
SBC6x Development Package Manual

The SBC6x Development Package Manual was prepared by the technical staff of Innovative Integration, April 3, 2002.

For further assistance contact:

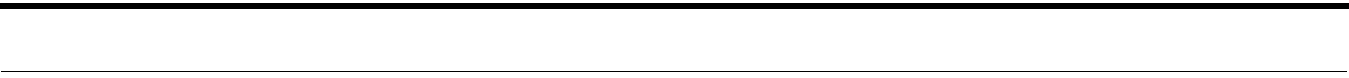
Innovative Integration
2655 Park Center Drive
Simi Valley, California 93065

PH: (805) 520-3300
FAX: (805) 579-1730

email: techsprt@innovative-dsp.com
Website: www.innovative-dsp.com

This document is copyright 2001 by Innovative Integration. All rights are reserved.

VSS\SBC62\documents\Hw-Sw Manual\SBC6x.book



	List of Tables.	7
	List of Figures.	9
CHAPTER 1	<i>Introduction</i>	11
	A Note about this Manual.	12
CHAPTER 2	<i>Installation</i>	13
	Host Hardware Requirements.	13
	Software Installation.	14
	<i>Begin Installation</i>	14
	<i>Launch Installation</i>	15
	<i>System DLL support Installation.</i>	15
	<i>Code Composer support Installation</i>	15
	<i>Dsp Component support Installation</i>	16
	17
	17
	<i>Applets support Installation</i>	17
	<i>Peripheral Libraries and examples Installation.</i>	17
	<i>HASP key Installation.</i>	17
	<i>Hasp Key Activation</i>	18
	<i>ReadMe Files</i>	19
	Hardware Installation	19
	<i>JTAG Emulator Hardware Installation.</i>	19
	<i>DSP Board Installation</i>	20
	Code Composer Studio Setup.	21
	Testing the Development Package Installation.	24
	<i>Configuring the Applets within the Development Package.</i>	24
	<i>Running the "JTAG Diagnostic" Utility</i>	26
	<i>Running an Example Program using UNITERMINAL</i>	26
	<i>Testing the Code Composer Debugger</i>	28
	Troubleshooting Installation Problems.	30
	<i>Most Commonly Asked Questions.</i>	30
	<i>Code Composer Studio Troubleshooting.</i>	34
	Multiple Board Support	34
CHAPTER 3	<i>Integrated Development Environment</i>	39
	The Texas Instruments C Compiler Toolset	39
	<i>C Compiler Toolset Usage</i>	40
	Code Composer Studio.	40
	<i>Editor.</i>	40
	<i>Debugger.</i>	40

CHAPTER 4	<i>Support Applets</i>	41
	The Terminal Emulator “UniTerminal”	41
	The COFF File UniDownloader	50
	<i>Using the Application</i>	50
	<i>Command Line Switches</i>	50
	<i>Manual Operation</i>	51
	The COFF File Dump Utility	52
	Creating a Binary File	54
	The Flash Burn Utility	55
	<i>Target Page</i>	55
	<i>Talker Page</i>	56
	<i>Flash Page</i>	58
	<i>Info Page</i>	59
	<i>Example Burn Sequence</i>	60
	Common Problems when Embedding Code	61
CHAPTER 5	<i>Developing Target Code</i>	63
	Introduction	63
	<i>Components of Target Code (.c, .asm, .cmd)</i>	64
	Edit-Compile-Test Cycle using Code Composer Studio	64
	A Simple Code Composer Studio Project	64
	<i>Build Options (M62, Q62, SBC62 Boards)</i>	66
	<i>Build Options (M67, Q67, SBC67 Boards)</i>	68
	<i>Build Options (SBC6711 Boards)</i>	70
	<i>Build Options (M44 Boards)</i>	72
	<i>Build Options (SBC32 Boards)</i>	74
	<i>Automatic projectfile creation</i>	77
	<i>Rebuilding a Project</i>	77
	<i>Running the Target Executable</i>	77
	Anatomy of a Target Program	78
	<i>Use of Library Code</i>	79
	<i>Compiling/Assembling/Linking Outside Code Composer Studio</i>	79
	The Next Step: Developing Custom Code	80
CHAPTER 6	<i>Developing Host Code</i>	81
	Dynamic Link Library	81
	<i>Sample Host Programs</i>	82
	DspComponent	84
	Host Example Programs for the SBC671x Baseboard	85
	<i>Overview</i>	85
	<i>MessageTest</i>	86
	<i>The Target Application</i>	88

	<i>VcMessage Test</i>	89
CHAPTER 7	<i>Creating Target Software</i>	93
	C Code Development	93
	<i>C Compiler</i>	93
	<i>C Library Reference</i>	94
	<i>SBC6x Zuma Toolset Libraries</i>	94
	<i>Texas Instruments C Libraries</i>	97
	<i>SBC6x Hardware Interaction</i>	98
	<i>Digital Input/Output</i>	99
	<i>Timers</i>	100
	<i>STDIO Communication</i>	101
	<i>Using Interrupts</i>	102
	The SBC6x Bulk Transport Interface	104
	<i>Overview of the Bulk Transport Interface</i>	104
	<i>Target API for Bulk Transport</i>	105
	<i>Host USB Support</i>	105
	<i>Host API for Bulk Transport</i>	105
	<i>Example Programs for the Bulk Transport Interface</i>	106
	Building Flash Programmable Applications	107
	Example Target Programs for the SBC6x	108
	<i>HELLO</i>	108
	<i>TEST</i>	108
	<i>EMBED</i>	109
CHAPTER 8	<i>Target DSP Peripheral Libraries</i>	111
CHAPTER 9	<i>Host DLL Reference</i>	119
CHAPTER 10	<i>DOS Environment Requirements</i>	123
CHAPTER 11	<i>SBC6x Hardware</i>	125
	SBC6x Hardware Functions	125
	Memory Map	126
	SBC6x Hardware Initialization Requirements	127
	External Memory	128
	SBC6x OMNIBUS	128
	<i>SBC6x OMNIBUS Memory Mapping</i>	129
	<i>OMNIBUS Power</i>	130
	FIFOPort I/O Expansion	130
	<i>Transmitting and Receiving FIFOPort Data</i>	131
	<i>Monitoring FIFO Status</i>	132

<i>FIFOPort Reset</i>	133
<i>Controlling the FIFOPort Programmable Almost-full Flag</i>	133
<i>Timer I/O and the FIFOPort</i>	134
<i>Designing External Hardware for use with the FIFOPort</i>	134
‘C6x01 Serial Ports	136
RS232 Serial Port	136
USB Port	137
<i>USB Interrupts</i>	138
<i>DMA Support and the USB interface</i>	138
<i>USB Software Development</i>	139
Timers	139
<i>On-chip Timers</i>	139
<i>AD9850 Direct Digital Synthesizer</i>	140
Digital I/O	141
<i>Digital I/O Timing</i>	142
External Mux Control	142
Interrupts	143
JTAG Test Bus	146
Miscellaneous Control	146
Power Requirements	147

CHAPTER 12

<i>Appendices</i>	149
Connector pinouts	149
<i>JP11, JP12, JP16, JP17 - OMNIBUS I/O Connectors</i>	149
<i>JP13, 18, 14, 19 - OMNIBUS Bus Connectors</i>	151
<i>JP1 - Digital I/O Connector</i>	153
<i>JP23 - FIFOPort Connector</i>	154
<i>JP6, JP7 - Processor Serial Port Connectors</i>	155
<i>JP8 - JTAG Debugger Connector</i>	156
<i>JP22 - Power Input Connector</i>	157
<i>JP20, JP21 - External Multiplexer Control Connectors</i>	158
<i>JP3 - Asynchronous Serial Port Connector</i>	159
Board Layout	159
TMS320C6x01 Limitations and Errata	161
<i>Processor Speed Limitations and External Memory</i>	161
<i>Texas Instruments Device Errata</i>	162

List of Tables

TABLE 1.	PCI Debugger Package Contents	16
TABLE 2.	Host Support Applications	24
TABLE 3.	Common Problems when Embedding Code in Flash ROM	61
TABLE 4.	Zuma Toolset Source Directories	94
TABLE 5.	Zuma Toolset Support Subdirectories	95
TABLE 6.	Texas Instruments Standard Library Functions	97
TABLE 7.	SBC6x External Peripheral Memory Map	98
TABLE 8.	Digital I/O Access Memory Location	99
TABLE 9.	Digital I/O Direction Configuration	99
TABLE 10.	Digital I/O Latch Configuration	100
TABLE 11.	Digital I/O Library Functions	100
TABLE 12.	C Language Timer Function	101
TABLE 13.	STDIO Driver Functions	102
TABLE 14.	Bulk Transport System Target Functions	105
TABLE 15.	Bulk Transport System Host Functions	106
TABLE 16.	Generic DLL Function List	119
TABLE 17.	Required Disk Directory Structure for II Development Tools.	124
TABLE 18.	SBC6x External Memory Map	127
TABLE 19.	SBC6x Bus Control Register Initialization Values	127
TABLE 20.	SBC6x I/O Bus Memory Mapping	128
TABLE 21.	I/O Bus Power Ratings	130
TABLE 22.	FIFOPort Level Status Register Definition	132
TABLE 23.	FIFO Port Timing Parameters	135
TABLE 24.	UART Control Registers	136
TABLE 25.	UART Control Registers	137
TABLE 26.	On-chip Timer Clock Source Control Register Definition	140
TABLE 27.	AD9850 Control Registers	140
TABLE 28.	Digital I/O Control Registers	141
TABLE 29.	Digital I/O Port Timing Parameters	142
TABLE 30.	External Mux Control Memory Map	142
TABLE 31.	External Interrupt Input Control Registers	143
TABLE 32.	External Interrupt Input 4 Source Select Register Values	144
TABLE 33.	External Interrupt Input 5 Source Select Register Values	144
TABLE 34.	External Interrupt Input 6 Source Select Register Values	144
TABLE 35.	External Interrupt Input 7 Source Select Register Values	145
TABLE 36.	Miscellaneous Control Register Definition	147
TABLE 37.	OMNIBUS Connector Pinouts	150
TABLE 38.	OMNIBUS Bus Connectors	151
TABLE 39.	I/O Module Bus Connectors	152
TABLE 40.	Digital I/O Connector	153

TABLE 41.	FIFOPort Connector	154
TABLE 42.	Processor Serial Port Connector	155
TABLE 43.	JTAG Debugger Connector	156
TABLE 44.	Power Input Connector	157
TABLE 45.	Multiplexer Control Connector	158
TABLE 46.	Asynchronous Serial Port Connector	159

List of Figures

FIGURE 1.	Hasp Key	21
FIGURE 2.	Terminal Emulator “UniTerminal” Applet	42
FIGURE 3.	UniTerminal File Menu	43
FIGURE 4.	UniTerminal DSP Menu	44
FIGURE 5.	UniTerminal Option Menu	45
FIGURE 6.	UniTerminal Display Options Tab	45
FIGURE 7.	UniTerminal DSP Options Tab	46
FIGURE 8.	UniTerminal Sound Options Tab	47
FIGURE 9.	UniTerminal Support Options Tab	48
FIGURE 10.	UniTerminal Help Menu	48
FIGURE 11.	The COFF Dump Utility	52
FIGURE 12.	COFF Dump Utility Output	52
FIGURE 13.	Creating a New Project in Code Composer Studio	65
FIGURE 14.	Adding Files to a Code Composer Studio Project	65
FIGURE 15.	Code Composer Studio Project Window	66
FIGURE 16.	Code Composer Studio Compiler Build Options	67
FIGURE 17.	Code Composer Studio Linker Build Options	68
FIGURE 18.	Code Composer Studio Build Results Window	68
FIGURE 19.	Code Composer Studio Compiler Build Options	69
FIGURE 20.	Code Composer Studio Build Results Window	70
FIGURE 21.	Code Composer Studio Compiler Build Options	71
FIGURE 22.	Code Composer Studio Linker Build Options for Non-Bios Project	72
FIGURE 23.	Code Composer Studio Build Results Window	72
FIGURE 24.	Code Composer Studio Compiler Build Options	73
FIGURE 25.	Code Composer Studio Assembler Build Options	73
FIGURE 26.	Code Composer Studio Linker Build Options	74
FIGURE 27.	Code Composer Studio Build Results Window	74
FIGURE 28.	Code Composer Studio Compiler Build Options	75
FIGURE 29.	Code Composer Studio Assembler Build Options	75
FIGURE 30.	Code Composer Studio Linker Build Options	76
FIGURE 31.	Code Composer Studio Build Results Window	76
FIGURE 32.	The MessageTest Example	87
FIGURE 33.	The VclMessage Example	90
FIGURE 34.	SBC6x Block Diagram	126
FIGURE 35.	FIFOPort Block Diagram	131
FIGURE 36.	FIFOPort Level Status Register	132
FIGURE 37.	FIFO Port Timing	135
FIGURE 38.	On-chip Timer Clock Source Control Register	139
FIGURE 39.	Digital I/O Port Timing	142
FIGURE 40.	Miscellaneous Control Register	146

FIGURE 41. Power Connector Pin Positions (side view, from front of connector, showing connector keying and locking tab along with printed circuit board position) 157

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/C++ code Compiler, Assembler, Linker, Debugger, 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:

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

Custom 32-bit Windows 9x/NT/2000 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 Innovative Integration software tools (please see the accompanying manuals for specific information on the T.I. toolset or Code Composer Studio 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:\SBC6711\TALKER\TALKER.OUT`

while user input and commands look like

ROM MYPROG.OUT

Installation of the Development Package consists of both hardware and software installation procedures. This document contains the complete installation directions for Innovative Integration's Development Package. This document also details the features of the Innovative Integration software generation tools, applets, and utilities for the target DSP board.

Refer to the online help file Host32.hlp for a detailed listing of available Host DLL functions and Zuma.hlp for a detailed listing of target peripheral library functions. Refer to the *Hardware* section of this manual for a discussion of hardware-specific configuration information.

The Development Package consists of software elements developed by Innovative Integration, Texas Instruments, and other sub-vendors. This document is intended to augment, not replace, the *Installation Supplement* and the documentation provided with the TI C compiler, Code Composer Studio, and other third-party software packages. Refer to the documentation provided with those products for a complete discussion of their features and use.

Warning: Do not install the hardware into the PC until the software and all related drivers have been completely installed. It is important to follow both the Software and Hardware installation sections in the order given in this chapter.

Host Hardware Requirements

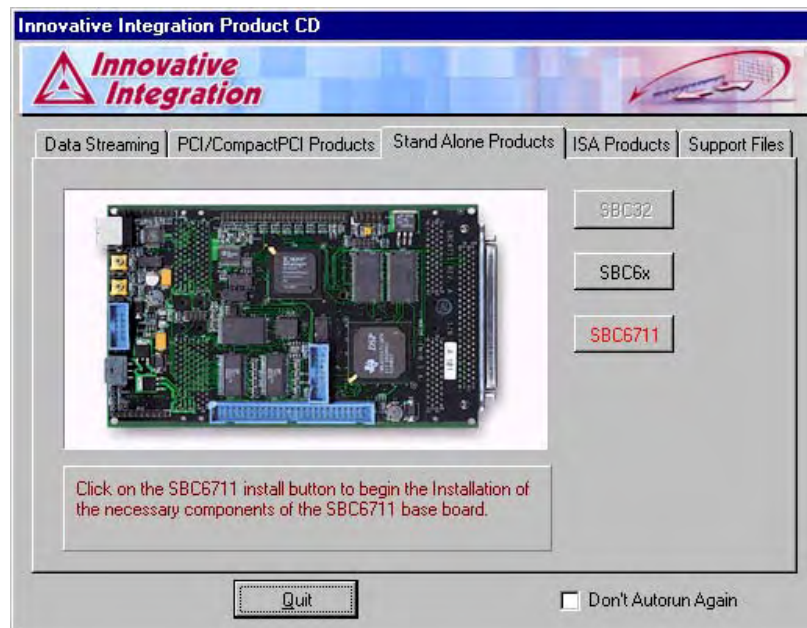
The software development tools for the Zuma toolset require an IBM or 100% compatible Pentium II-class or higher machine for proper operation. The host system must have at least 64 Mbytes of memory (128 Mbytes highly recommended), up to 84 Mbytes available hard disk space, a CDROM drive and a free PCI slot. Windows 9x, NT, 2k or XP (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.

Software Installation

The installation consists of the following major components: Zuma Toolset install and Code Composer Studio support. The installation time will take approximately 10 to 15 minutes depending on the system's speed. If you have not purchased some of the above listed components, you can go through and deselect the unavailable items. If Code Composer Studio was purchased, it must be installed before installing the Zuma Toolset.

Begin Installation

To install DSP board based products, the Logger PCI, or any of our support DLL's, start the host operating system and insert the installation CD. If the CD does not auto start, click on the <Start> button, then <Run>. Enter the path to the SETUP.EXE program located at the root of your CD-ROM drive, i.e. D:\SETUP.EXE. The setup program will run. Select the tab for the type of installation you are going to do. From there, select the exact product you wish installed. All necessary components including the Hasp key drivers, the board drivers, the peripheral libraries and debugging drivers will be automatically installed to your host PC during installation.



Important, Microsoft Windows NT/2K/XP Users Please Note: The installation of the Windows device driver requires that the logged-in user has Administrator rights on the system. This does not mean you have to be the actual Administrator login, as long as you have administrator rights.

Additionally, applications that receive interrupts from a target board must be *run* by a user with Administrator rights.

Launch Installation

The Zuma Toolset installation program will be launched. This installation program is set to install all the components needed for your Zuma Toolset development package. The various options to install, detailed below, can be individually checked or not checked. In the development phase, they all will need to be installed. However, later for distribution, maybe only the DLLs and Drivers are necessary. Click on the <Install> button to start the installation.



System DLL support Installation

The necessary DLLs and board drivers will be installed automatically for you. Do not uncheck this install unless you have previously installed the DLLs and are re-installing some other portion such as the applets. This option may be used alone for application deployment at a subsequent time to development.

Code Composer support Installation

If you do not have Code Composer installed this will not be a default install. This install includes the board configuration files necessary for this board to work properly in Code Composer.

Innovative Integration Development Packages include a JTAG-based, hardware-assisted C/Assembler Source Debugger called Code Composer Studio and a PCI style JTAG debugger which is a plug and play device. The JTAG debugger drivers are automatically installed for you and there is no acknowledgment window. Also, an applet, JtagDiag.exe (a Jtag diagnostic tool) will be installed.

Before installation of files, you will be asked to indicate an install path. The default path will be displayed. Unless you have a specific reason for changing it, the default path is preferred. This is espe-

cially true if you have Code Composer already installed. The path that you chose for that install is displayed as your default and should not be changed. Innovative Integration highly recommends that the default installation drive and directory be used whenever possible. The Code Composer workspace files for the sample DSP applications have been setup with the default directory paths in mind. If an alternate drive or directory is used, the workspace project setups will need to be changed to reflect the new path. See the Code Composer documentation for more details on the use of projects and workspaces.

If you have purchased an Innovative Integration hardware-assisted debugger, additional hardware, software and documentation have been included in your shipment. You should have received the following:

Item	Function
JTAG debugger board (PCI-bus compatible)	This is the PCI-bus-compatible JTAG emulator host interface board which plugs into your PC to allow communication with the target DSP over the JTAG scan path.
Target interconnect host cable	This provides an electrical connection between the host interface board and the target digital signal processor CPU.
PCI JTAG pod/pod target cable	PCI Debugger pod and pod target cable.
Code Composer Debugger package (included on installation CD)	This is the host software, which implements the debugger interface. Custom versions exist for each different DSP family - C2x, C3x, C4x, C5x and C6x.

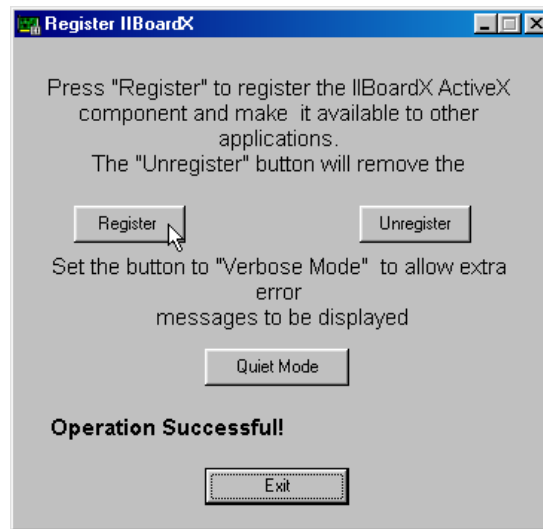
TABLE 1. PCI Debugger Package Contents

Dsp Component support Installation

Borland C++ Builder users and Microsoft Visual C++ users will get the necessary DLLs and components to have simplified access to the DSP board controlling the DLL from within the host application written in MSVC or Borland C++Builder. If you do not have one or the other of these two IDEs, the install will detect such and not install for that IDE. If you have both, it will install for both.

The registering of the OCX will occur automatically with the DspComponent install. This allows execution of the Microsoft Visual C examples, which use the OCX control.

Note: If you plan to use the OCX and it does not appear to have been properly installed, go to the start menu. Under programs, Innovative, find the DspReg program and run it. Click the **<Register>** button. After the OCX is registered, click the **<Exit>** button. This will register the OCX.



Applets support Installation

BinView, a binary file viewer and several other applets including UniTerminal, an applet to allow downloading of COFF files to the DSP are installed. Burn, Coffdump, PciConfig, PromImage, UniDownload and UsbView are also installed to your computer's "Start - Programs menu. These applets are provided to allow added functionality in the development cycle.

Peripheral Libraries and examples Installation

The Innovative Integration Peripheral Library is included with the purchase of all Development Packages. The Peripheral Libraries include example DSP (target) software and a complete set of peripheral control libraries as well as sample host applications for use in host code development.

