOARD	Dev Pkg Applets	<b>set II_BOARD</b> =<board dir> ie set II_BOARD=c:\M62cc
C_DIR	All TTC Compilers  All II peripheral libraries	<b>set C_DIR</b> =%ii_board%;%ii_board%\include\target;<compiler dir> ie set C_DIR=c:\M62cc;c:\M62cc\include\target;c:\c6xtools Specified order is critical!

C_OPTIONS	TI Flt Pt C Compiler	<b>set C_OPTIONS=&lt;switches&gt;</b> ie set C_OPTIONS =-q -x2 -o2 -g -ss
A_DIR	All TI Assemblers	Same as C_DIR above
D_DIR	TI Debuggers (Not Code Composer)	<b>set D_DIR=&lt;debugger dir&gt;</b> ie set D_DIR=c:\c3xhll
D_SRC	All Debuggers	<b>set D_SRC=&lt;source code dir1&gt;;&lt;dir2&gt;;...;&lt;dir n&gt;</b> ie set D_SRC=c:\M62cc\stdio;c:\M62cc\dsp; c:\M62cc\periph\analog;c:\M62cc\periph\digital;... ;c:\M62cc\periph\bus
PATH	All II products All TI Tools	<b>set path=&lt;old path&gt;;&lt;compiler dir&gt;;&lt;board dir&gt;;&lt;host lib dir&gt;</b>  set path=%path%;%dsp_compiler%;%ii_board%;%ii_board%\host\lib

**TABLE 22. Required disk directory structure for II development tools.**

The II TI C Development System for the M62 requires the following environment variables be set properly for correct operation:

```
set dsp_compiler=c:\c6xtools

set ii_board=c:\M62cc

set c_dir=%ii_board;%ii_board%\include\target;c:\c6xtools

set a_dir=%ii_board;%ii_board%\include\target;c:\c6xtools

set d_src=c:\M62cc\stdio;c:\M62cc\dsp;

c:\M62cc\periph\analog;c:\M62cc\periph\digital;

c:\M62cc\periph\misc;c:\M62cc\periph\flash;

c:\M62cc\periph\bus

set c_option=-ss -o2 -g -x2 -q
```

---

---

```
path=%path%;%dsp_compiler%;%ii_board%;%ii_board%\lib\host
```