Before installation of files, you will be asked to indicate an install path. The default path will be displayed. Unless you have a specific reason the default path is preferred. Innovative Integration highly recommends that the default installation drive and directory be used whenever possible.

HASP key Installation

The Hasp key is a parallel port "dongle" type key. It is included in your development package. It should be attached to your parallel port and this install should be checked. See the section later in this manual regarding the Hasp activation.

Click **<Install>** when you are satisfied with the elements of the install.

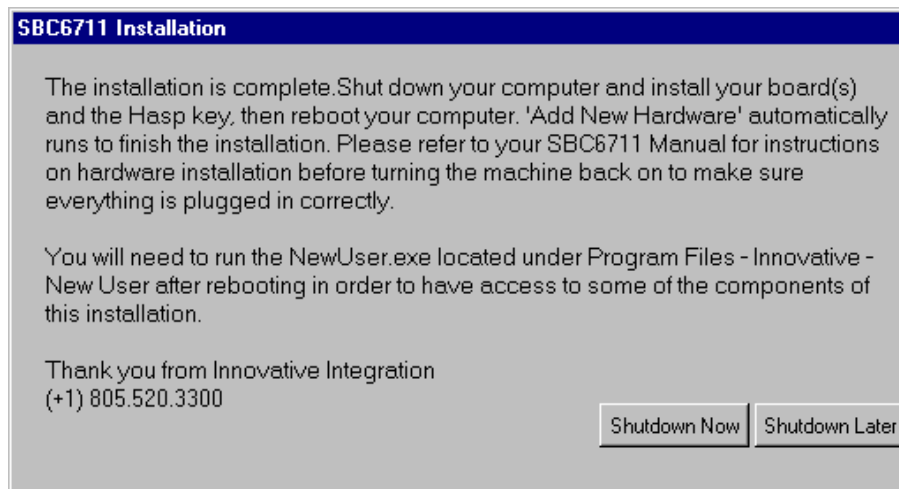
Hasp Key Activation

Upon completion, the Install Utility will show this screen below. It is simply an acknowledgment that the Hasp install went successfully. Your computer must be restarted before any changes made will take effect. Click the <OK> button, it will NOT restart your computer.



If you need at any time to run the Hasp Device Driver Installation Utility, simply run this installation again, unchecking all but the Hasp install.

Now you will see the final screen for this installation, shown below.



Exit by clicking on the <**Shutdown Now**> button or the <**Shutdown later**> button.

If you choose to shut down now, this would be the best time to install the hardware including JTAG debugger, if purchased. Upon restart, your hardware should come up as plug n play in the hardware wizard.

ReadMe Files

Release Notes for the libraries and applets are located in a documents subfolder of the main board's folder. Go there to view release notes on various products which may have changed in this release.

At this point you have finished the software installation process. Next, you are advised to go to the Hardware Installation section of this manual. This will conclude the Innovative Integration Development Software Package installation. Remove the CD from the drive and shutdown your computer system in preparation for installing the hardware.

Hardware Installation

The software components of the Development Package have been installed. To proceed with the Development Package Kit installation, it will be necessary to configure and install your hardware.

JTAG Emulator Hardware Installation

First, the emulator hardware must be configured and installed into your PC. The emulator hardware is described in the table below:

Type	Features
Pod-based	Uses a special ribbon cable with integrated line drivers to connect the target DSP emulation signals to the JTAG debugger card. Usable on 3.3 volt or 5 volt designs. (Including 'C54x and 'C6x)

PCI Pod-Based Emulator Installation

To install the PCI pod based emulator, follow the instructions below:

- Shut down Windows and power-off the host system.
- Touch the chassis of the PC to dissipate any built up static charge.
- Securely install the JTAG board into the host computer.

- Connect the host pod cable from the JTAG board external connection on the end bracket to the JTAG pod connection. Then, connect the target cable from the JTAG pod to the target DSP card connection.

DSP Board Installation

Innovative Integration makes DSP products that fall into three basic categories. Hardware installation directions are given below for the target card. When installing the target card:

1. Power off the host system and **touch the chassis** of the host computer system to dissipate any static charge.
2. Remove the DSP card from its protective static-safe shipping container, being careful to handle the card only by the edges.
3. If you have an SBC type card, place the card in a “ESD” or static safe workstation for software development. For PCI or ISA cards, install them in the appropriate slot of your computer.
4. If you have a card equipped with a USB connection, connect the Type-A end of the supplied USB cable to the host computer. Connect the Type-B end of the USB cable to the standard Type-B receptacle connection.
5. If you have a single board computer (SBC), connect the power supply provided with the *Developers Package* (or an equivalent power supply) to the board’s power connector. The power supply included in the *Developers Package* is a triple output model and has been fitted with a connector, which mates directly to (JP16) on the SBC6711.
6. Connect the target cable from the JTAG board to the DSP board’s JTAG connector marked (JP17).
7. Securely install the Hasp Key (see figure below) provided with your board into a parallel port now, usually LPT1. UniTerminal will not run without this key. **IMPORTANT:** Make sure you have installed some sort of parallel device, usually a printer to LPT1 in Windows. Sometimes without this printer install, Windows will not recognize the parallel port even with the Hasp key attached.
8. After completing the hardware installation, boot your PC.
9. The target card is plug and play, which Windows will detect at start-up. **NOTE:** Sometimes with the SBC6711, you may have to reset the card after Windows is running in order for the device manager to recognize and enumerate the card.

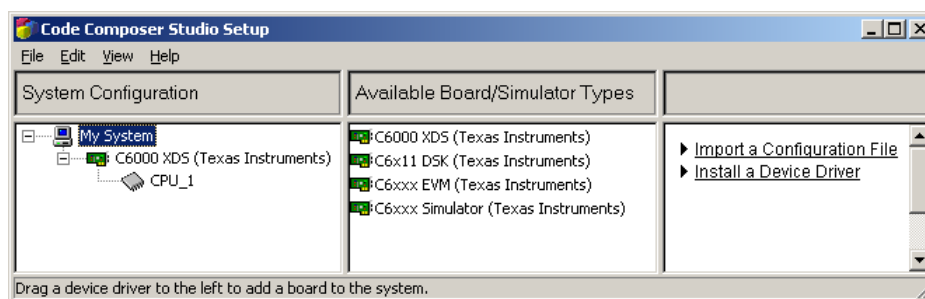
Note: If you are installing on a Windows operating system, the Add New Hardware Wizard will guide you through the Baseboard or JTAG universal device driver installation. **NOTE:** When asked for the location of the universal driver .sys file, explore to the C:/Windows/System32/Drivers directory for the driver file. On a Windows 2000 or Windows NT machine, the file will be c:\WinNT\System32\Drivers.



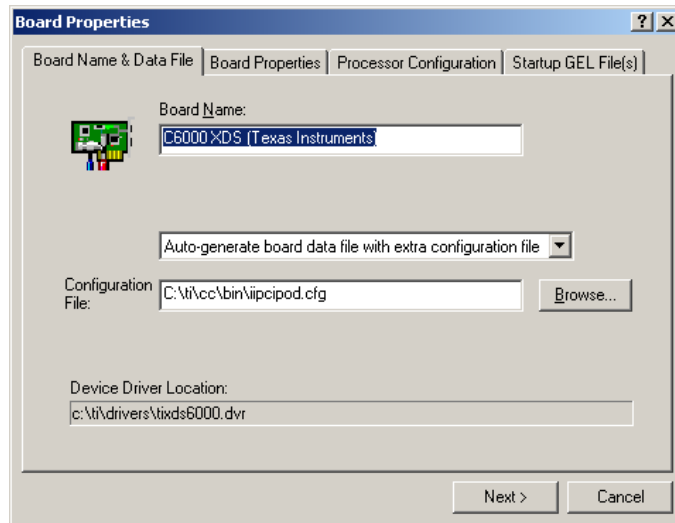
FIGURE 1. Hasp Key

Code Composer Studio Setup

To setup Code Composer Studio and activate the driver, the Code Composer Setup Utility must be run. Code Composer Studio setup must be configured to use the XDS510 driver for the C6000 (remove the default driver from the System Configuration). This driver is named C6000 XDS within the Code Composer setup utility. In the figure below, it is the first item listed in the Available Board/Simulator Types column of the Setup program.

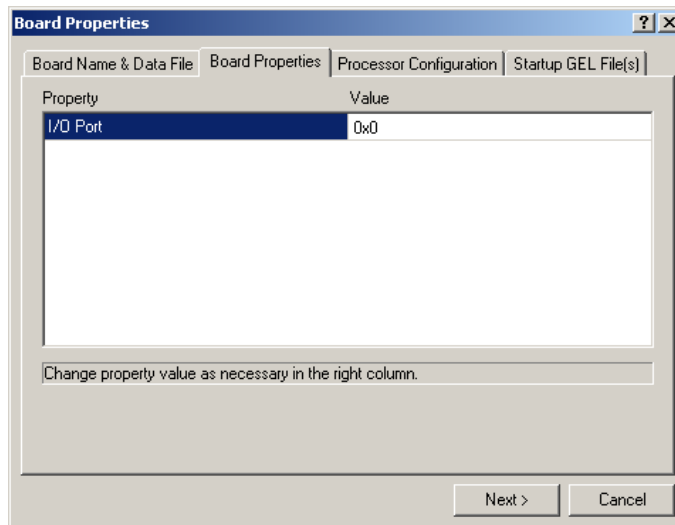


Click this driver from the “Available Board/Simulator Types” control within the setup utility and drag it into the “System Configuration” control. Then, right-click on this C6000 XDS object to invoke the Properties Dialog for the driver.

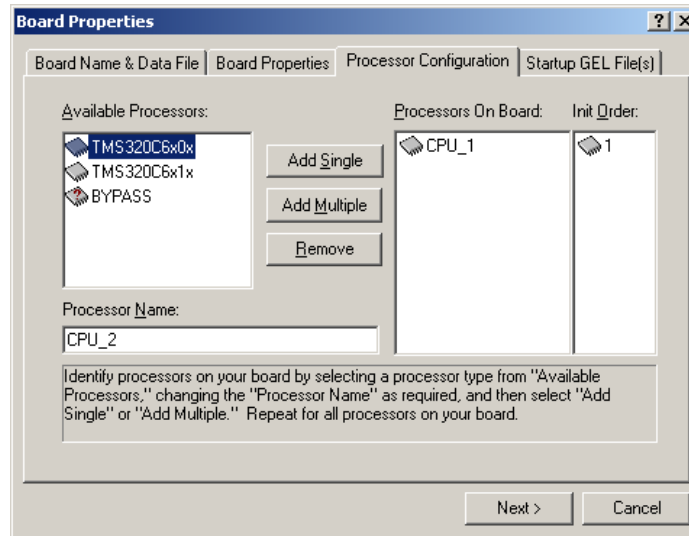


Under the “Board Name & Data File” tab, the board name edit box should list "C6000 XDS (Texas Instruments)". The “Configuration File” combo box should be changed to "Auto-generate board data file with extra configuration file". The “Configuration File” edit box should be changed to "<drive>:\ti\drivers\IIPciPod.cfg", where <drive> is the letter for the drive onto which CCS 2.0 was installed.

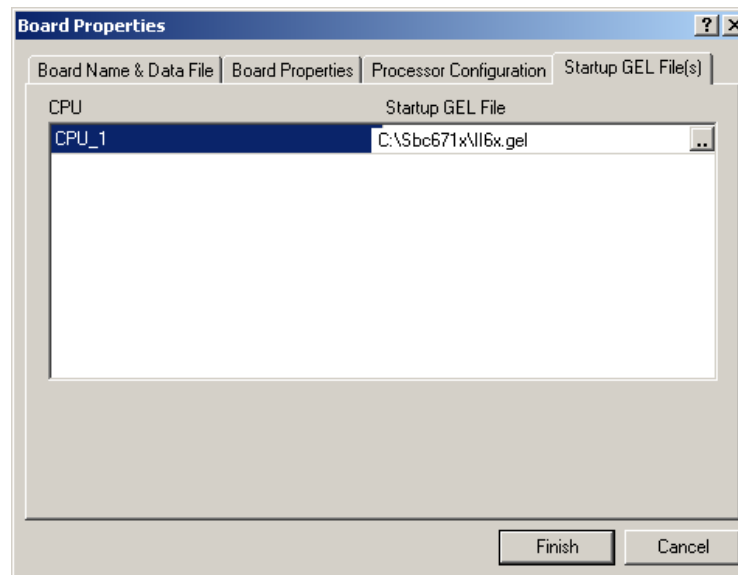
Under the “Board Properties” tab, the I/O port value for the driver should be set to virtual device address “0”.



Under the “Processor Configuration” tab, one or more processors of the appropriate type (C6x0x or C6x1x) should be added to the “Processors on the Board” list box.



Under the Startup Gel Files tab, the Startup Gel File combo box should be the board-specific initialization GEL script for the target board. Innovative supplies an appropriate, board-specific GEL initialization file for each product located in the root Zuma toolset directory. For example, for the M6x, this should be set to `c:\M6x\II6x.gel`.



Finally, the setup configuration should be saved. After saving the configuration and shutting down the setup tool, Code Composer Studio should launch successfully. If you encounter difficulty launching CCS 2.0, Run the JtagDiag.exe utility provided in your Zuma toolset to reset the debugger interface. Then restart Code Composer Studio.

As a consequence of these steps, a new ccBrd0.dat file will be created. This file has been customized for the particular target being used.

Testing the Development Package Installation

At this point, all of the core software and hardware elements of the Innovative Integration Development Package have been installed. Through this section, <target directory> represents the target boards directory (example C:\M4x or C:\M6x). In order to test your installation, follow the instructions below.

Configuring the Applets within the Development Package

Each of the Development Packages are supplied with several, standard Windows applets, which are used to perform common functions with the DSP board. These standard applets include:

Applet	Function
UNIDOWNLOAD.EXE	Application to download a debugged DSP application to a DSP target board without using JTAG debugger.
UNITERMINAL.EXE	Application to act as a terminal emulator to standard I/O requests posted by the target DSP board during target executing.
COFFDUMP.EXE	Application to display memory usage of target executables.
BURN.EXE	Application to support burning application code and Talker code into FLASH ROM on FLASH-based DSP products.
USBVIEW.EXE	Displays information gathered during most recent USB device enumeration cycle.
PROMIMAGE.EXE	Generates flash-loadable executable image from debugged COFF .out file.
JTAGDIAG.EXE	Initializes JTAG debugger interface for subsequent debugging session using Code Composer Studio.

TABLE 2. Host Support Applications

These applets are located on the Start menu | Programs | Innovative | <board type>, submenu.

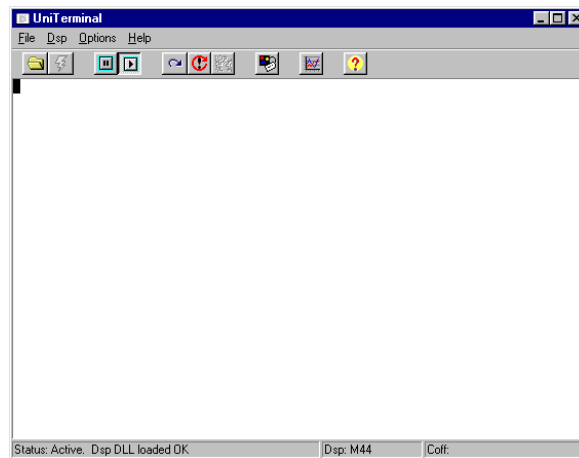
The target applets can be configured based on single or multi-board system:

For single DSP board systems (“Directly through the applet” on page 25)

For multi DSP board systems (“Through the Start menu command line” on page 25)

Directly through the applet

For single DSP board systems, the applet can be configured through the applet itself. Start the applet by simply double-clicking the applet from within the program group to run it. For instance, “C:\<target directory>\UniTerminal.exe”. Then, to modify the DSP board type or Target Number from within the applet, click on the **Option/Edit** menu. Under the DSP tap, select the target board type & target number, and click OK. If the configuration is correct, the “Status: Active. DSP DLL Loaded OK” will be displayed at the bottom of the UniTerminal applet as shown below.



It is important to note that on single-board DSP's, target zero refers to serial port COM1 and target one refers to serial port COM2, etc.

Through the Start menu command line

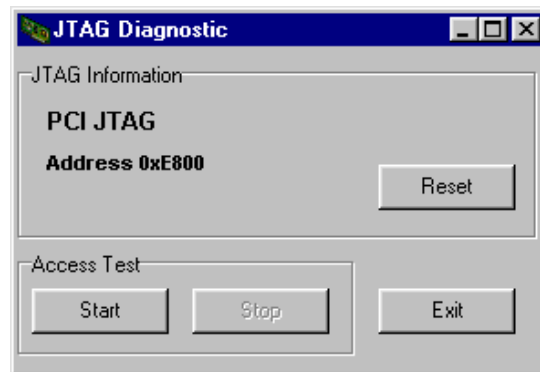
For multi DSP board systems, the applet can be configured from the “Start Menu” in order to run the applet for each target board. For instance “Start | Programs | <target directory> | UniTerminal”. If you need to modify the Target Number of the DSP Board, then you must modify the shortcut command line to the applet to use the correct Target Number. Bring up the Properties sheet for the shortcut of the applet by right-clicking UniTerminal off the Start menu.

In the UniTerminal properties window, select the “Shortcuts” tab. To set the target board and target number modifying the “Target” to read “C:\<target directory>\UniTerminal.exe -t1 -b<Target Board>”. Click <Apply> and then <Close>. For additional information on the arguments used in the command line refer to the help files provided in the application.

As a variation to the above theme, you may want to launch a program on a particular target DSP board each time the PC is powered up. This can be done by configuring the shortcut command line and placing it in the startup directory.

Running the "JTAG Diagnostic" Utility

To verify that the JTAG is functioning properly, open the JTAG Diagnostic by right-clicking the <Start> button, clicking <Open>, double-clicking <Programs>, double-clicking on the <target directory>, and double-clicking on <JTAG Diagnostic>. The JTAG Diagnostic utility should run, as seen below.



Then press the <Reset> button before running the test.

If the following 2 items are true:

- The JTAG card must be properly jumpered to match the Input/Output Range specified by Windows (ISA based JTAG only).
- The JTAG Diagnostic "JTAG Address" must have the same setting assigned to the JTAG device.

If the above conditions are all met and the JTAG is operating properly, then the START/STOP test will cause the LED on the JTAG card to blink slowly. Click the <Start> button. This should cause the LED to blink. When you are satisfied that it is operating correctly, click the <Stop> button.

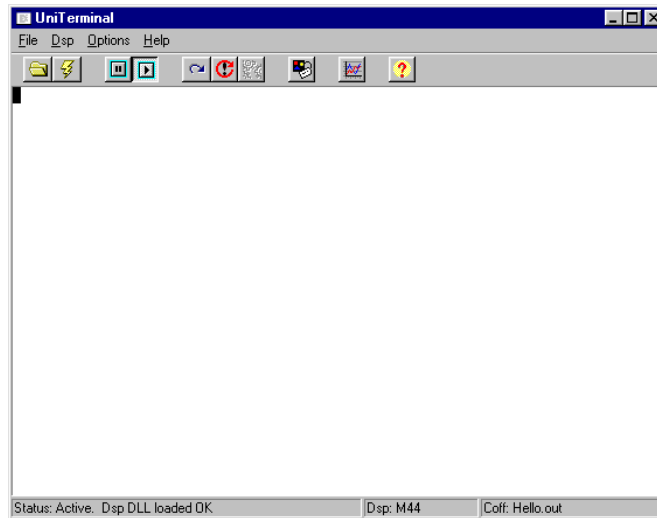
Close the Jtag Diagnostic utility by clicking the <Exit> button.

Running an Example Program using UNITERMINAL

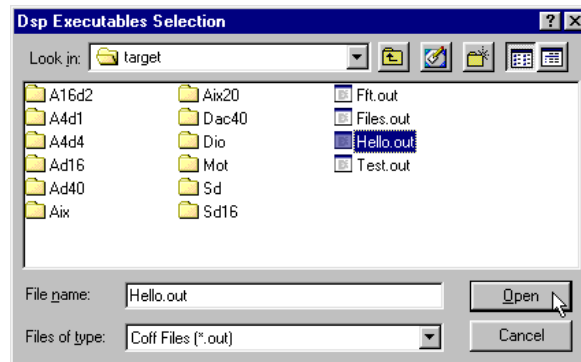
Each of the Development Packages is supplied with a terminal emulator application called "UNITERMINAL", which can be used either stand-alone or in conjunction with Code Composer Studio. The terminal emulator application is a small, Windows applet, which acts as a receptacle for standard I/O requests generated by a target DSP application. Refer to the "Support Applets" section of this Manual for detailed information on the UniTerminal application.

Invoke the UNITERMINAL utility now. If successfully started, UniTerminal will display "Status: Active. DSP DLL Loaded OK" at the bottom of its client window. If the Talker fails to start, refer to the

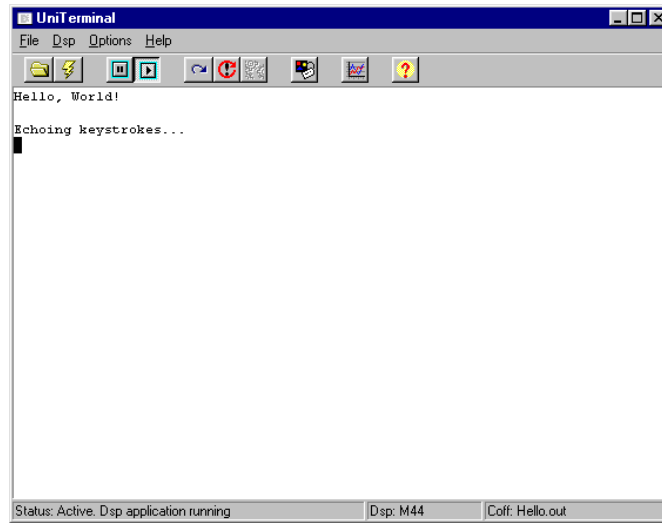
Troubleshooting section of this manual. Reiterate this step until Talker initializes successfully. You should see a window similar to the following:



From the <File> menu, select <COFF File> then <Download>, to begin downloading a program to the target DSP. This will open a dialog box from which you can select a target DSP program to run. The examples can be found in the C:\<board directory>\examples\target\ directory:



Select **HELLO.OUT** (.MPO file for the Quatro6x boards) from the file list and click <Open> to download and run the classic “Hello World!” program to the target DSP. The UniTerminal should display “Hello World!”, as shown below:

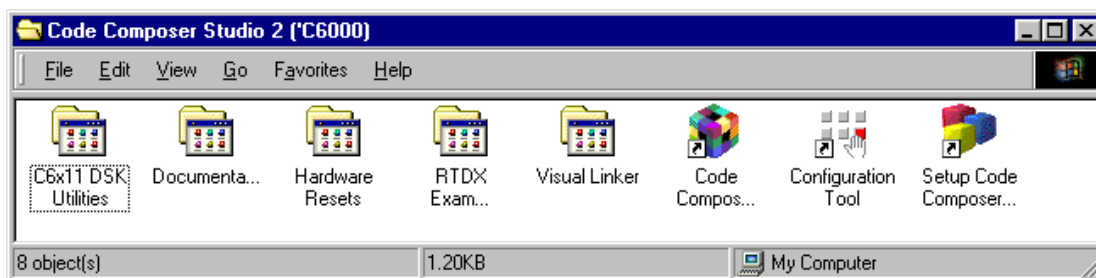


If so, close the “UniTerminal” program and proceed to running the next test application. Otherwise, refer to the Troubleshooting section of this manual for the most frequently asked questions and solutions.

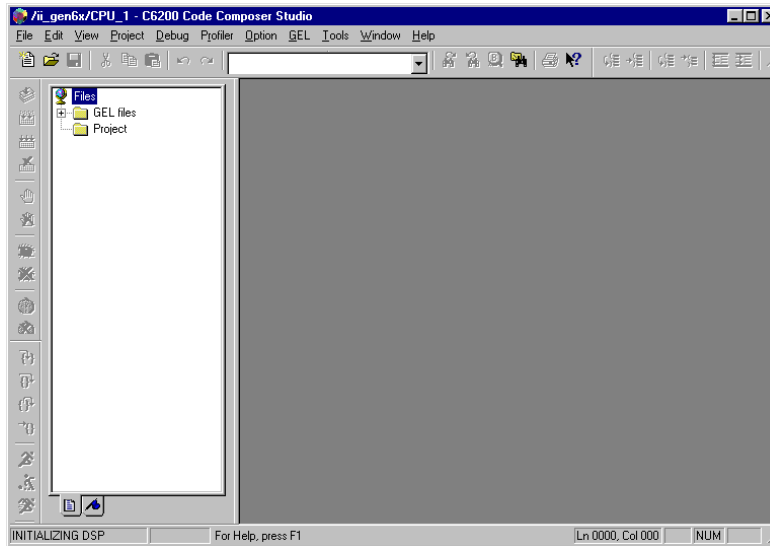
Testing the Code Composer Debugger

If you will be running an application which employs standard I/O (i.e. most of the I.I. example programs), start UniTerminal.exe. (Refer to **Running an Example Program using UNITERMINAL** for instructions) Note that UniTerminal must be launched prior to Code Composer Studio because UniTerminal physically resets the target DSP board during its initialization, which disrupts the JTAG hardware used by Code Composer Studio.

Next, open the Code Composer folder by right-clicking the <Start> button, clicking <Open>, double clicking <Programs>, and double clicking (opening) the Code Composer Studio folder. This should open a window containing icons similar to the following:



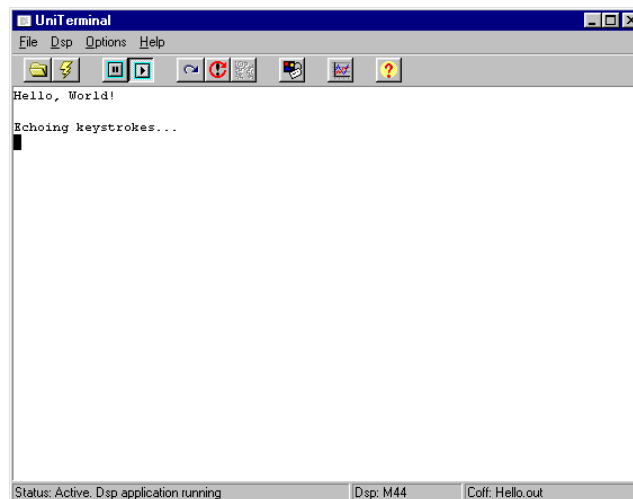
Double-click on the Code Composer icon to launch the debugger. You should see a window similar to the following:



If you do not see the above window, refer to the Troubleshooting section of this manual for the most frequently asked questions and solutions.

If you do see the above window, load and run the HELLO.OUT target application as described below from Zuma Toolset Examples\Target directory. To load, use File | Load Program and to run, use Debug | Run.

The DSP target application should run, displaying "Hello World!" in the UniTerminal window:



You have successfully run your first DSP program from within the Code Composer Studio Source Debugger environment! Refer to the Code Composer Studio documentation for complete instructions on how to take advantage of all the features within the debugger software.

Troubleshooting Installation Problems

Most Commonly Asked Questions

This section includes answers to some of the most commonly asked question relevant to installation and initial testing. If after troubleshooting, components of the Developer's Package still do not operate correctly, contact Innovative Integration for technical support.

I already had a licensed copy of the TI tools, so I omitted them from the Development Package. Whenever I attempt to compile, assemble or link a program from within Code Composer Studio, the build window shows “Bad command or filename” errors.

Edit your **AUTOEXEC.BAT** file to add the directory containing your TI toolset to your default path, i.e.:

```
path = c:\windows;c:\windows\system;c:\fltc
```

Code Composer Studio won't start. It shows a dialog box that says, “Can't initialize target DSP. Trouble with JTAG controller. Please insure the I/O port is set properly.

There are several common reasons for this error. Verify each of the following:

- You have properly configured the ISA JTAG debugger board according to the I/O assignment produced by Windows during the ISA JTAG device driver installation and that all jumpers are properly oriented for communication with your target.
- Verify that your JTAG cables are properly connected to the host and DSP target board.
- ISA DSP board users: Verify that you have properly configured the DSP target board according to the I/O and interrupt assignments produced by Windows during the DSP board device driver installation.
- Stand-alone DSP board users: Verify that the DSP board is powered up and the supply voltages are correct. If you are using a serial mouse, change the target number setting in the UniTerminal application.

- You may have selected the incorrect driver for your DSP target within the Code Composer Studio setup utility. If you are using a multiprocessor target, check the multiprocessor settings as well.
- The Baseboard either is in Reset mode or doesn't have a program running in the CPU. By running the hello project in UniTerminal, you ensure that both the board is not in Reset and a program is running in the CPU. To achieve this, close Code Composer Studio, run UniTerminal, download the Hello project, then start up Code Composer. If Code Composer Studio starts up, UniTerminal can be close.
- When using the pod-based debugger with an JTAG pod, make sure the target is set up to provide the 'C3x H3 clock (see DSP card *Hardware Section* for details).
- Verify that the JTAGDIAG "Access Test" passes. Launch the "JTAG Diagnostic" from the program folder of the board and click the <Reset> then <Start> button
- Check for the proper and most current DLL driver available.
- Verify that DLL being used is as new as the Innovative Intergration's website (www.innovative-dsp.com) version.

When I attempt to start Code Composer Studio, my PC "hangs" and won't respond to the mouse or keyboard.

If you are using a C3x-based DSP board, insure that the JTAG cable is properly connected between the debugger board's C3x JTAG connector and the target DSP board's JTAG connector.

For all other targets, insure that the JTAG board clock select is configured for OSC and that the on-board oscillator is seated in its socket.

I have checked and re-checked the settings for my JTAG board and it's connections to the DSP target, but Code Composer Studio still won't start.

The ISA/PCI card edge connector on the JTAG board may be dirty. Clean the ISA/PCI card edge connectors for the JTAG board using a pencil eraser until the edge connector is free of any film or residue. For stand-alone boards make sure that the DSP power is on.

I can't seem to load any of the Code Composer Studio workspaces for the Development Package example programs.

The project workspaces (*.WSP files) were created for proper execution when the DSP board directory exists on the C:\ drive. If you installed your DSP board directory onto another drive, you will have to recreate each of the project workspaces. However, it is possible to edit each of the project make files (*.MAK), modifying all drive letter designators in order to allow them to work on another drive.

Code Composer Studio appears to operate properly (I can load, execute and step through programs), but standard I/O doesn't appear on my UniTerminal window when I run the example programs.

Insure that the UNITERMINAL applet is configured to communicate with your target board. For single-board users, the target number corresponds to the PC com port being used (target 0 = COM1, target 1 = COM2, etc.). For all other targets, the target number is usually zero. If you need to edit the applet target board or target number, refer to the "Configuring the Applets within the Development Package" section for complete instructions.

I have installed a PCI-based DSP card and now my PC won't boot.

You do not have an available, uncommitted IRQ for use by the PCI card. Enter the system BIOS setup and reserve an IRQ for use by the DSP board.

I have installed a PCI-based DSP card and my PC boots. But an ISA adapter card in my system (network card, etc.), which used to work fine, is no longer operating.

The PCI BIOS has assigned the DSP board an IRQ that was already in use by the ISA board. Enter the system BIOS and reserve an IRQ for use by the DSP board.

Our host application is not Windows-based. We're using DOS, UNIX, OS2, etc. for our host environment. How can we develop and debug my target application?

Install Code Composer Studio and the TI tools onto a Windows-based PC and umbilical over to the DSP board installed in a second machine that is running under the "foreign" operating system. You will not be able to run UNITERMINAL and UNIDOWNLOAD under the foreign OS, but the Windows-based system can be used to develop and deliver code to the target DSP over the JTAG link.

I get a "Talker didn't start!" message when attempting to download to my single-board DSP from within UNITERMINAL or UNIDOWNLOAD.

- Insure that the applet is configured to communicate with your target board. For single-board users, the target number corresponds to the PC com port being used (target 0 = COM1, target 1 = COM2, etc.). For all other targets, the target number is usually zero. You may need to configure the applets, refer to "Configuring the Applets within the Development Package" section.
- For Stand Alone boards, verify that the board is getting power and the serial cable is well seated. Also, make sure all external reset sources connected to the board are not stopping the card from running.
- Verify that the jumpers on the card are set to the factory defaults.

- Check that the COM port in use is enabled at the BIOS level.
- Check that the port is enabled and available from within the Windows Device Manager
- In Code Composer Studio, click on Debug|Halt, then Debug|Run Free, then close CC Studio.
- In UniTerminal, click on Options/Edit menu, then the DSP tab and verify that the Init JTAG on Boot box is checked if you are using an I.I. debugger system.

After installing a stand-alone board, my serial mouse no longer works.

UniTerminal is using the same target number as your mouse (target 0 = COM1, target 1 = COM2, etc.). Change the target number setting in the UniTerminal application, refer to “Configuring the Applets within the Development Package” section.

After installing a stand-alone board, my computer won’t reboot, I get a keyboard error at boot up, or Windows hangs at start up.

Your DSP may be stuck in a bad state. Turn off your computer. Remove power to the board. Turn on your computer and wait for windows to boot (if prompted to start in safe mode, ignore the message and do a normal boot). Once windows has started, power up the board.

I have downloaded a new driver for my board from you FTP site. How should I install it?

Re-run the driver installation as documented earlier in this document. Your old driver will be overwritten during the installation.

My embedded application runs well, but when I download the same application form Code Composer Studio, it does not run.

For SBC6711 cards, remember that the linker command file is different for the host download (generic.cmd) and for embedded operations (embed.com). (Refer to “Creating a Binary File” section)

Installing Code Composer Studio separately.

The path in the autoexec.bat must be change manually if Code Composer Studio is installed in a different directory or different drive then the default.

Code Composer Studio Troubleshooting

If Code Composer Studio did not install or operate properly, it may be helpful to refer to the Code Composer Studio instruction manual for additional information.

Code Composer Studio requires third party device drivers to be installed along with the executable application in order to support Innovative Integration's debugger hardware. Therefore, if these device drivers did not install properly, Code Composer Studio may need to be configured for use with the Innovative Integration DSP board you have purchased before being run (refer to "Code Composer Setup" section).

Please Note: *If the target is a multiprocessor DSP card*, then the number of processors entered in the scan path list **must** be equal to the actual number of processors in the emulator scan path. Note also that order of the CPU IDs must match the order of the CPU's in the JTAG scan path. This is accomplished by entering the identifiers in what appears to be reverse order (with `cpu_2` before `cpu_1` and `cpu_3` before `cpu_2`) in the *Processor List*:

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 Studio for multiple 'C3x targets. Since the modified JTAG standard used on the 'C3x processors does not support multiple processor debugging, Code Composer Studio 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. Go through the normal installation of the support software per the instructions above.
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 UniTerminal 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, then 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]\UNIDOWNLOAD.EXE -t 0
```

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

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

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

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 start-up. If one of the shortcuts specifies a -d[FILENAME] switch in its property box, then that shortcut will download the specified filename on start-up, rather than the default application selected in the configuration utility.

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 the tools included in the development package and gives descriptions of each applets features and use. A brief introduction is given regarding the software programs provided and their use within the Developer's Package. The user is referred to the individual manuals accompanying these software products for complete documentation.

The Texas Instruments C Compiler Toolset

The compiler supplied with the Developer's Package is the Texas Instruments (T.I.) Floating Point C/C++ Compiler toolset for the DSP target board. 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 supports 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 Code Composer Studio editor/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 as described in the TI toolset documentation. Also included in the installation directory are make files useful for manually rebuilding applications programs within the DOS environment. To rebuild an example or library from the command line, execute **gmake -f <filename.mak>**.

Code Composer Studio

Code Composer Studio is a flexible, high-performance, integrated code generation environment developed by Texas Instruments and bundled into the Innovative Integration Zuma toolset. For complete documentation to the features of Code Composer Studio package, refer to the accompanying *Code Composer Studio Manual* provided. If the user wishes to compile outside of Code Composer Studio (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.

Editor

Code Composer Studio supports code editing and emulates the most popular editing packages (CUA, etc.). Code Composer Studio is also a Window editor. Custom DSP code development can take place entirely within the Code Composer Studio environment using its project management tools to place source files, libraries and linker command files into projects (.MAK) in order to build executables. The example programs included in the Developer's Package each have a Code Composer project file (.MAK) associated with them which may be used to re-compile the example.

Debugger

Code Composer Studio is a software program for high-level TI C and Assembly Language debugging which supports high-performance, JTAG or MPSD-based hardware assisted debugging directly on the target DSP. This allows the user 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 Studio software package is a code management subsystem for editing files as well as creating and compiling DSP projects.

The Developer's Package includes a number of support applications supporting general DSP development: the terminal emulator (UNITERMINAL.EXE), the COFF file downloader (UNIDOWNLOAD.EXE), the COFF file display utility (COFFDUMP.EXE), the PromImage utility (PROMIMAGE.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. See the discussion below for applet-specific parameters.

The Terminal Emulator “UniTerminal”

The terminal emulator “UniTerminal” provides a C language-compatible, standard I/O terminal emulation facility for interacting with the `stdio` library running on an Innovative Integration target DSP processor. Display I/O calls such as `printf()`, `scanf()`, and `getchar()` are routed between the DSP target and the Host UniTerminal 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.

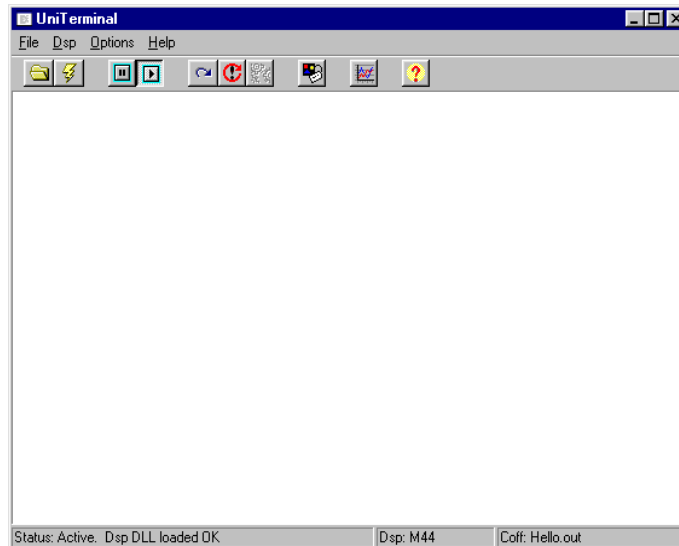


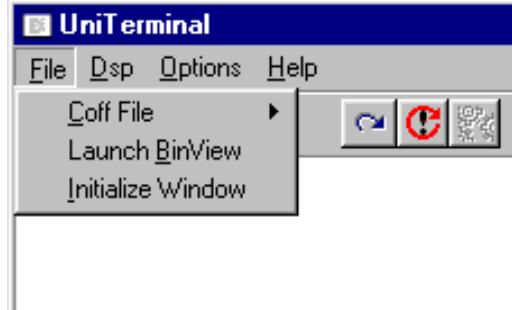
FIGURE 2. Terminal Emulator “UniTerminal” Applet

The terminal emulator is straightforward to use. The terminal 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. The 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 RS232 bus-based targets. The 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 `Zuma.hlp` file for the types and usage of these library functions as their usage is not 100% ANSI compliant.

Important Note: Before using UniTerminal, you must register your Zuma Toolset. Until you do so, usage will be restricted to a 20-day trial period for UniTerminal and other applets contained in the Zuma Toolset. To register, fill out the contents of the Registration Form, then click on the Register Now button. This will print a Registration report which, must be faxed to Innovative Integration. Innovative Integration will E-mail you an Access Code, which must be typed into the Registration Form for all the Zuma features to be enabled.

Terminal Emulator Menu Commands. The terminal emulator provides several menus of commands for controlling and customizing its functionality. These functions are available on the menu bar, located at the top of the UniTerminal main window. Speed button equivalents for each of the menu options are also available on the button bar located immediately beneath the menu bar. The following is a description of each menu entry available in the terminal emulator, and its effects.

File Menu:**FIGURE 3. UniTerminal File Menu**

- File | COFF File | Download - provides for COFF (Common Object File Format) program downloads from within the terminal emulator. When selected, a file requester dialog box is opened and the full 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.

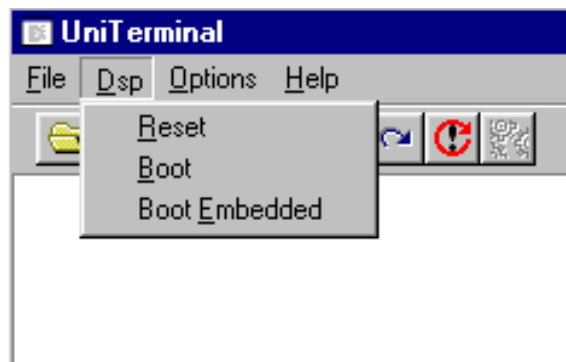
Quatro6x Users: Terminal supports downloading of either Coff files (.OUT) or multi-processor out (.MPO) files. The (.MPO) files provide a means of downloading separate (.OUT) files to multiple processors simultaneously, which greatly simplifies the task of synchronizing execution in a multi-processor environment.

- File | COFF File | Reload - Reloads and executes the COFF file last downloaded to the target. It provides a fast means to re-execute the application program most recently loaded into the target board.

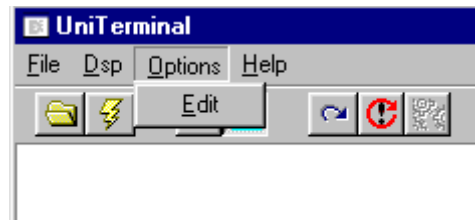
NOTE: File | COFF File | Download and File | COFF File | Reload functions physically resets the target DSP (in order to initiate the target Talker program) prior to the download. When using the terminal emulator in conjunction with the Code Composer debugger, use Code Composers File | Load Program facility 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, it is not reliant on the target Talker to perform the download.

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. 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 | Launch Binview – launches Binview, so that you may use it to examine the contents of a binary windows data file.
- File | Initialize Window - Resets the application window. Use this feature to reset the application if emulation ceases and will not resume, as an alternative to restarting the application. Emulation can be disrupted by outside programs, such as the JTAG-based Code Composer debugger.

DSP Menu:**FIGURE 4. UniTerminal DSP Menu**

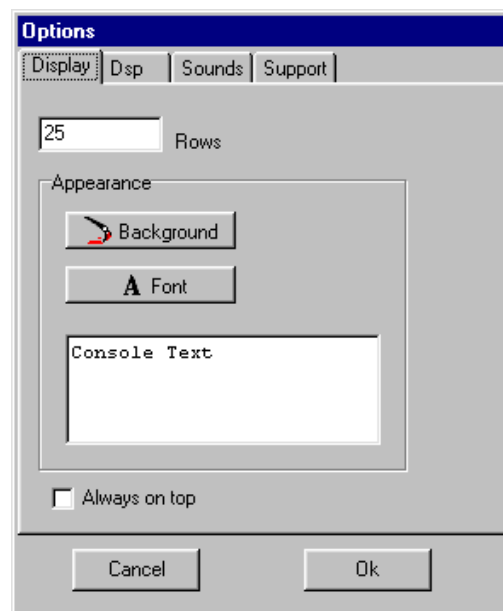
- Dsp | Reset - causes the terminal emulator to assert, then de-assert the target's physical reset pin, bringing the target board into a cold-start, uninitialized condition. This function is also available via the speed button toolbar. This is a special utility function which is not needed in normal operation.
- Dsp | Boot - performs the Dsp | Reset function, then boots the Talker program on the target. On plug-in cards, the Talker program is downloaded into target program memory, the target is reset (launching the Talker program), then UniTerminal polls for the Talker sign-on command via the stdio mailbox. On stand-alone, single-board targets, the target is reset which launches Talker from onboard Flash memory, then UniTerminal polls for the Talker sign-on command via the stdio mailbox. This function is also available via the speed button toolbar. This is a special utility function which is not needed in normal operation.
- Dsp | Boot Embedded - performs the Dsp | Reset function, simultaneously driving the CTS line inactive which causes single board targets to launch the application code embedded into sector one of their onboard Flash memory. This function is also available via the speed button toolbar. This is a special utility function which is only useful for single-board DSP users when testing embedded application programs.

Option Menu:**FIGURE 5. UniTerminal Option Menu**

- Options | Edit - invokes the user options dialog. The four, tabbed pages on the dialog box allow customization of display settings, dsp type, use of sound and specification of the location of support applets, such as BinView. This function is also available via the speed button toolbar.

Display Options Tab:

The display options tab on the Options dialog (seen below) contains controls to allow user-customization of the appearance of UniTerminal.

**FIGURE 6. UniTerminal Display Options Tab**

The Rows edit box controls the number of rows of text available within the terminal emulator window.

The Appearance group box contains separate buttons which are used to adjust the background color and font of the text within the terminal emulation window.

The Always On Top check box controls whether UniTerminal is forced to remain a foreground application, even when it loses keyboard focus. This is useful when running stdio-based code from within the Code Composer environment, when it's preferable to make terminal 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.

Display Options Tab:

The Dsp tab on the Options dialog (seen below) contains controls to allow selection of the type and number of the target board being serviced by this instance of UniTerminal.

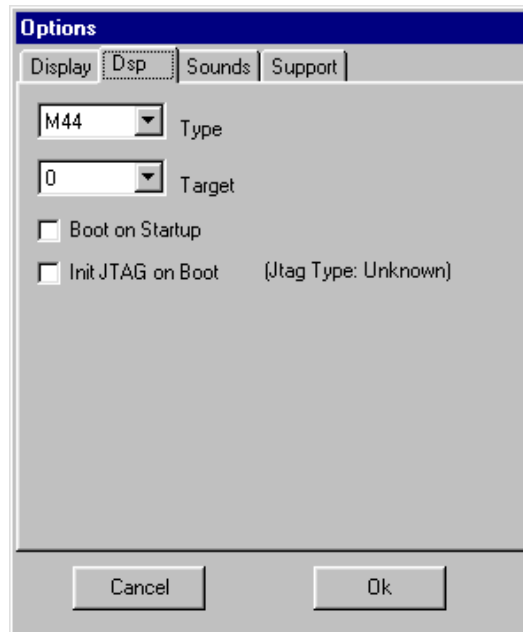


FIGURE 7. UniTerminal DSP Options Tab

The Type combo box specifies the type of target board being serviced by this instance of UniTerminal.

The Target combo box specifies the number of the target board being serviced by this instance of UniTerminal.

The Boot On Startup check box controls whether the target processor is booted when UniTerminal is initially invoked. Disable this check box when using UniTerminal in conjunction with Code Composer, to avoid resetting the target and corrupting the JTAG state upon UniTerminal invocation.

The Init JTAG on Boot check box controls whether the JTAG is reset when the DSP board is booted. If this box is checked, the JTAG reset will disconnect any prior logical connections upon running UniTerminal. Disable this check box when using any debugger other than Innovation Intergration's. This is it to prevent any communication problem between the target board and debugger.

Sound Options Tab:

The Sounds tab on the Options dialog contains controls to specify whether UniTerminal provides audible indicators during its operation.

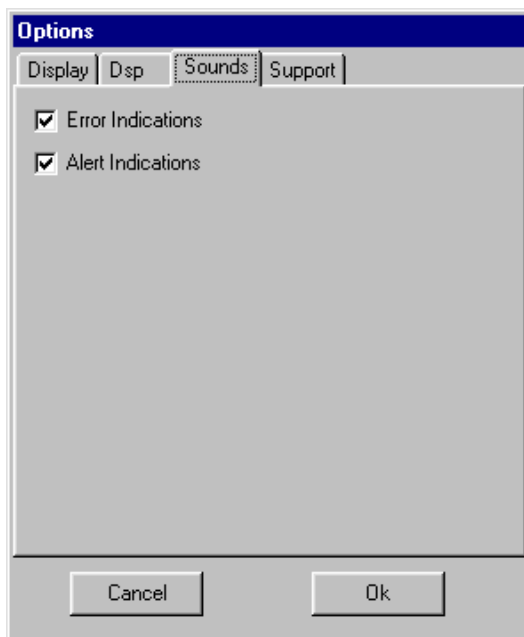


FIGURE 8. UniTerminal Sound Options Tab

The Error Indications check box controls whether sounds are played upon receipt of errors during emulation.

The Alert Indications check box controls whether sounds are played upon receipt of errors during emulation.

Support Options Tab:

The Support tab on the Options dialog contains controls to allow specification of the location of companion support applets used by UniTerminal.



FIGURE 9. UniTerminal Support Options Tab

The Binview Button is used to browse to the location of the BinView binary data viewer applet. UniTerminal communicates directly (using TCP/IP, which must be installed) with the BinView applet when target applications attempt to display graphical data files.

Help Menu:

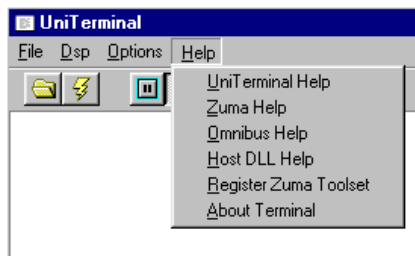


FIGURE 10. UniTerminal Help Menu

The Help menu provides convenient access to the technical resources/Help Files provided in the Zuma Toolset.

- Help | Register Zuma Toolset - invokes a dialog box to be filled out and sent to Innovative Integration
- Help | About Terminal - invokes a dialog displaying copyright and version information about UniTerminal, plus information pertaining to the use of Host resources by the target DSP board. Use this information when contacting Innovative Integration for technical support regarding UniTerminal.

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.

Multiple instances of UniTerminal may be invoked simultaneously in order to support installations utilizing multiple target boards. Instances of UniTerminal, after the first loaded instance must be configured via command line switches in order to properly communicate with their associated target.

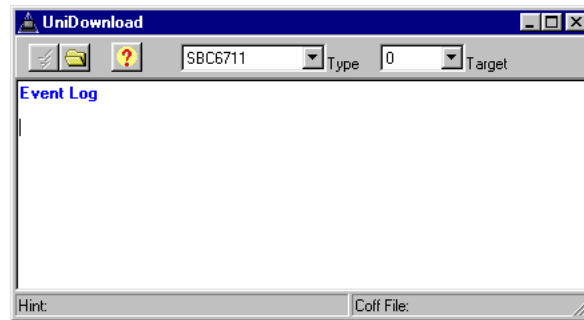
- **-boardtype** - Use the -b switch to force an instance of UniTerminal to communicate with a specific type of target board, boardtype. Supported boardtypes are listed below:

Board Type	Supported Product(s)	Communication Path
None	No DSP selected	N/A
ADC64	ADC64 DSP Board	PCI Bus
M44	M44 DSP Board	PCI Bus
PC31	PC31 DSP Board	ISA Bus
PC32	PC32 DSP Board	ISA Bus
PC44	PC44 DSP Board, up to two installed TIM processors	PCI Bus
PCI32	PCI32 DSP Board	PCI Bus
PCI44	PCI44 DSP Board, up to three installed TIM processors	PCI Bus
M6x	M62/M67 DSP Board	PCI Bus
Q6x	Q62/Q67 DSP Board with four C6x processors	PCI Bus
SBC31	SBC31 stand-alone DSP Board via serial port	COM1: or COM2:
SBC32	SBC32 stand-alone DSP Board via serial port	COM1: or COM2:
SBC54	SBC54 stand-alone DSP Board via serial port	COM1: or COM2:
SBC6x	SBC62/SBC67 stand-alone DSP Board via serial port	COM1: or COM2:
SBC6711	SBC6711 stand-alone DSP Board	USB

- **-tX** - Use the -t to allow the user to force the UniTerminal to interact with a specified target number. 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.
- **“filespec.”** - Use the **“filespec.”** field, which includes both the path and file name as a unit, to allow 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 field is particularly useful in situations where the UniTerminal is “shelled to” from within an other Host applications to facilitate the automatic execution of target applications employing standard I/O.

The COFF File UniDownloader

The UniDownload application provides a means of downloading debugged programs to a target DSP. The utility is useful in situations where a host application program capable of performing the DSP application download operation is unavailable or unnecessary.



Using the Application

The application may be operated through the use of buttons on the toolbar located at the top of the applet window, or through the use of command line switches. The latter technique is more common, since this allows the applet to be invoked within the Windows startup folder to cause DSP application execution to commence when Windows boots.

Command Line Switches

The following command line switches are supported. Each switch consumes an in line argument, which must be present for proper operation.

Switch Argument

-b name

-f "filename"

-t number

The **-b** switch is used to specify the name of the target DSP to which code is to be downloaded. The following DSP board names are acceptable: None, ADC64, M44, PC31, PC32, PC44, PCI32, PCI44, SBC31, SBC32, SBC54, M6x, Q6x, SBC6x, SBC6711. Case is significant and the name must be spelled exactly as shown. **Note:** The command line must be formatted such that a space character is present between the switch and its argument.

The **-f** switch is used to specify the name of the COFF executable image to be downloaded to the target. The filename parameter should be a full unambiguous pathspec to be image file.

The **-t** switch is used to specify the target instance number, 0..15. In a single target system, zero should be supplied.

Manual Operation

The applet may be operated manually using the buttons located at the top of the applet window. The operation of each of the buttons is described below.



The file open button is used to browse to and subsequently download a COFF image file located somewhere on the local machine.



The reload button is used to re-download the last COFF executable image which was successfully downloaded.



The help button is used to invoke the help documentation associated to this application.



The board type combo box is used to change the type of DSP target to which code is to be downloaded. The following DSP types are supported: None, ADC64, M44, PC31, PC32, PC44, PCI32, PCI44, SBC31, SBC32, SBC54, M6x, Q6x, SBC6x, SBC6711.



The target number combo box is used to change the instance of the DSP target to which code is to be downloaded.



The event log is used to display the history of the status of manual operations which have been performed.

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.

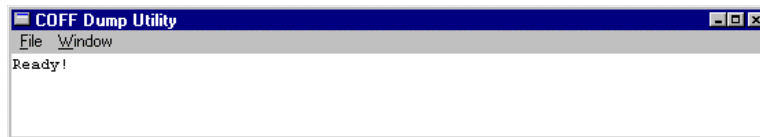


FIGURE 11. The COFF Dump Utility

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, Notepad 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 contains the following text:

```
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: 90135fh  
Initialized data starts at: 900800h  
Text section starts at: 900800h  
  
SECTION HEADER INFORMATION:  
BL_vec starts at 00900000h, length 0000040h, contains initialized data.  
.text starts at 00900800h, length 0000162fh, contains executable code.  
.data starts at 00900800h, length 0000000h, contains initialized data.  
.cinit starts at 00901e2fh, length 000001e9h, contains initialized data.  
.const starts at 00902018h, length 0000055bh, 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 12. 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 – Involves 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.

Creating a Binary File

When using single-board computers, such as the SBC32 or SBC6711, it is possible to embed application code into Flash ROM so that the application begins executing immediately after power-up. In order to accomplish this, you must first create a Code Composer project, build it, and then debug/test it by downloading it with UniTerminal and Code Composer.

The standard Code Composer build process will generate a `.out` file, which is suitable for use within UniTerminal or Code Composer, but is unsuitable for placement in Flash ROM. Once you have a viable `.out` file, it must be converted into a binary file before it can be burned into the target DSP card's Flash ROM. The process of converting an `.out` file into a binary file (`.bin`) is target specific.

SBC31, SBC32 and SBC54 targets

To briefly describe this conversion process, the `.out` file from the Code Composer project will be converted to a `.a0` file using a hex converter and a `.cmd` file. The `.a0` file is a readable ASCII version of the `.out` file. Then this intermediate `.a0` file is converted to a `.bin` file using a hex to bin converter (`hex2bin.exe`) provided in your release. To accomplish this conversion quickly and easily, Innovative Integration has provided a template (`application.mk`) that can be modified to convert your file. This template is located in the target DSP card's root directory.

To convert your `.out` file into a `.bin` file to be burned to the Flash ROM, proceed as follows:

1. To start off, you must have a viable `.out` file (ie. `yourfile.out`).
2. Next you will have to create a `.mk` file for the out to bin conversion, using the template provided `application.mk`.
 - Copy the template, `application.mk` file (in `C:\<I.I. Target Board>`) to the directory where `yourfile.out` resides and rename it to `yourfile.mk`.
 - Open this new file with a text editing program like Notepad.
3. Edit the new `.mk` file as follows:
 - Change the `OUTPUT` and `OUTPUT_BASE` to the desired output name for the bin file. For example, to convert `yourfile.out` to `yourfile.bin`, edit this template file in the following manner:

```
OUTPUT = yourfile.out

OUTPUT_BASE = yourfile.
```
 - You must also be sure that the `application.cmd` and the `hex2bin` files are found by the new `convertyourfile`. To do this:
 - a. Change the `HEX_ARGS` to point the where the `application.cmd` file is (ie if you have copied this file to the `..examples\target` directory, then you must change `HEX_ARGS = ..\..\application.cmd`).

b. Similarly, you must point to the location of the hex2bin file in the HEX2BIN variable.

- Then save the changes and close the editing program.

Note: It is always a good idea to remove or rename any old .bin file of the same name to ensure a new .bin file is created.

4. Open a DOS prompt and CD to the directory where the yourfile.out and the Convertyourfile.mk files reside. Type: **nmake -f yourfile.mk** and press <Enter>. This will create the yourfile.bin in the same directory.

SBC67 and SBC6711 Targets.

Run the supplied applet PromImage.exe. Browse to your application .out file then click the Make button to generate the image file. The auto-generated image file will have the same name as the input .out file, but will be named with the .bin extension. This file is suitable for embedding into sector one of the flash ROM on the target board.

Note: SBC67x must be equipped with version 1.2 or later of Talker in order to use these image files.

The Flash Burn Utility

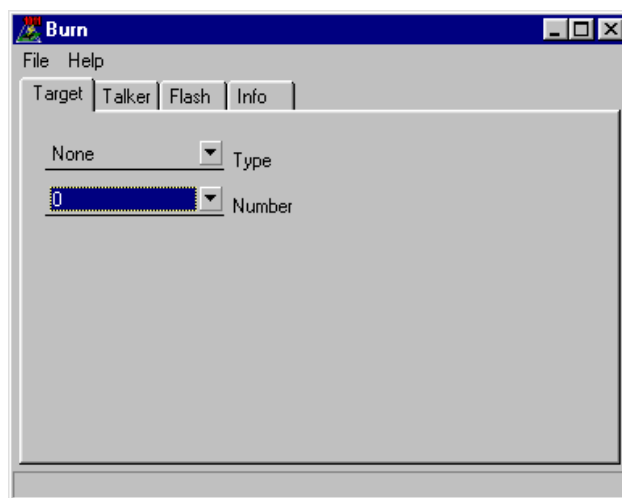
The Flash ROM programming utility (BURN.EXE) provides the capability to embed debugged application code within the flash ROM onboard Target DSPs which feature FLASH ROM. Currently, this includes each of the Innovative Integration single board computers (the SBC31, SBC32, SBC54, SBC62, SBC67, and SBC6711) products. Refer to the hardware manual for your DSP target to determine whether FLASH ROM is available on your system.

The utility supports both the 29F010 and 29F040 ROMs and provides special support for users with JTAG links between the host and the target in order to support programming.

The PROM utility contains only two menus, File and Help plus four tabbed page controls labelled Target, Talker, Flash, and Info. Currently, the Help menu is used to invoke help contents and the application About box, which indicates the revision of the application. The File menu provides access to the Options dialog which controls the timeout on JTAG loads and flash erase byte values (discussed later). In addition to the ability to convert Ascii hex text files into binary files, suitable for consumption by the Burn utility. The following is a brief description of the functions available under each of the primary tabs.

Target Page

The Target page contains two controls used to select the DSP type being used and its target number, as shown below.



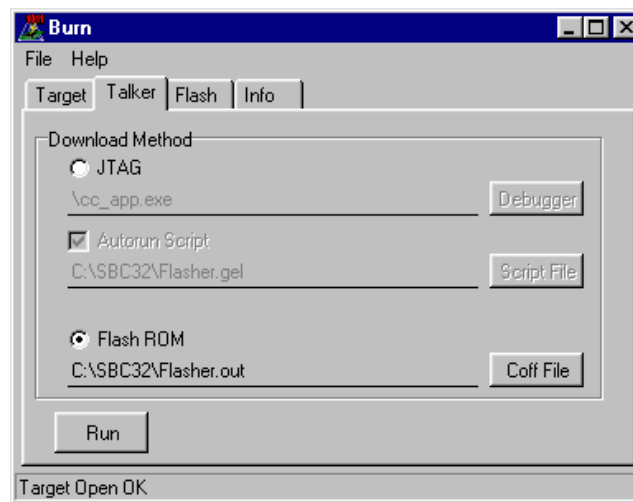
Click on the Number combo control and select the number of the DSP target you're using (usually 0). It is important to remember that for single-boards targets, zero corresponds to COM1 and target one corresponds to COM2, ect. Next, click on the Type combo control to display a list of supported targets, then select your DSP type.

After you have selected the DSP type, the applet will attempt to open the target device driver to establish communication with the target. The status of this attempt is displayed in the status bar at the bottom of the Burn applet window. If the driver is successfully opened (Target Open OK), click on the Talker tab to proceed. If the target driver does not successfully open, check the parameters being requested and the installation of your DSP. Also, close any other open applications (such as UNITERMINAL.EXE, which could have previously opened the DSP device driver. Due to Windows restrictions, only a single application may open a com port device at a time.

Talker Page

To support all single-board DSP targets similarly, BURN must download a simple DSP support executable to the target before an embedded application may be burned into the Flash ROM. This executable is `FLASHER.OUT`, located in the root directory of the Zuma Toolset. This small, pre-written executable provides all of the Flash I/O access functions necessary to allow the BURN utility to interact with the target Flash ROM.

The Talker page contains a number of window controls used to specify the mechanism to use to deliver the `FLASHER.OUT` support executable into the target DSP. Two methods are currently supported, JTAG and downloading via the existing Talker in Flash ROM.



You must use the JTAG download method whenever the target DSP Flash ROM does not contain a viable (bootable) image of the Talker program. In the default factory condition, the target contains a bootable Talker, which is used by application programs through the DLL and device driver to download executable programs to the target DSP. If this Talker image has been erased or corrupted, you must use the JTAG download method.

If the Talker is still viable in Flash, then you may use either of the download method. The Talker method is the preferred method, especially in circumstances where a JTAG debugger is unavailable, such as when performing in-the-field software updates.

JTAG Download. To download using JTAG, click the JTAG radio button and then click the Debugger button to browse to the location of your JTAG debugger software (usually `c:\com-poser\cc_app.exe` if you're using Code Hammer or `c:\ti\cc\bin\cc_app.exe` if you're using Code Composer Studio) and click <Open>. Next, click on the Script File button to browse to the location of the `Flasher.gel` file located in the root directory of the board-specific Zuma libraries (ie. `c:\SBC6711\Flasher.gel`) and click <Open>.

Under most circumstances, you should enable the Autorun Script check box as well. With this feature enabled, the specified Code Composer GEL script file is automatically executed as the debugger is started. A default script file has been provided and is located in the root directory of the Zuma Toolset installation. This GEL file automatically initializes the target DSP, downloads the `Flasher.out` support executable and launches it using the debugger Run Free command.

In rare circumstances, you may elect to disable the Autorun Script option. When this feature is disabled, BURN will automatically launch the debugger when you click the Run button, but you must manually load and run the `Flasher.out` executable. For example, this option is useful when using a debugger other than Code Hammer.

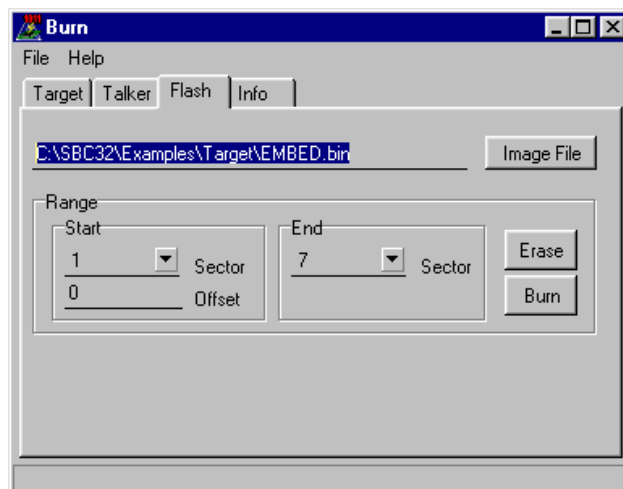
Flash ROM Download . To download using the Talker in flash ROM, click on the Flash ROM radio button and then click on the Coff File button to browse to the location of the `Flasher.out` file. This

file is usually located in the root of the board-specific Zuma libraries directory (ie `c:\SBC6711\Flasher.out`).

Downloading the Flash Support Code. Close Code Composer, if you had been using the debugger, click on Debug|Run Free before exiting. To download the flash support executable, `Flasher.out`, click on the Run button. The program will deliver the `Flasher.out` executable image into the Target using the selected method. When the support code has been download and is running, the status bar will display Flash Info read OK. If you have difficulty running the support executable, refer to the “Common Problems” section for troubleshooting suggestions.

Flash Page

When the support code has been downloaded and is running, the status bar will display “Flash Info read OK”. Next, you may burn binary images into the Flash ROM of the target DSP. Click on the Flash tab to change to the Flash Burn page (seen below).



Click on the Image File button to select the binary image file that is to be burned into the DSP flash. Note that BURN is incapable of burning DSP COFF executables (.OUT files) directly. You must use the other tools in the Zuma Toolset (specifically CodeWright, TIDeps and the TI HEX Conversion Utility) to create a viable binary image before attempting to burn an application into Flash ROM. Example embed-able projects for each II single-board target are available on the Innovative web site at www.innovative-dsp.com.

Controlling Region of Erasure. Adjust the Start Sector and End Sector combo boxes within the Start and End group boxes to control the location within the target flash in which the image is to be placed. Note that this controls only the range of sectors erased prior to attempting to burn the application into Flash.

The number of sectors actually needed is determined by the size of the binary image file. The BURN applet performs a range check to insure that your image will actually fit into the erased sectors. You must be sure to erase a sufficient number of sectors to contain your entire application image.

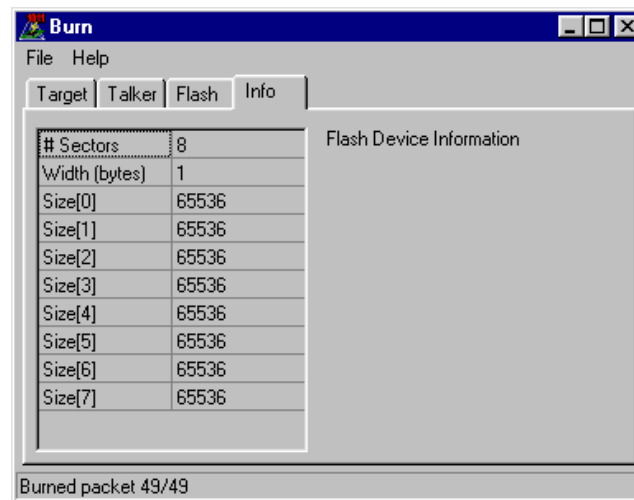
Re-burning the Talker Image. To burn or re-burn the factory Talker image, select 0 as both the starting and ending Sector. On all targets except C32-based targets, select 0 as the Offset. On C32 targets, the Offset should be set to 0x1000.

Warning: Avoid burning application code other than the factory Talker image into sector zero. On some targets, notably the SBC62, it may be impossible to initialize the JTAG debugger on a target board containing an invalid boot image. This situation can only be corrected by shipping the target DSP back to the factory for rework.

Burning a User-Written Application. To burn a user-written application, select 1 as the starting Sector and adjust the ending sector to the uppermost available sector, less any reserved sectors used as data storage within your DSP application program. The Offset field should be set to zero for all target board types.

Info Page

Detailed autoselect information read from the Flash ROM is available on this Info Tab. This information is utilized internally by the BURN applet. You need not enter information on this tab.



Example Burn Sequence

The following is a typical example of burning a binary file (.bin) into the target card's Flash ROM.

1. Run the Flash ROM programming utility (BURN.EXE) from the Start Menu/Programs/<Target Board>/Burn.
2. Click on the Target Tab.
 - Then click on the Number combo box and select the number of the DSP target you are using (usually 0).
 - Next, click on the Type combo box and select the target DSP type. (At the bottom of the display, "Target Open OK" should be displayed)
3. Click on the Talker Tab.
 - In this example, from the two download methods, the Flash ROM method is selected. Click on the <Coff File> button and browse to the root directory of the Zuma Toolset and find the Flasher.out file.
 - Then, click on the <Run> button. (At the bottom of the display, "Flash info read OK" should be displayed)
4. Click on the Flash Tab.
 - Then, click on the <Image File> button and browse to the binary file that is to be downloaded to the target DSP flash.
 - Next, adjust the Start Sector combo box to 1 and the End Sector combo box to 7.
 - Then, click on the <Burn> button to start the download. The downloading progress should be displayed. (At the bottom of the display, "Burn Completed" should be displayed)
5. Testing the burned .bin file.

SBC32:

- To run the application from the target DSP, turn the power off, disconnect the JTAG & serial port, and turn the power back on. Alternately with the serial cable attached, you may click on the Boot Embedded Application speed button on UniTerminal to launch the embedded application, rather than the embedded Talker.

SBC6x:

- To run the application from the target DSP, turn the power off, install jumper (JP24) and turn the power back on. Alternately with the serial cable attached and JP24 installed, you may click on the Reset Dsp speed button on UniTerminal to launch the embedded application, rather than the embedded Talker.

SBC6711:

- To run the application from the target DSP, turn the power off, install jumper (JP11) and turn the power back on. Alternately with the USB cable attached and JP11 installed, you may click on the Reset Dsp speed button on UniTerminal to launch the embedded application, rather than the embedded Talker.

Common Problems when Embedding Code

There are several problems frequently encountered when ROMing code. If you encounter one of the symptoms listed below, attempt the corrective action listed before calling Innovative technical support:

Symptom	Possible Corrective Actions
Inadvertently overwritten the Talker in sector 0.	Re-burn the Talker (per above).
Code Hammer cannot initialize the target.	<p>Verify that the I/O address specified in the Code Composer setup is correct and that the JTAG board is properly connected to the DSP board.</p> <p>Configure and run the Innovative JTAGDIAG.EXE utility to reset the debugger hardware. Code Composer Studio users should also run the XDS510 Reset Utility (using the same I/O address as entered into JTAGDIAG.EXE) to initialize the Studio Debugger.</p> <p>The target may be held in reset. Verify that the boards device driver is installed and operational.</p> <p>Code Composer versions prior to 4.01 are incapable of communicating with C67xx processors. Also, v4.01 is better able to recover from invalid images burned into sector 0 on SBC6711 DSPs. Contact I.I. for upgrade information.</p>

TABLE 3. Common Problems when Embedding Code in Flash ROM

Symptom	Possible Corrective Actions
<p>The embedded application will not boot.</p> <p>The application works properly from within Code Composer Studio but does not run (or is erratic) when embedded.</p>	<p>The target is being held in reset. For single-board targets, disconnect serial port #1 to the host and reboot. If serial port #1 must be connected, insure that the DTR output from the PC is de-asserted (since it controls target reset). Further, insure that the SBC's CTS serial input line is de-asserted since the Talker cannot launch a co-existing embedded application if CTS is asserted during cold-boot. On the SBC62, insure that the boot jumper JP24 is installed.</p> <p>You burned the application at an incorrect offset or starting at a sector other than 1.</p> <p>The application burn image was created improperly. Inspect the .A0 file and insure that the target image has an appropriate boot record. Be sure to convert the .A0 file into binary using the Options dialog for consumption by the Burn applet. On the SBC62 and SBC54 targets, be sure to locate the .const and cinit sections into the ROM so that they may be resurrected at boot time.</p> <p>The Target may still be held by the JTAG. Run the JTAG Diagnostic utility and then select the <Reset> button to release the target.</p> <p>The .bss section of your application is not automatically initialized to zero. Modify your linker command file to zero-fill the .bss section.</p> <p>Your embedded application may be too large. Use the Coff Dump utility to inspect the size of the .OUT file to verify that it can fit into the seven residual 16 kByte sectors in the AMD 29F010 (64 kByte sectors in the 'F040).</p>

TABLE 3. Common Problems when Embedding Code in Flash ROM

Introduction

The Innovative Integration (I.I.) Zuma Toolset allows users of I.I. 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 and code authoring environment as well as I.I.'s custom Windows applets (such as the `UNITERMINAL.EXE` terminal emulator).

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

On C6x platforms, such as Innovatives M6x, SBC6x, SBC6711, and Quatro6x, Code Composer Studio may be used for both code authoring and code debugging. Details of constructing projects for use on Innovative DSP platforms using Code Composer 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 I.I. 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 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. In the Innovative Intergration development system these stages are accomplished within the Code Composer program.

By using Code Composer Studio, these stages of the programming cycle are accomplished entirely within the integrated Studio environment. The project features of Code Composer Studio support component file editing and compilation stages, along with allowing the executable result to be downloaded and tested on the target hardware. This fully integrated programmers environment is more user-friendly than the basic command line interface, which comes standard with the TI tools.

A Simple Code Composer Studio Project

The following sequence illustrates the creation of a project to build the “Hello World!” program from within Code Composer Studio.

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

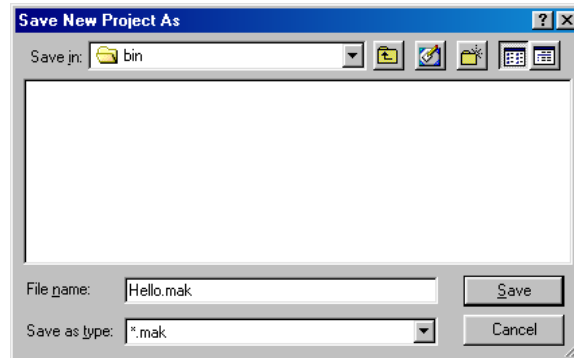


FIGURE 13. Creating a New Project in Code Composer Studio

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 may browse into the `<I.I. Target Board>\EXAMPLES\TARGET` directory.

Next, open the **Project** | **Add Files** to Project dialog box to add files to the project. Add the `HELLO.C` file from the `C:\<I.I. Target Board>\Examples\Target` directory and the `GENERIC.CMD` file from the `C:\<I.I. Target Board>` directory to the project. Remember to choose the correct file type then when you have selected the file to add, click **<Open>** button.

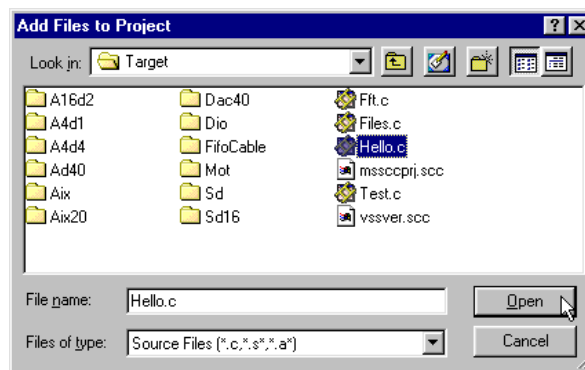


FIGURE 14. Adding Files to a Code Composer Studio Project

SBC6x Only: When building an application to be burned into the Flash ROM, the `EMBED.CMD` file (located `C:\<I.I. Target Board>\Examples\Target`) should be added to the project instead of `GENERIC.CMD` (refer to Chapter 7 “Building Flash Programmable Application”).

It is imperative that you add an appropriate command file to the Code Composer 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.

The library files will be required, but do not add them directly into the project like the `hello.c` and `generic.cmd` files. 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.

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

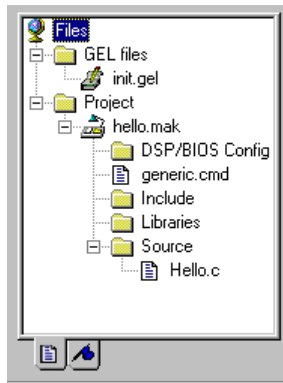


FIGURE 15. Code Composer Studio Project Window

Next, you must configure the project compiler settings so that when `Hello.c` is compiled, the appropriate memory model and switches are used.

Build Options (M62, Q62, SBC62 Boards)

Click on `Project | Build Options` to open the Build Options dialog box, then click on the Compiler tab to show the current compiler options. The required Build options are described in the following text, all other settings are optional and are described in the Code Composer Studio documentation.

Configure the compiler options to use the following settings: In the category column, choose Basic and click on the Generate Debug Info combination box and select “Full Symbolic Debug”. Then click on the Opt Level combination box and select “-O2: Function”. Next, in the category column, choose Advanced, click on the Endianness combination box and select “Big Endian”. Then click on the Mem-

ory Models combination box and select “Far Calls & Aggregate Data”. In the category column, choose Feedback, and click on the Banners combination box and select “No Banner”. In the category column, choose Files and in the Obj Directory box type “C:\M6x\Examples\Target”. Again in the category column, choose Preprocessor and in the Include Search Path box type C:\M6x\Include\Target”. When finished, the compiler dialog screen should look very similar like one below.

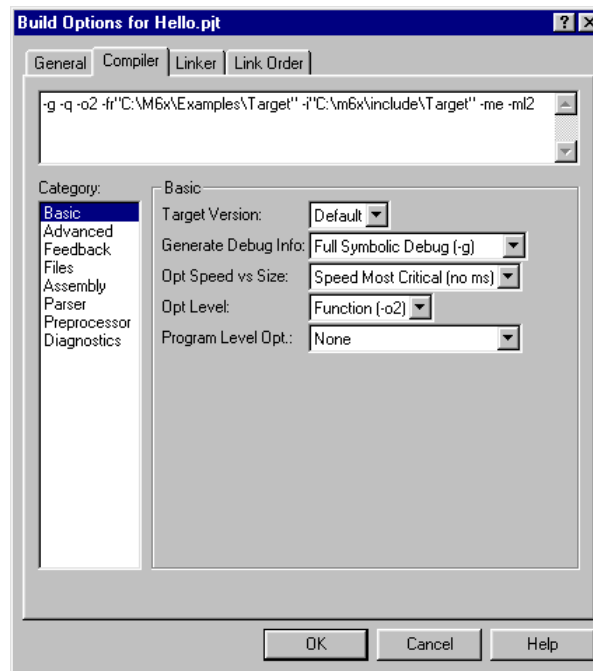


FIGURE 16. Code Composer Studio Compiler Build Options

Now, click on the Linker tab and configure the linker build options. In the category column, choose Basic. In this screen, make sure the Exhaustively Read Libraries has been selected. Then in the Output Filename box and Map Filename box type the project name. Click on the Autoint Model combination box and select “Run-time Autoinitialization”. Next, set the Heap Size to 0x400 bytes and the Stack Size to 0x800 bytes, the Heap & Stack size may change depending on your application’s needs. In the Library Search Path, type “.\\.\lib\Target. Now, add stdio.lib; periph.lib; dsp.lib; and rts6201e.lib into the Include Libraries edit box (in that order). Then on the Include Libraries edit box line, add either init.lib for non-DSB Bios applications or biosinit.lib for DSP Bios applications, between the periph.lib and dsp.lib entries. When finished, the linker dialog screen should look like one below.

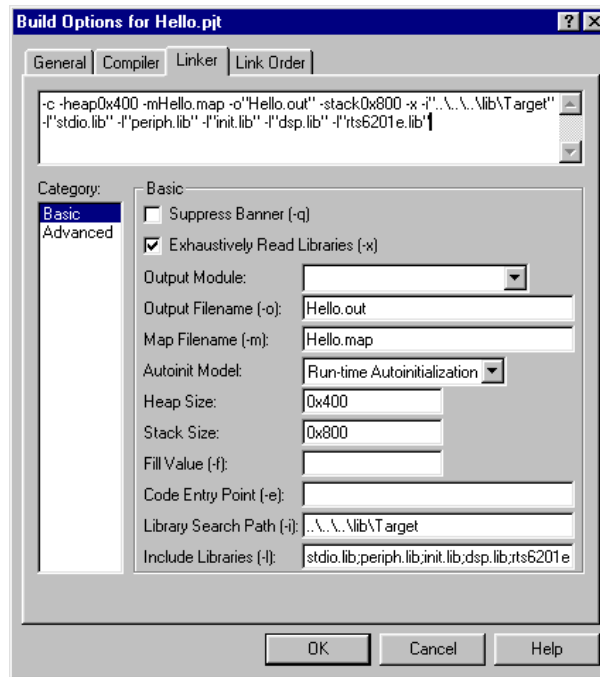


FIGURE 17. Code Composer Studio Linker Build Options

Once all the build options have been set, rebuild your project by clicking Project|Rebuild All in the Code Composer Studio menu bar. If errors are encountered in one or more source files, they are listed in the output window. You may visit and repair each error by either double clicking on each error in the Output window.

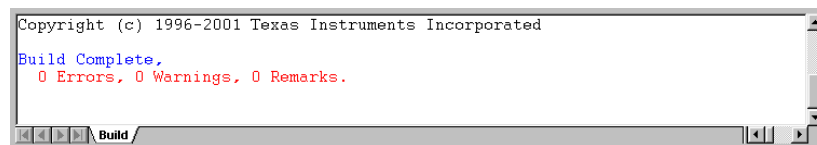


FIGURE 18. Code Composer Studio Build Results Window

Build Options (M67, Q67, SBC67 Boards)

Click on Project | Options to open the Build Options dialog box, then click on the Compiler tab to show the current compiler options. The required Build options are described in the following text, all other settings are optional and are described in the Code Composer Studio documentation.

Configure the compiler options to use the following settings: In the category column, choose Basic and click on the Target Version combination box, and select “670x”. Click on the Generate Debug Info combination box and select “Full Symbolic Debug”. Then click on the Opt Level combination box and select “-O2: Function”. Next, in the category column, choose Advanced, click on the Endianness combination box and select “Big Endian”. Then click on the Memory Models combination box and select “Far Calls & Aggregate Data”. In the category column, choose Feedback, and click on the Banners combination box and select “No Banner”. In the category column, choose Files and in the Obj Directory box type “C:\M6x\Examples\Target”. Again in the category column, choose Preprocessor and in the Include Search Path box type C:\M6x\Include\Target”. When finished, the compiler dialog screen should look very similar like one below.

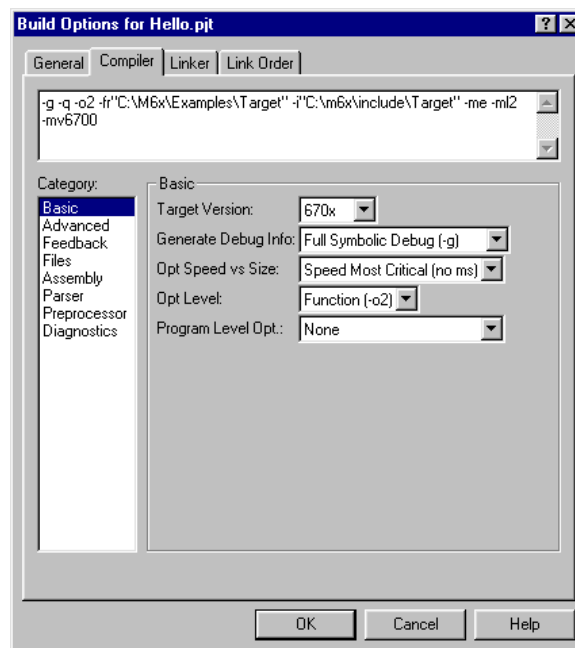
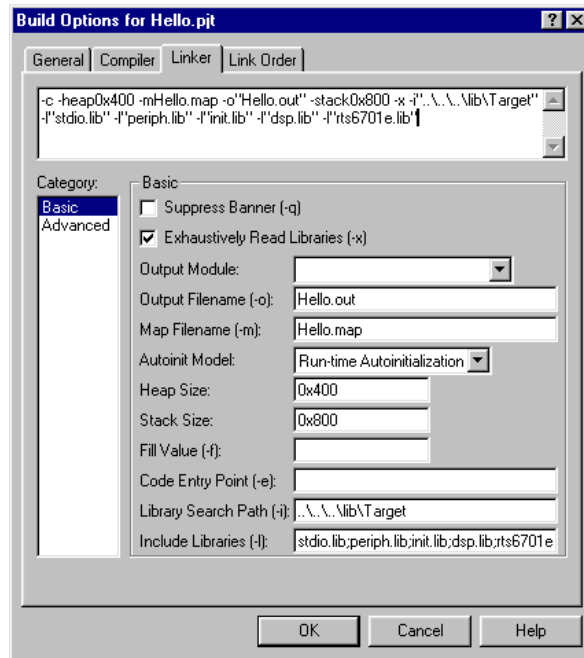


FIGURE 19. Code Composer Studio Compiler Build Options

Now, click on the Linker tab and configure the linker build options. In the category column, choose Basic. In this screen, make sure the Exhaustively Read Libraries has been selected. Then in the Output Filename box and Map Filename box type the project name. Click on the Autoint Model combination box and select “Run-time Autoinitialization”. Next, set the Heap Size to 0x400 bytes and the Stack Size to 0x800 bytes, the Heap & Stack size may change depending on your application’s needs. In the Library Search Path, type “..\..\lib\Target. Now, add stdio.lib; periph.lib; dsp.lib; and rts6201e.lib into the Include Libraries edit box (in that order). Then on the Include Libraries edit box line, add either init.lib for non-DSB Bios applications or biosinit.lib for DSP Bios applications, between the periph.lib and dsp.lib entries. When finished, the linker dialog screen should look like one below.



Once all the build options have been set, rebuild your project by clicking Project|Rebuild All in the Code Composer Studio menu bar. If errors are encountered in one or more source files, they are listed in the output window. You may visit and repair each error by either double clicking on each error in the Output window.

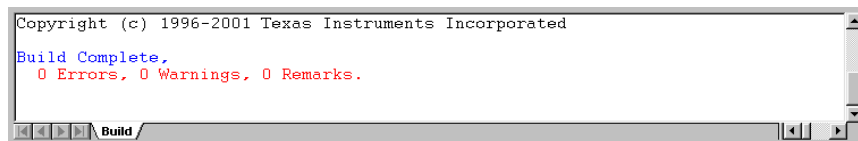


FIGURE 20. Code Composer Studio Build Results Window

Build Options (SBC6711 Boards)

Click on Project | Build Options to open the Build Options dialog box, then click on the Compiler tab to show the current compiler options. The required Build options are described in the following text, all other settings are optional and are described in the Code Composer Studio documentation.

Configure the compiler options to use the following settings: In the category column, choose Basic, click on the Target Version combination box, and select “671x”. Then click on the Opt Level combina-

tion box and select “-02: Function”. Next, in the category column, choose Advanced, click on the Endianness combination box and select “little Endian”. Then click on the Memory Models combination box and select “Far Calls & Aggregate Data”. Again in the category column, choose Feedback, and click on the Banners combination box and select “No Banner or File Names”. Under the Files category the Obj Directory can be added. When finished, the compiler dialog screen should look very similar like one below.

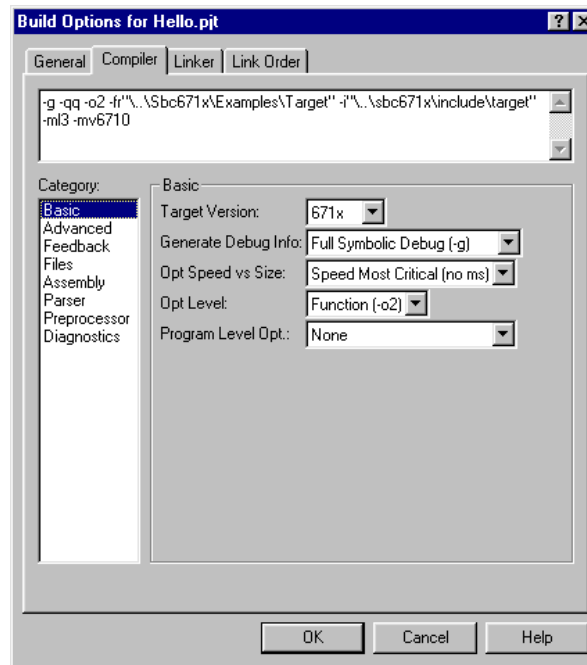


FIGURE 21. Code Composer Studio Compiler Build Options

Next, click on the Linker tab and configure the linker build options as follows. In the Output Module combination box select “Relocatable Executable”. Then, click on the Autoinit Model combination box and select “Run-time Autoinitialization”. Next, set the Heap Size to 0x400 bytes and the Stack Size to 0x800 bytes, the Heap & Stack size may change depending on your application’s needs. Make sure the Exhaustively Read Libraries has been selected. Now, add `periph.lib`, `init.lib`, `stdio.lib`, `dsp.lib`, and `rts6700.lib` into the Include Libraries edit box (in that order). Then on the Include Libraries edit box line, add either `init.lib` for non-DSB Bios applications or `biosinit.lib` for DSP Bios applications, between the `periph.lib` and `stdio.lib` entries. When finished, the linker dialog screen should look like one below.

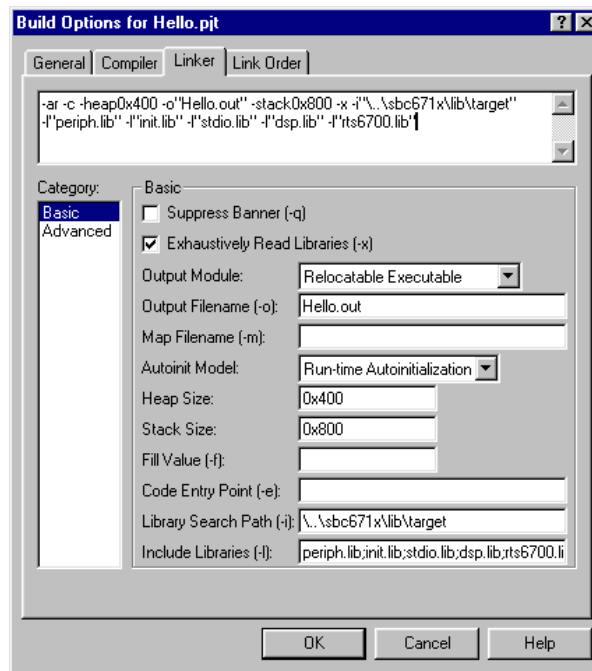


FIGURE 22. Code Composer Studio Linker Build Options for Non-Bios Project

Once all the build options have been set, rebuild your project by clicking Project|Rebuild All in the Code Composer Studio menu bar. If errors are encountered in one or more source files, they are listed in the output window. You may visit and repair each error by either double clicking on each error in the Output window.

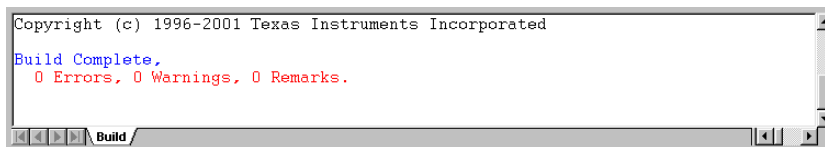


FIGURE 23. Code Composer Studio Build Results Window

Build Options (M44 Boards)

Click on Project | Options to open the Build Options dialog box, then click on the Compiler tab to show the current compiler options.

To configure the compiler options type the following command line in the compiler option box: “-gq -v40 -mn -o2 -x2 -frC:\M44\Examples\Target” When finished, the compiler dialog screen should look exactly like one below.

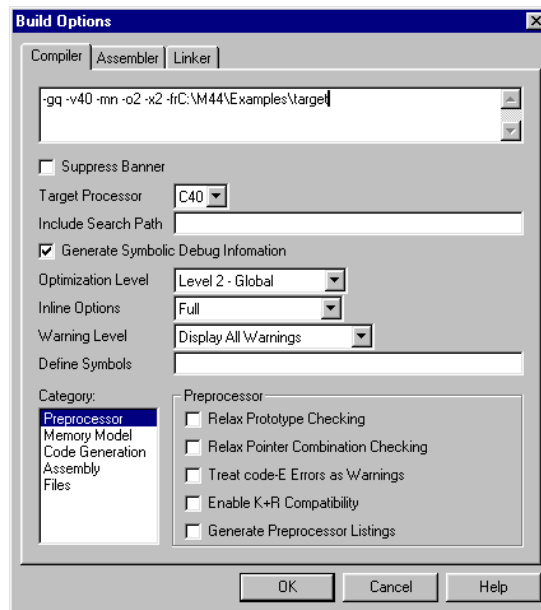


FIGURE 24. Code Composer Studio Compiler Build Options

Next, click on the Assembler tab and configure the assembler build options. In the assembler option box type the assembler build option “-r”. When finished, the assembler dialog screen should look like one below.

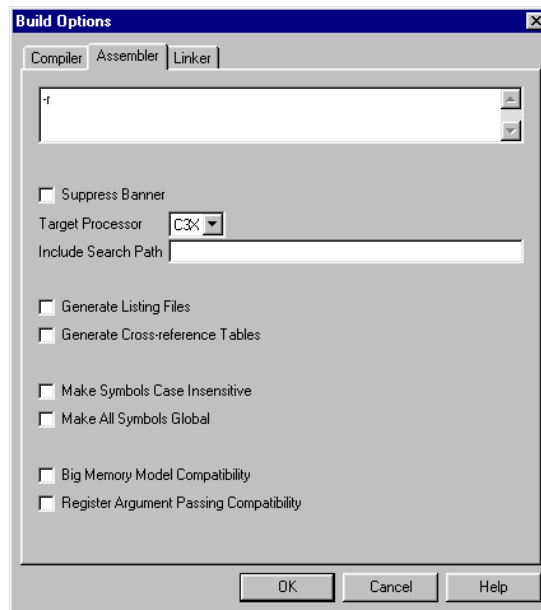


FIGURE 25. Code Composer Studio Assembler Build Options

Finally, click on the Linker tab and configure the linker build options as follows. In the Output Module combination box, select “Relocatable Executable”. In the Output Filename box, type in the name of the output file. Then in the Map Filename combination box, select “ROM Autoinitialization Mode”. Next, set the Heap Size to 0x400 bytes and the Stack Size to 0x200 bytes, the Heap & Stack size may change depending on your application’s needs. Make sure the Exhaustively Read Libraries have been selected. Now, add stdio.lib; periph.lib; dsp.lib; prts40.lib; and rts30.lib into the Include Libraries edit box (in that order). When finished, the linker dialog screen should look like one below.

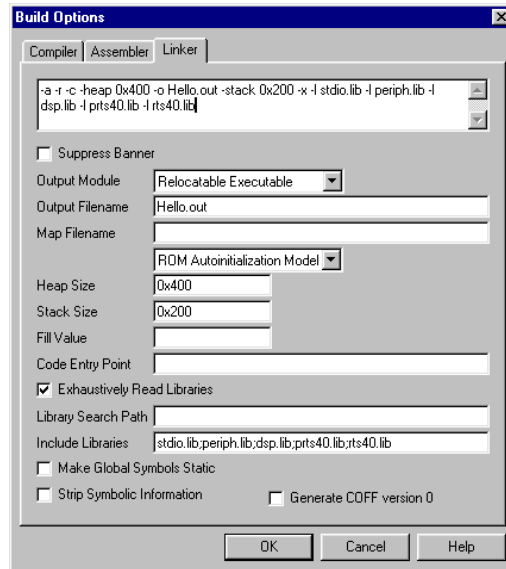


FIGURE 26. Code Composer Studio Linker Build Options

Once all the build options have been set, rebuild your project by clicking Project|Rebuild All in the Code Composer Studio menu bar. If errors are encountered in one or more source files, they are listed in the output window. You may visit and repair each error by either double clicking on each error in the Output window.

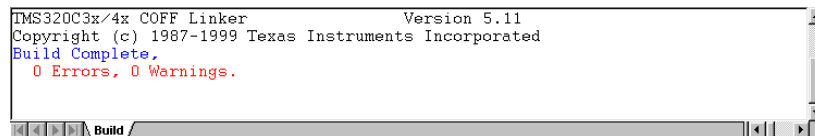


FIGURE 27. Code Composer Studio Build Results Window

Build Options (SBC32 Boards)

Click on Project | Options to open the Build Options dialog box, then click on the Compiler tab to show the current compiler options.

To configure the compiler options type the following command line in the compiler option box: “-gq -v32 -o2 -x2 -frC:\SBC32\Examples\Target” When finished, the compiler dialog screen should look exactly like one below.

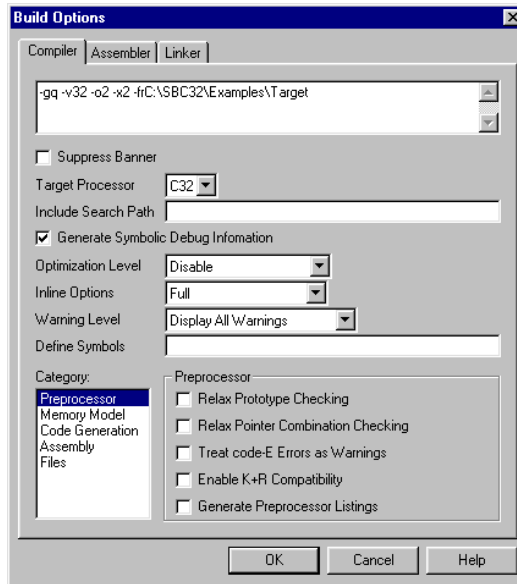


FIGURE 28. Code Composer Studio Compiler Build Options

Next, click on the Assembler tab and configure the assembler build options. In the Target Processor combination box, select “C3X”. Then type the last assembler build option “-r” in the assembler option box. When finished, the assembler dialog screen should look like one below.

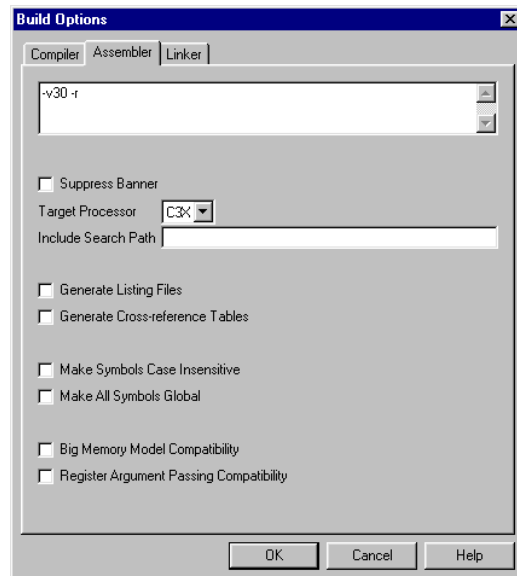


FIGURE 29. Code Composer Studio Assembler Build Options

Finally, click on the Linker tab and configure the linker build options as follows. In the Output Module combination box select “Relocatable Executable”. In the Output Filename box, type in the name of the output file. Then in the Map Filename combination box, select “ROM Autoinitialization Mode”. Next, set the Heap Size to 0x400 bytes and the Stack Size to 0x200 bytes, the Heap & Stack size may change depending on your application’s needs. Make sure the Exhaustively Read Libraries have been selected. Now, add stdio.lib; periph.lib; dsp.lib; and rts30.lib into the Include Libraries edit box (in that order). When finished, the linker dialog screen should look like one below.

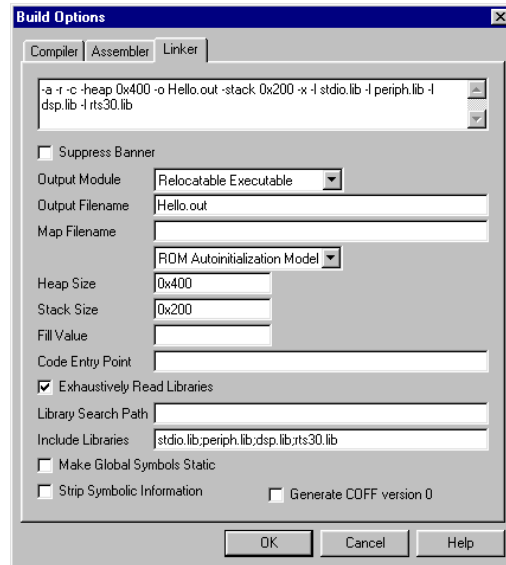


FIGURE 30. Code Composer Studio Linker Build Options

Once all the build options have been set, rebuild your project by clicking Project|Rebuild All in the Code Composer Studio menu bar. If errors are encountered in one or more source files, they are listed in the output window. You may visit and repair each error by either double clicking on each error in the Output window.

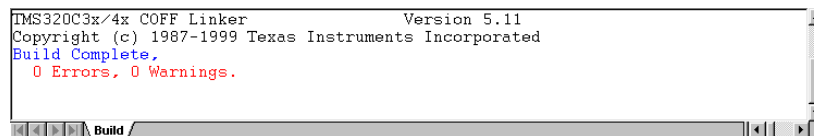


FIGURE 31. Code Composer Studio Build Results Window

Automatic projectfile creation

When a project is created, opened, modified, built or rebuilt, the Code Composer Studio dependency generator automatically generates a project makefile (named `<project file>.pjtc`, 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 Code Composer 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 | Rebuild All` from the Code Composer 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 to echo any keystrokes and exiting. Bring up the "Hello, World" source file edit screen. Scroll down the source file by using cursor down button 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 Code Composer 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`). Code Composer 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 board. At this point, the program may be run using the terminal emulator applet, which may be invoked using the UniTerminal shortcut located within the target board program group created during the Zuma Libraries installation process. In the terminal emulator, download the `hello.out` file. The program runs and outputs the message "Hello, Brave New World" to the terminal emulator window.

If errors are encountered in the process, Code Composer Studio detects them and places them in the build output window. If the error occurred in the compiler or assembler (such as a C syntax error), the cursor may be moved to the offending line by simply double-clicking on the error line within the build output window, and the error message will be displayed in the Code Composer Studio status bar. If the linker returns a build error, the build 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 build output window.

Note: Be sure to start the terminal emulator **BEFORE** starting Code Composer, to avoid resetting the DSP target in the midst of the debugging session. If the terminal emulator 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 Code Composer Studio application.
3. Invoke the terminal emulator “UniTerminal” application.
4. Restart the Code Composer Studio application.

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

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 I.I. C Development Environment, 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 M6x version of the `hello` program example. 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
 *      Test file/program for target board.
 */

#include "periph.h"
#include "stdio.h"

main()
{
    int key;
    enable_monitor();
    clrscr ();

    printf("Hello World!\n");
    printf("\nEchoing keystrokes...\n");
}
```

```

do
    {
        key = getchar();
        putchar(key);
    }
while(key != ESC);

monitor();
}

```

The two lines of the program that begin with a “#” 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 `enable_monitor()` function will setup the standard monitor I/O interface and reset the terminal window. The next two lines perform basic standard I/O functions; clearing the terminal emulation screen and printing “Hello World!” & “Echoing keystrokes...”. These two lines are where custom code should be inserted.

The following `getchar()/putchar()` sequence simply echoes keys typed at the terminal emulator back to the terminal display, until the Esc key is pressed. When Esc is pressed, the `monitor()` function effectively terminates the program, except that interrupts are still active and interrupt handlers (if they had been installed) would still execute properly.

The `hello` program is very simple, but it contains the basic components of a typical DSP application, as well as the initialization needed to interact with the terminal emulator.

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 resides in, as well as the header file, which that be included for the function.

Compiling/Assembling/Linking Outside Code Composer Studio

Under certain circumstances, it may not be possible to use Code Composer Studio macro definitions to compile inside the editor. `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 the above cases, if any errors occur, an error file (`mycode.err`) is generated by the software tools. The `mycode.err` file contains the full console output of each of the tools. Any error that is generated by the tools will be recorded in this file.

The Next Step: Developing Custom Code

In building custom code for an application, Innovative Intergration recommends that you begin with one of the sample programs as an example and extend 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. It is recommended that you familiarize yourself with the sample programs provided. The sample programs will provide a 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.

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 Innovative Integration's DSP co-processor and data acquisition cards. Virtual device drivers (Windows 9x VxD or NT/2K Kernel Mode Driver) and dynamic link libraries (DLL) are included to provide an easy-to-use, portable low-level interface for the target hardware. Sample applications show how to call the DLL functionality and present basic interface examples with guidelines for on processor card control requirements and data movement.

Host software development is directly supported under either the Borland C++ Builder environment or the Microsoft MSVC environment for generating 32-bit Windows applications. Example application programs included in the development package are supplied with Borland project files, making program modification and regeneration as simple as possible. These programs utilize DspComponent, a VCL C++ component class provided in the Development package, to gain access to DLL features. Examples with similar functionality, utilizing an ActiveX wrapper derived from DspComponent are available for Microsoft Visual C++ users. See the online DspComponent.hlp file for a full description of the DspComponent.

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 interactions takes place through calls to the supplied target DSP 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 busmaster memory locking and access control.

The function calls available under the DLL are documented in the Host32.hlp Windows help file, provided in the Zuma Toolset. 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 sixteen target DSP boards simultaneously by default (contact II if more 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;        // Target 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;

```

DspComponent

In the last few years, several different Rapid Application Development (RAD) environments have been developed to improve programmer productivity. Microsoft first produced Visual Basic, and Borland followed with Delphi (Object Pascal) and most recently, C++ Builder (which uses C++). What these tools have in common is the concept of a "component", a packaged piece of software that encapsulates the behavior of a button or other Windows control.

These can be arranged on forms with a visual editor provides a way to "drop" components into a window and add code to process events generated by components. For example, a button component generates a "Button Press" event, which can be set to automatically call a handler function by the RAD tool. The handler code will be called whenever the button is pressed.

DspComponent is a non-visual component provided by Innovative Integration, which provides a component interface to the board DLL that is included in each Zuma tool set. This allows the application programmer to avoid learning the details of connecting to the DLL, in favor of a drop-in package that is integrated into the language. This allows the developer to concentrate on the user interface and the DSP target programming, and not on the mechanics of Windows DLLs and drivers.

The DSPComponent Package consists of two components, an ActiveX and a Borland C++ Builder VCL component. Both perform the same function, and have essentially the same feature set, so that unless indicated a reference to the TIIDspComp component for C++ Builder applies also for the IIBoardX ActiveX control. The control was ported to ActiveX to allow its use in MSVC and Delphi.

The TIIDspComp component provides a high level interface to the Innovative Integration board DLL included as part of the Zuma tool set for a particular target board. This interface is common across all targets, and the same control can be used for all Innovative boards by picking the Target type at design time or even at run time. Mailbox access to the target is fully supported to the limits of the board selected, as simply as reading or writing to an array in the program. Interrupts in both directions, from host to target and target to host are supported. An interrupt from the target, if enabled, triggers a "software event" that can be handled just as any other event, such as a button click can. Resetting the board and downloading COFF files is simplified into a single call that returns a status code for failure conditions.

Features:

- Drop in VCL Component for C++ Builder and Active X version for Visual C++ and Delphi.
- A BoardType property selectable at design time or run time allowing one component to handle any Innovative target board.
- *Enabled* property to open and close target DLL
- Simplified board Booting and COFF downloading methods.

- Interrupt support for both Target to Host and Host to Target interrupts.
- Target to Host interrupt handler by use of standard Event handler mechanism.
- Simplified mailbox property array (Mailbox[]) to access on-board mailboxes.
- Mailbox flag arrays to check for incoming mailbox data or free outgoing mailboxes without waiting for data.
- Shared memory properties to give access to board shared memory for high-speed data transfer to dualport or busmaster memory.
- Windows NT and Windows 95 support using Innovative's Zuma DLLs.

See the DspComponent subdirectory on your hard disk for detailed online help for DspComponent (Windows help: DspComponent.hlp) and for examples illustrating use of the component for streamlined Host application development.

Host Example Programs for the SBC671x Baseboard

Overview

The SBC671x Baseboard differs from other baseboards in that the normal means of communicating with the host is at a higher level than other cards. On other single-boards, the serial interface lends itself to low-level, character based communications well. The Dsp cards support some means of bulk data transfer and a separate low-level signalling and command interface (For example, Busmastering for bulk transfer and mailboxes for signalling and commands on the M62). For compatibility with other SBCs, the SBC671x also implements a simulation of mailboxes for transferring characters and low level data.

Message Packets. On this baseboard, using USB, the most natural way to transport information is by means of asynchronous packets of data that are transferred and decoded by the destination. These messages may be of varying sizes, allowing large messages to be efficient in sending bulk data while allowing small command messages to be mixed into the message stream. When delivered to the destination, the messages can be parsed and can result in any kind of processing desired.

By having the receipt of a message trigger the generation of additional messages, a message protocol can be developed to allow the transfer of data or the execution of control functions on demand from the other side of the link. This is even more natural if the applications are written in an event-driven style. The arrival of messages are the events to which the application responds.

Commands. The SBC671x also supports a second, more limited method of communication called *commands*. These are implemented via the USB VendorCommand mechanism. The host side can send a 32-bit word with a Write command, or send a 32-bit word and receive a 32-bit word reply with a Read command synchronously from the target. The requirement that the host initiate the transfer limits the

use of commands, but the fact that the host will wait for the response before proceeding makes commands useful for synchronizing the target and host applications.

Interrupts. As with all baseboards, the SBC671x supports target-to-host and host-to-target interrupts.

On other baseboards, the standard host example was named “Scope” and demonstrated data transfer via the bulk interface signalled by interrupts. Because of the differences in the interface, the example programs for the SBC671x demonstrate the use of the forms of communication between the target and host and give an example of how to construct a command protocol between two applications, the target application and the host application. The two applications are very similar, the major difference being the platform used for development.

MessageTest

The MessageTest.exe example resides in the Examples\HostMessage directory. The target application matched with it is located in the Dsp subdirectory. The two applications share a header file, `Messages.h`, that contains the ID codes for the commands that are passed between the target and host. The IDs are entries in an enumeration, which assures that the IDs will be unique in each application. By using a single source file, we avoid the problem of two lists of IDs losing sync and producing difficult-to-detect errors during development.

The application uses the VCL DspComponent to provide the component interface to the application.

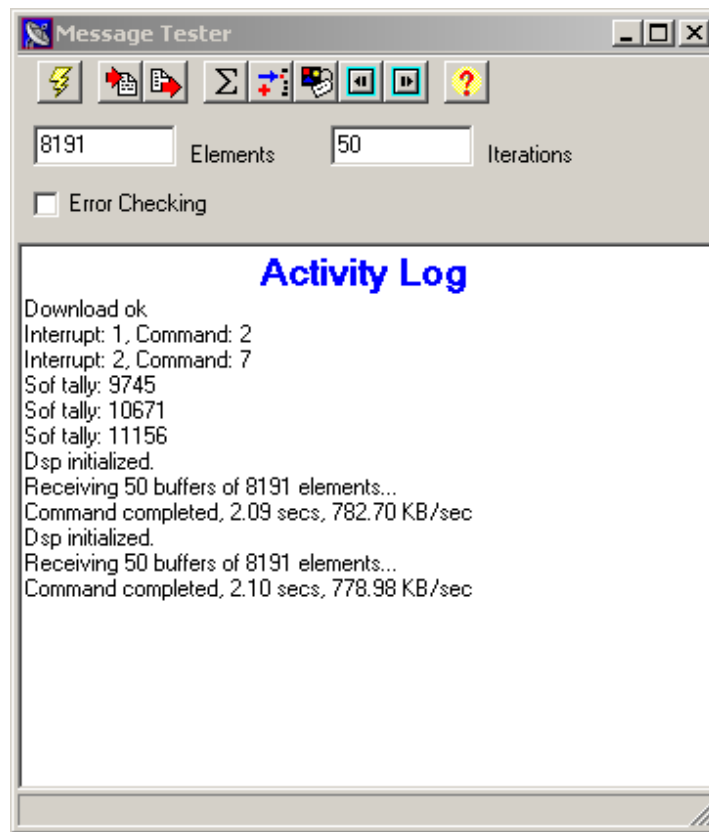


FIGURE 32. The MessageTest Example

Most of the area of the window is a visual log of the results of the functions performed when the buttons across the top of the screen are pressed. The remaining controls provide arguments to some of the other functions below. The code for this window can be found in `MessageTestMain.cpp`.

Download. The leftmost button is the only one active when the application begins. When pressed, it brings up the browser window, allowing the user to select which `.OUT` file will be downloaded to the target. There are two versions of `MsgTest.OUT` available, residing in the `Examples\Host\Message\Dsp` and `Examples\Host\Message\Bios` directories. Both have identical functionality, however one uses DSP/BIOS and the other does not. The browser defaults to the non-BIOS version. Once the file is selected, `TForm1::DownloadButtonClick()` is called. This function will attempt to prepare the target processor and download the target application onto it. It first attempts to load the baseboard DLL, printing a specific message if the DLL cannot be found. If the DLL is present, the DSP is booted and the target code is downloaded and executed. If the program downloads without error, the other buttons are enabled, allowing the communication tests to be run.

Message Receive Test. The second button performs a message receive rate test. (See `TForm1::RcvButtonClick()`). A single message to the target is used to trigger the sending of *Iterations* messages back, each containing data of size *Elements* in words in addition to the header information. These parameters are

entered from the edit controls on the main window. In addition, if the ErrorChecking box is checked, known data is inserted in the packets and checked for accuracy when it arrives at the host.

Messages are sent by calling the DspComponent function `XmtMessage()`. It takes a command code, and a pointer to the user-defined parameter region and its size in words. Note that since the message is asynchronously sent off, the host application cannot wait for the reply to arrive here. This portion of the processing must end with the posting of a request for action by the target application. When the target sends the reply messages, they will be processed by the message receiver code.

For this code, view the function `TForm1::DspRcvMessage()`. This function is an event handler for the Dsp Component's `OnRcvMessage` event. Whenever a message is received from the host, this handler is called to allow applications to process the message. The command ID of the message is used to select the processing function for the message. In this case, the `TForm1::HandleTargetDataMessage()` function is called to check the packet for errors, count the bytes transferred, and when all the messages have arrived, add to the log display the data rate for incoming data.

Message Transmission Test. The third button performs a similar test for message transmission. When the button is pressed, the `TForm1::XmtButtonClick()` function sends *Iterations* messages to the target, each containing data of size *Elements* in words in addition to the header information. If the ErrorChecking box is checked, known data is inserted in the packets for checking by the target on arrival.

SOF Counter Commands. The next two buttons demonstrate the use of commands. `TForm1::ReadSofButtonClick()` uses a Read Command to obtain a counter from the target that counts the number of SOF packets detected on the USB. These packets are sent about 1000 times a second. `TForm1::InitDspButtonClick()` uses a Write Command to clear the SOF counter to 0.

Revision Code Message. The next button uses a command message and a reply message to create a simple protocol that in this case retrieves a revision code from the target. `TForm1::RevisionButtonClick()` sends a message that requests the revision code. The target program creates a reply message, which is processed by `TForm1::HandleRevisionReply()`. This processing function adds the revision code to the log. The non-BIOS example will report "Version: 1.01", while the DSP/BIOS example will return "Version: 1.02".

Target to Host Interrupts. The next button calls `TForm1::HostInterruptButtonClick()`, which sends a message to the target that requests an interrupt to the host. This interrupt results in the `OnInterrupt` event on the DspComponent. `TForm1::DspInterrupt()` is the handler for this event and prints a message in the log.

Host to Target Interrupts. The final test button demonstrates Host to Target interrupts. In `TForm1::TargetInterruptButtonClick()`, first the interrupt is generated by calling the Dsp Component `Interrupt()` method. After this, a message is posted to the target requesting the count of interrupts on the target. When this message arrives, `TForm1::HandleMailboxTallyReply()` posts the count to the log display.

The Target Application

The target application is involved in all of the transactions of the Message Test application. Each command or message must be processed and a reply might possibly be returned if required by the protocol. In this section we will run through how the target can perform these actions. See the file `MsgTest.c` to follow along with the description.

The main() Routine. The main routine is relatively simple, as most of the work involved in communication with the host is performed by the Zuma library. The interface to the library code is a number of callback function pointers that will be called when processing is required. These act very much like events do on the host side application. These handlers are:

Event Variable	Handler Function	Purpose
OnWriteCommand	WriteCommandHandler()	Process Write Commands from Host
OnReadCommand	ReadCommandHandler()	Process Read Commands from Host
OnProcessMessage	ProcessMessageHandler()	Process Messages from Host

After installing these handlers, the host communications are started with a call to `init_communications()`. Next, a handler for mailbox interrupts from the host is installed to process interrupts from the host. Finally, the main function remains in a loop calling the library function `ProcessRcvMessage()`. This function needs to be called to allow messages to be processed. It should be called in any long loop or idle polling process to process messages that may arrive in a timely manner.

ReadCommandHandler(). This handler is installed to process Read Commands from the host. The system passes in the command ID, along with other parameters. The data to be returned is passed back by the data argument. For example, the Get SOF command has the ID of `ccGetSofCnt`. The switch statement loads the data argument with `get_sof_tally()`'s return value. This sends the SOF count back as the response.

WriteCommandHandler(). This handler processes Write Commands from the host. Here the data argument is a parameter sent by the host. As an example, the `ccInit` command resets system parameters and the SOF counter.

ProcessMessageHandler(). This handler processes messages from the host. In a simple case, the `ccGetRevision` ID is sent to obtain the revision ID. The processing for this message is a simple call to `XmtMessage()`, which transmits the message to the host with the revision code inserted as part of the message. In a more complex version, the `ccGenerate` command results in the posting of a number of messages to the host to perform the receive message test. Each message coming from the host results in a call to a processing function.

Host to Target Interrupts. When an interrupt is sent by the host, the handler installed for mailbox interrupts is called. `MailboxInterruptHandler()` acknowledges the interrupt and increments a counter. This counter is retrieved by the host by a message request in order to verify the delivery of the interrupt.

Target to Host Interrupts. One message sent by the host, `ccHostXrpt`, generates an interrupt to the target by calling the library function `mailbox_interrupt()`. The parameter is delivered through the mailbox to the host for logging.

VcMessage Test

The second example is implemented using Microsoft Visual C++. It also demonstrates the means of communication between the target and host applications. It uses the same target application as the VCL example, and uses the same header file, `Messages.h`, to assure that the commands IDs remain consistent in the two applications.

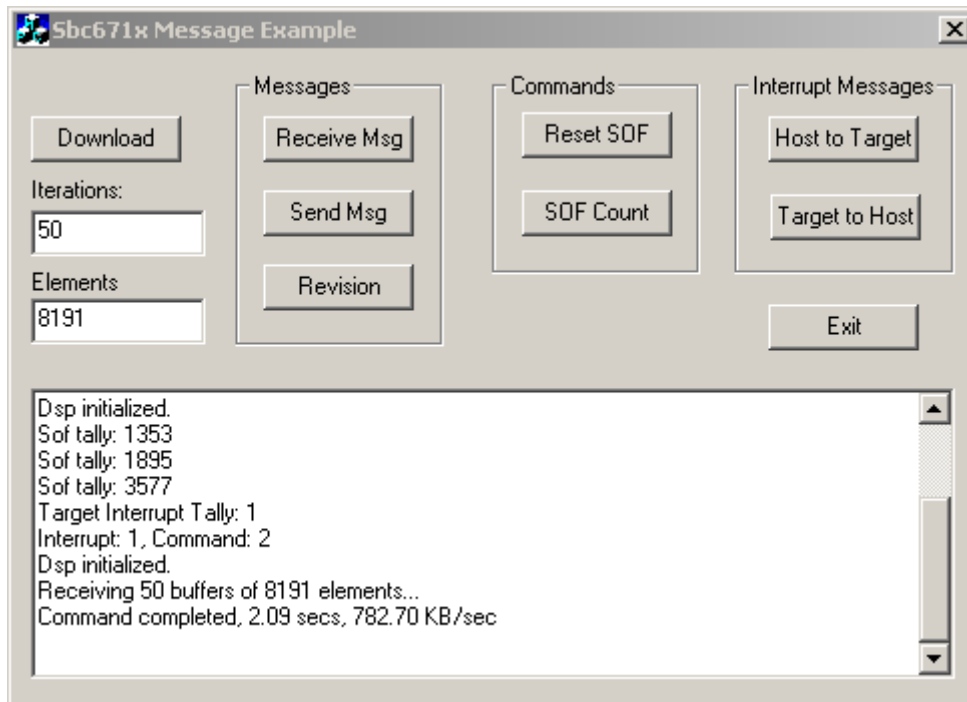


FIGURE 33. The VclMessage Example

To provide an object-oriented interface to the target, the VclMessage example uses the Dsp Component ActiveX control. This component allows the use of similar methods, properties, and events as the VCL component.

At the bottom of the window is a visual log of the results of the functions performed when the buttons in the top half of the window are pressed. The remaining controls provide arguments to some of the other functions.

The code for this application is located in Examples\Host\VcMessage directory. It is a dialog box based MFC application. The code for the window controls can be found in VcMessageDlg.cpp.

Download. The Download button is the only one active when the application begins. When pressed, it brings up the browser window, allowing the user to select which .OUT file will be downloaded to the target. There are two versions of MsgTest.OUT available, residing in the Examples\Host\VcMessage\Dsp and Examples\Host\VcMessage\Bios directories. Both have identical functionality, however one uses DSP/BIOS and the other does not. The browser defaults to the non-BIOS version. Once the file is selected, CVcMessageDlg::OnDownload() is called. This function will attempt to prepare the target processor and download the application. It first attempts to load the baseboard DLL, printing a specific message if the DLL can not be found. If the DLL is present, the DSP is booted and the target code is downloaded and executed. If the program downloads without error, the other buttons are enabled, allowing the communication tests to be run.

Message Receive Test. The first group of buttons collects tests involving Messages. The first button in this group performs a message receive rate test. (See `CVcMessageDlg::OnRcvMessage()`). A single message to the target is used to trigger the sending of *Iterations* messages back, each containing data of size *Elements* in words in addition to the header information. These parameters are entered from the edit controls on the main window.

Messages are sent by calling the `DspComponent` function `XmtMessage()`. It takes a command code, and a pointer to the user-defined parameter region and its size in words. Note that since the message is asynchronously sent off, the host application cannot wait for the reply to arrive here. This portion of the processing must end with the posting of a request for action by the target application. When the target sends the reply messages, they will be processed by the message receiver code.

For this code, view the function `CVcMessageDlg::OnRcvMessageIboardx1()`. This function is an event handler for the `Dsp Component's OnRcvMessage` event. Whenever a message is received from the host, this handler is called to allow applications to process the message. The command ID of the message is used to select the processing function for the message. In this case, the `CVcMessageDlg::HandleTargetDataMessage()` function is called to count the bytes transferred, and when all the messages have arrived, add to the log display the data rate for incoming data.

Message Transmission Test. The `SendMsg` button performs a similar test for message transmission. When the button is pressed, the `CVcMessageDlg::OnSendMsg()` function sends *Iterations* messages to the target, each containing data of size *Elements* in words in addition to the header information.

Revision Code Message. The `Revision` button uses a command message an reply message to create a simple protocol that in this case retrieves a revision code from the target. `CVcMessageDlg::OnGetRevision()` sends a message that requests the revision code. The target program creates a reply message, which is processed by `CVcMessageDlg::HandleRevisionReply()`. This processing function adds the revision code to the log. The non-BIOS example will report "Version: 1.01", while the DSP/BIOS example will return "Version: 1.02".

SOF Counter Commands. The central group contains two buttons that demonstrate the use of commands. `CVcMessageDlg::OnGetSof()` uses a `Read Command` to obtain a counter from the target that counts the number of SOF packets detected on the USB. These packets are sent about 1000 times a second. `CVcMessageDlg::OnResetSof` uses a `Write Command` to clear the SOF counter to 0.

Target to Host Interrupts. The right group of buttons contains buttons that test interrupts. The `Target to Host` button calls `CVcMessageDlg::OnTargettoHost()`, which sends a message to the target that requests an interrupt to the host. This interrupt results in the `OnInterrupt` event on the `DspComponent`. `CVcMessageDlg::OnDspInterruptIboardx1()` is the handler for this event and prints a message in the log.

Host to Target Interrupts. The final test button demonstrates Host to Target interrupts. In `CVcMessageDlg::OnHostToTarget()`, first the interrupt is generated by calling the `Dsp Component Interrupt()` method. After this, a message is posted to the target requesting the count of interrupts on the target. When this message arrives, `CVcMessageDlg::HandleMailboxTallyReply()` posts the count to the log display.

Software is created for the target DSP by using one or more of the tools included in the Developer's Package, either alone or in concert with each other. These tools are used 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 Manual* details the use of the individual tools in the package to create executables for the target. In addition, give step-by-step instructions on how to use the C compiler and Code Composer Studio 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 SBC6x 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 TMS320C6x 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 the *TMS320C6x C User Guide*, Runtime-Support Functions chapter for a complete list of TI C functions.

The Innovative Intergration SBC6x Developer's System also includes extensive high-level libraries useful in interacting with the various peripherals on the SBC6x boards. The following sections describe by peripheral type the functions provided in the peripheral library.

SBC6x Zuma Toolset Libraries

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

The peripheral libraries for the SBC6x 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
<code>\DSP</code>	Standard Digital Signal Processing Routines.
<code>\PERIPH\ANALOG</code>	Drivers for the SBC6x A4D4 instrumentation-grade analog I/O module and the complementary TERM mux module. Drivers for the SD and SD16 high-performance audio modules. Drivers for the AIX, AIX20 and A4D1 high-speed modules.
<code>\PERIPHERAL\BUS</code>	Drivers for the TI 16C750 UART.
<code>\PERIPH\DIGITAL</code>	Digital I/O, 8254 Timer control, module FLASH ROMs, etc. Drivers for DIO module. Drivers for MOT motion control module.
<code>\PERIPHERAL\FLASH</code>	AMD 29F0x0 flash ROM programming drivers.
<code>\PERIPH\MISC</code>	Miscellaneous processor control and data conversion functions.
<code>\PERIPH\RTS</code>	Modified boot-up routines for the SBC6x baseboard.
<code>\PERIPH\BUS</code>	Universal serial bus device code.
<code>\STDIO</code>	Console and terminal emulation functions.
<code>\TALKER</code>	Start-up umbilical 'C6201 software.

TABLE 4. Zuma Toolset Source Directories

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

Directory	Library Source
\EXAMPLES\HOST	Example host PC programs illustrating use of the DLL to control the DSP board from within Borland Builder (C++) programs.
\EXAMPLES\TARGET	Example target DSP 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 C++ and Visual Basic programs.
\LIB\TARGET	Linkable library files for target TI C and assembler programs.
\Source	Useful public domain source files for the C6201 processor.

TABLE 5. Zuma Toolset Support Subdirectories

STDIO Console Terminal Driver. The Developer's Package contains a full-featured terminal emulator application (uniterminal.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 SBC6x 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 directory contains code to support high-level access to the 16750 UART driver, which implements interrupt-driven input and output to the device configured as an asynchronous serial port. Source code for the functions is given in the \PERIPH\BUS directory. Strategic communications support functions have been coded as inline functions. Source for these functions is located in \INCLUDE\TARGET\BUS.

FLASH ROM Programming Drivers. The flash ROM directory contains code to support high-level access to the onboard 1 Mbit AMD flash ROM (4 Mbit for AMD 29F040). The code supports byte, word, device, and sector programming along with device and sector erasure. Source code for the functions is given in the \PERIPH\FLASH 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 SBC6x 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.

USB Peripheral Drivers. The BUS directory contains code that implements a high-level queue based streaming interface with the host using the National Semiconductor USB chip to allow data transfer to the host at rates beyond that supported by the serial port. While source code for the functions is given in the `\PERIPH\BUS` for the system, the complexity of USB is such that altering this code is not recommended by Innovative Integration. On the host side a device driver is included that when installed will detect the SBC6x and allow the host to communicate with the target.

Further details of the USB interface are given in the section on the Bulk Transport Interface for the SBC6x.

Note that the USB interface requires Windows 98 or Windows 2K to function.

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 or 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: A4D4, A4D1, AIX, SD, SD16 and A16D2 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 A4D1 peripheral drivers provide control functions for the optional A4D1 plug-in module. The functions provide high-level C access to the module's four, 10 MHz, 14-bit analog input channels and single 15 MHz analog output channel. Source code for the functions is given in the `\PERIPH\ANALOG\A4D1` directory.

The A16D2 peripheral drivers provide control functions for the optional A16D2 plug-in module. The functions provide high-level C access to the module's sixteen, 12.5 kHz, 16-bit analog input channels and two 200 kHz analog output channels. Source code for the functions is given in the `\PERIPH\ANALOG\A16D2` directory.

The SD peripheral drivers provide control functions for the optional SD plug-in module. The functions provide high-level C access to the SD 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 SD16 peripheral drivers provide control functions for the optional SD16 plug-in module. The functions provide high-level C access to the SD16 module's sixteen, audio-grade, 18-bit analog input and output channels. Source code for the functions is given in the `\PERIPH\ANALOG\SD16` 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:

Library	Operation
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 6. Texas Instruments Standard Library Functions

SBC6x 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 access each peripheral.

The development system provides routines to access all integrated SBC6x 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`.

Function	Address	C Language Mneumonic	Memory Space
16750 UART	0x0400000	Periph->Duart	CE0
USB chip-level Reset	0x0410000	Periph->UsbReset	
USB Registers	0x0420000	Periph->Usb	
USB DMA Acknowledge	0x0430000	Periph->UsbDmaAck	
Embedded application boot detect	0x0440000	Periph->BootApp	
Fifo Port status readback	0x0450000	Periph->FportStatus	
Miscellaneous Control	0x0460000	Periph->Misc	
AD9850 Reset	0x0470000	Periph->DDS[0]	
AD9850 Frequency Update	0x0480000	Periph->DDS[1]	
AD9850 Write Clock	0x0490000	Periph->DDS[2]	
Digital I/O Data Register	0x04A0000	Periph->Dio[0]	
Digital I/O Direction Control	0x04B0000	Periph->Dio[1]	
Digital I/O Input Latch Clock Control Register	0x04C0000	Periph->Dio[2]	
External Mux Control 0	0x04D0000	Periph->Mux[0]	
External Mux Control 1	0x04E0000	Periph->Mux[1]	
Internal Timer 0 Clock Source Control	0x04F0000	Periph->Telk0Source	
External Interrupt Input 4 Select	0x0500000	Periph->Interrupt[0]	
External Interrupt Input 5 Select	0x0510000	Periph->Interrupt[1]	
External Interrupt Input 6 Select	0x0520000	Periph->Interrupt[2]	
External Interrupt Input 7 Select	0x0530000	Periph->Interrupt[3]	
I/O Module Strobe 0	0x0540000	Periph->Module[0]	
I/O Module Strobe 1	0x0550000	Periph->Module[1]	
I/O Module Strobe 2	0x0560000	Periph->Module[2]	
I/O Module Strobe 3	0x0570000	Periph->Module[3]	
I/O Module Strobe 4	0x0580000	Periph->Module[4]	
I/O Module Strobe 5	0x0590000	Periph->Module[5]	
I/O Module Strobe 6	0x05A0000	Periph->Module[6]	
I/O Module Strobe 7	0x05B0000	Periph->Module[7]	
Flash ROM	0x1400000	Periph->Prom[0..0x100000]	CE1
SDRAM (16Mbyte) (optional)	0x2000000	Periph->SDRam[0..0x400000]	CE2
Fifo Port Data	0x3000000	Periph->Fport	CE3

TABLE 7. SBC6x External Peripheral Memory Map

This document does not describe peripheral hardware specifications and other hardware issues. Refer to the *SBC6x Hardware* section of this 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 the 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	Periph->Dio[0]
Digital I/O Direction Control (4 bytes)	Periph->Dio[1]
Digital I/O Latch Control	Periph->Dio[2]

TABLE 8. Digital I/O Access Memory Location

The `Periph->Dio[0]` 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[1]` 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.DirectionRegister 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 9. Digital I/O Direction Configuration

The `Periph->Dio[2]` 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 10. 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
<code>DIO_dir()</code>	Sets the direction of all four bytes of the onboard 32-bit digital I/O port.
<code>DIO_read()</code>	Returns current state of all 32-bits of digital I/O port.
<code>DIO_write()</code>	Sets current state of all 32-bits of digital output port currently configured for output.
<code>DIO_latchcontrol()</code>	Sets the latch method of all four bytes of the onboard 32-bit digital I/O port.

TABLE 11. 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 three timebase sources built in to the SBC6x: two 32-bit timers internal to the 'C6201 processor 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 *SBC6x Hardware* section of this manual for more details on how to set up the SBC6x board.

C Language Timer Functions. The following functions give high-level access to the timer hardware.

Function Name	Operation
timebase()	Configures a specified timer channel (3..5) for periodic counting at a specified frequency using a specified source clock rate.

TABLE 12. C Language Timer Function

The `timebase()` function can be used to set a particular timebase to a particular frequency. For example, the following call sets internal timer channel 1 to generate a 1000 Hz output pulse stream, assuming the system CPU clock as the timebase.

```
timer(5, 1000.0, MHZ);
```

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 `timebase()` 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 9850 DDS timer device, see the data sheet. 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

The C `stdio` terminal emulation is provided in the Peripheral Library. The `stdio` library communicates with the host `UNITERMINAL.EXE` program via the TI 16C750 UART to provide `stdio` support to DSP applications running on the SBC6x. 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. Refer to the Target DSP Peripheral Libraries chapter 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 13. STDIO Driver Functions

Using Interrupts

The SBC6x 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 interrupt isr_fcn()           for C handlers or
isrfcn                            for assembly language
```

where `isr_fcn` can be any legal C function name. For each interrupt, a procedure must be coded which will be executed upon acknowledgment of the appropriate by the 'C6201. This is described in Chapter 6 the TMS320C6x C Compiler Users Manual.

Consider the following code example:

```
//
// EXAMPLE.C
//
```

```

main()
{
    EnableInterrupts();                // Enable unmasked xrpts

    timebase(5, 1000.0, MHZ);         // Internal timer 1 at 1kHz

    // install interrupt handler on TINT1

    InstallIntVector(ms_isr, TCLK1_INTERRUPT);

    EnableInterrupt(TCLK1_INTERRUPT);

        .                               // Bulk of application
        .

    DisableInterrupt(TCLK1_INTERRUPT); // Disable TINT0 xrpt
}

//

// ISR for timer 0 - Tally a variable

//

uint32 milliseconds

void interrupt ms_isr()
{
    milliseconds++;                  // Internal timer 1 is used to

}                                    // synthesize a timebase

```

In this code, the internal timer 1 is configured to output a pulse every millisecond which drives TCLK1 on the 'C6201. The vector is installed into the jump table with a call to InstallIntVector() and the bit associated with TCLK1 in the interrupt enable register, is enabled. Finally, main() calls EnableInterrupt(TCLK1_INTERRUPT) which sets the appropriate bit in the interrupt enable register so that all unmasked interrupts can be processed.

Each time the counter expires, the routine `ms_isr()` 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. If an application needs to make use of interrupts, those vectors which are affected need to be changed with `InstallIntVector()` at run time.

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

The SBC6x Bulk Transport Interface

The USB interface built onto the SBC6x provides a means for the rapid transport of data from the target to the host, which many applications will find useful. In order to achieve this speed, however, the USB system requires considerable interaction from the target side and considerable support on the host side as well. In order to isolate users from the details of this mechanism, Innovative Integration has layered a software interface over the raw USB software, thus insulating the user from most of the details of interacting with the USB system itself.

NOTE: USB support is only provided for Windows 98 and Windows 2K systems. Do not attempt to use these functions in Windows 95 or WinNT installations!

Overview of the Bulk Transport Interface

USB allows each target to define a number of logical targets that constitute separate unidirectional data streams between the target and host. These targets are termed “endpoints”. The six USB Endpoints available on the SBC6x are arranged into three “Channels”, each of which provides bi-directional movement of data from host to target, independent of the others. These channels can be opened separately for possible multiple application support.

The channels do not all have the same efficiency, due to the different FIFO depths on the USB hardware itself. Channels 0 and 1 have a 32-byte FIFO, while channel 2 has a 64-byte FIFO. This difference means that there is a substantial improvement in data transfer speeds when using the larger FIFO. Channel 2 has been measured at 300,000 Bytes/sec, while channels 0-1 transfer about 200,000 Bytes/sec.

On the target, each opened channel is associated with two QUEUE structures laying out portions of memory for storing data for transmission and the reception of data. These queues can then be examined for data that has arrived. Data inserted in the outbound queue will be sent to the host as soon as possible, with the caveat that ordinary data transmission is in integral numbers of FIFO blocks. Any residue will not be sent, unless an explicit “Flush” command is sent to force the sending of the partial packet. Because of the loss of efficiency inherent in the transmission of partial packets, the user should be careful to use transfers that evenly fill out a FIFO.

Target API for Bulk Transport

The API for the Bulk Transport is intended to resemble the familiar "Read/Write to File" interface style familiar to most programmers. Note that both read and write functions will 'block' to assure all data is transferred. If blocking is not desired, the "Data Available" functions should be used to see if the request would be satisfied before actually calling the read or write function itself.

Function Name	Operation
InitBulkTransport()	Sets up the Bulk Transport interface and enumerates the USB connection.
StopBulkTransport()	Shuts down the Bulk Transport interface.
IsBulkTransportReady()	Returns TRUE is the enumeration of a channel has completed.
OpenBulkTransport()	Opens one of the Bulk Transport channels.
CloseBulkTransport()	Closes an opened Bulk Transport channel.
ReadBulk()	Reads a block from a channel. Will block if data is not available.
WriteBulk()	Writes a block to a channel. Will block if room is not available.
BulkDataAvailable()	Returns the number of integers available in the inbound queue.
BulkSpaceAvailable()	Returns the number of integers available in the outbound queue.
FlushBulk()	Flushes any partial packets in the outbound queue.

TABLE 14. Bulk Transport System Target Functions

Refer to the Target DSP Peripheral Libraries section of this manual for additional details on these functions.

Host USB Support

USB Device Driver Installation. Supporting USB on the host requires a device driver be loaded to link the specifics of the SBC6x interface to the native USB support provided in Windows 98 or Windows 2K. This driver, SB62USB.SYS, must be installed before host applications will be able to provide support for USB data transfer.

The best way to install this driver is to place its installation files on the hard disk, then run one of the example programs supplied in the Zuma distribution. When the USB enumeration begun by the target detects the SBC6x, a device it has never seen. It will prompt the user for a driver in the same way that the installation of a PCI card does on the first system start following the installation of the card.

From the dialog box of the "Add New Hardware" expert now displayed, select the option to load the driver from a disk supplied by the hardware manufacturer. To select the driver, browse to the directory where the USB driver and INF file are located on the hard disk and select the SB62USB.INF file provided. This will install the driver and in all future cases will automatically install the driver when a SBC6x is detected on the USB chain.

Host API for Bulk Transport

On the host, the API for the Bulk Transport is more complicated than on the target because two interfaces are supplied for the programmer to choose between. The BULK interface, provides a support that assumes a deeper knowledge of Windows I/O to use effectively, but given that, it adds a little overhead

that may not be desired in some cases. The STREAMS interface adds queues on the host side that accept data sent from the target and stores it, and transfers data entered into the outbound queue to the target as time permits. The advantage of this is that the details of posting I/O requests and handling received packets are taken care of internally by the system. Therefore, the user application need not concern itself with the details of overlapped I/O, or multiple thread support. Instead, the application need just insert data to be transmitted into the outbound queue and read data from the incoming queue. This is the API recommended by Innovative Integration, unless there are specific reasons for using BULK operations.

Function Name	Operation
BULK_GetNumDevices()	Returns the number of SBC6x USB devices detected.
BULK_OpenDevice()	Opens the USB Device for a SBC6x board.
BULK_CloseDevice()	Closes the USB Device for a SBC6x board.
BULK_GetNumChannels()	Returns the number of channels supported on a device.
BULK_OpenChannel()	Opens a channel in BULK style.
BULK_CloseChannel()	Closes a BULK channel.
BULK_Read()	Reads a block from a BULK channel.
BULK_Write()	Writes a block to a BULK channel.
BULK_GetOverlappedReadResult()	Returns the Overlapped result on the read of a channel.
BULK_GetOverlappedWriteResult()	Returns the Overlapped result on the write of a channel.
BULK_Cancello()	Cancel pending I/O requests on a channel.
STREAM_Open()	Open a stream on a channel.
STREAM_Close()	Close a stream on a channel.
STREAM_ReadAvailable()	Returns the amount of data available to read from the stream.
STREAM_WriteAvailable()	Returns the amount of space available to write data into the stream.
STREAM_Read()	Read a portion of the data from the stream.
STREAM_Write()	Write data into the output stream.
STREAM_Flush()	Flush remaining output data to the target.
USB_VendorCommand()	Not Implemented.

TABLE 15. Bulk Transport System Host Functions

Refer to the Host DLL Reference section of this manual for additional details on these functions.

Example Programs for the Bulk Transport Interface

Mfctest. The Mfctest program is intended to test the throughput of the USB BUS using all three pipe lines independently. It uses the BULK interface to the USB device. It can be configured to optionally use overlapped I/O. This application was built using MSVC++v 5.0.

Stream. The "Stream" test is intended as a loopback test. This directory consists of two components, the Barber and the Stream. The Barber is a simple Ascii-text barber pole generator. It generates a stream of visible characters, as they wrap from line to line across the display form apparent diagonal stripes. This display makes dropped or missing characters apparent since the disruption of the "stripes" is easily picked up by the eye.

Stream is a simple loopback program that receives data from STDIN and transfers it across to the SBC6x through the USB device, and then back. The characters received are then finally printed to STD-

OUT. These are both console mode programs. To use the programs from a Windows Command Prompt types:

```
barber | stream
```

Stream uses the preferred STREAM interface to the USB device. The programs were built using MSVC++5.0.

Testapp. The Testapp program is a basic Borland C++ Builder application. It is a GUI-implementation of the Stream loopback program. This application uses the STREAM interface to the USB device.

This application was built using C++Builder v3.0.

Building Flash Programmable Applications

The SBC6x's flash EEPROM and automatic power-up bootload support provide the means to allow customers to program applications into the SBC6x's EEPROM memory. An application program installed in this non-volatile memory may be optionally bootstrap loaded automatically on power-up or board reset, allowing the SBC6x to operate in an isolated environment with no requirement for connection to a host computer.

The SBC6x Zuma Toolset contains software support for the SBC6x's bootload capability, and allows the user to easily generate custom applications and program them into the SBC6x flash EEPROM for automatic start-up. The program build process is supported under the Code Composer Studio environment, and the Zuma Toolset includes a flash programming utility, which is used to embed the finished application into flash EEPROM.

The program build process for creating an application intended for programming into the flash memory is identical to that that used to generate COFF files for download using a JTAG debugger or the UNITERMINAL application. However, after the executable .out file has been created, it must be converted into binary, boot-loadable format in order to be placed into flash ROM using the supplied BURN.EXE applet. This operation is accomplished using the PromImage.exe applet, provided in the Zuma toolset.

Follow the sequence below in order to embed an application into Sbc6x flash memory.

1. Build and debug the embedded application using Code Composer Studio.
2. Use the PromImage applet to convert the application COFF file into proprietary boot-loadable format.
3. Run the Burn applet to embed the boot load image into sector one of Flash memory.
4. Install the boot jumper onto the Sbc6x. Reset the Sbc and the application will load and run from flash.

The SBC6x Zuma Toolset package includes an example project, EMBED, which is set up for generating flash programmable object files and which may be used as a template for creating custom applications.

Example Target Programs for the SBC6x

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 SBC6x cards 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 with the Code Composer Studio environment by loading the HELLO.MAK project from the `\target\examples` directory, and rebuilding the project by selecting Project/Rebuild All from the menu. Refer to the Code Composer Studio 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 SBC6x card on which HELLO is running, the application will appear to hang at the first instance of a stdio function call (usually a `getint()` or `putint()` call). This is due to the fact that all stdio calls use the SBC6x bus mailbox interface and are handshaken with the host terminal emulator application,. 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 SBC6x to double-check proper hardware functionality. As such, it contains routines for exercising each of the peripherals on the SBC6x, 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.

EMBED

EMBED is a simple example which shows how to build applications intended for programming into the SBC6x's flash EEPROM memory and which are therefore compatible with the Zuma Toolset's flash burn utility. Programming an application into flash EEPROM allows the SBC6x to boot load the application on powerup and to operate as a stand-alone device without need for connection to a host computer.

The EMBED program itself simply initializes the serial interface and sends a logon character similar to the one used by the TALKER program. This allows UNITERMINAL to recognize the SBC6x's connection to the serial port before dropping into a loop which toggles all the digital I/O interface bits in output mode. This allows the user to verify the successful boot loading and operation of the EMBED program once it has been programmed into flash EEPROM and the SBC6x power is cycled.

For more information regarding the processing of applications for programming into flash EEPROM, refer to the section entitled "Building Flash Programmable Applications" in this section. For more information concerning the flash programming utility, refer to the section entitled "The Flash Programmer" in the Support Applets section of Chapter 3.

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
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

Category	Name	Description	
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
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.
putchar		ANSI put character to console.
C Standard I/O Library Emulation Functions		fclose
	ferase	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.
	type	Send a character string to the terminal emulator.
Terminal Applet Extensions	get_busmaster_addr	Retrieve host busmaster address from the terminal emulator.
	plot	Transfer data buffer to host for plotting.
	stdio_reset	Reset the terminal emulator program.
	stdio_terminate	Send the termination code to the terminal emulator.

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.
	vmul	Multiply two vectors into a third vector.
	Matrix Functions	matrix_add
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.
matrix_transpose	Transpose a matrix, return pointer to new MATRIX.	
Queue Support Functions	dequeue_ptr	Remove data from a queue and adjust pointer.
	enqueue_ptr	Load data into Queue and update pointers.
	enqueued	Return count of data elements in a Queue.
	queue_init	Initialize memory Queue structure.
BERR Sequence Generation Functions	berr_decode	Tests a value in a BERR sequence.
	berr_encode	Generate the next value in a BERR sequence.
	berr_initialize	Set up a BERR sequence generator.
Data Compression Functions	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 `BASEBOARD\INCLUDE\HOST\TARGET.H` file. The names of these functions are aliai of the actual board-specific library function names, which are proto-typed in `BASEBOARD\INCLUDE\HOST\ALIAS.H`. For examples on using the DLL, install `DspComponents` from the installation CD.

TABLE 16. Generic DLL Function List

Category	Function Prototype	Function Description
General	<code>BOOL target_open(int target)</code>	Opens driver for specified target DSP board. Returns boolean.
	<code>BOOL target_close(int target)</code>	Closes driver for specified target DSP board. Returns boolean.
	<code>LPVOID target_cardinfo(int target);</code>	Returns address of <code>cardinfo</code> structure for target.
	<code>int iicoffld(char *, int target, HWND hParent);</code>	Loads a COFF executable file onto target DSP.
Interrupt Functions	<code>BOOL host_interrupt_enable(int target);</code>	Enables a previously installed virtual interrupt handler.
	<code>BOOL host_interrupt_disable(int target);</code>	Disables a previously enabled virtual interrupt handler.
	<code>void host_interrupt_install(int target, void (*virtual_isr)(void *), void * context);</code>	Installs a virtual interrupt handler.
	<code>void target_interrupt(int target);</code>	Interrupts target DSP board.
	<code>void host_interrupt_deinstall(int target);</code>	Removes a virtual interrupt handler.
	<code>void mailbox_interrupt(int target, unsigned int value);</code>	Interrupts the target DSP after writing value to special mailbox.
	<code>unsigned int mailbox_interrupt_ack(int target);</code>	Acknowledges target to Host interrupt, returns special mailbox contents.

Control Functions	<code>void target_reset(int target);</code>	Physically asserts reset on the target DSP board.
	<code>void target_run(int target);</code>	Deasserts reset on the target DSP board.
	<code>void target_outport(int target, int port, int value);</code>	Outputs a value to specified DSP board I/O port address.
	<code>int target_inport(int target, int port);</code>	Inputs a value from specified DSP board I/O port.
	<code>void target_opreg_outport(int target, int port, int value);</code>	Outputs a value to specified DSP board operation port address.
	<code>int target_opreg_inport(int target, int port);</code>	Inputs a value from specified DSP board operation port.
	<code>void target_control(int target, int bit, int state);</code>	Modifies a bit in the control register of the target DSP board.
Mailbox Functions	<code>int read_mailbox(int target, int);</code>	Reads the specified mailbox of the target DSP board.
	<code>void write_mailbox(int target, int, int);</code>	Writes to the specified mailbox of the target DSP board.
	<code>BOOL check_outbox(int target, int);</code>	Interrogates the specified output mailbox status.
	<code>BOOL check_inbox(int target, int);</code>	Interrogates the specified input mailbox status.
	<code>int read_mb_terminate(int target, int, int *, int wide);</code>	Reads the specified input mailbox, if full.
	<code>int write_mb_terminate(int target, int box_number, int value, int wide);</code>	Writes to the specified output mailbox, if empty.
	<code>void clear_mailboxes(int target);</code>	Clears all mailboxes to empty state.
	<code>int target_key(int target);</code>	Reads terminal mailbox, returns an 8-bit contents.
	<code>void target_emit(int target, int value);</code>	Writes 8-bit value to terminal mailbox.
	<code>void target_Tx(int target, int value);</code>	Writes 32-bit value to terminal mailbox.
	<code>int target_Rx(int target);</code>	Reads 32-bit value from terminal mailbox.
Bulk Transport Interface Functions	<code>int BULK_GetNumDevices();</code>	Returns the number of SBC62 USB devices detected.
	<code>BOOL BULK_OpenDevice(int iDevice, HANDLE *phDevice)</code>	Opens a device for BULK transport access.
	<code>BOOL BULK_CloseDevice(IN HANDLE hDevice)</code>	Closes a device for BULK transport access.
	<code>BOOL BULK_OpenChannel(int iDevice, WORD wChannel, BOOL fOverlapped, BULK_HANDLE *pHandle);</code>	Opens a data channel in BULK mode.
	<code>BOOL BULK_CloseChannel(BULK_HANDLE Handle)</code>	Closes a data channel opened with <code>BULK_OpenChannel()</code> .
	<code>BOOL BULK_Read(BULK_HANDLE Handle, LPVOID lpBuffer, DWORD dwNumberOfBytesToRead, LPDWORD lpNumberOfBytesRead, LPOVERLAPPED lpOverlapped);</code>	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. (in integers)
	WORD STREAM_ReadAvailable(BULK_HANDLE handle)	Returns the amount of data available on the STREAM channel. (in integers)
	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 (I.I.) Developers Packages and 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 I.I. libraries. Note that several 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 I.I. 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
II_BOARD	Dev Pkg Applets	set II_BOARD=<board dir> ie set II_BOARD=c:\SBC6x
C_DIR	All TI C Compilers All I.I. peripheral libraries	set C_DIR=%ii_board%;%ii_board%\include\target;%ii_board%\lib\target;%compiler dir%\rt dx\include;%compiler dir%\rt dx\lib;%compiler dir%\cgtools\include;%compiler dir%\cgtools\lib;%compiler dir%\bios\include;%compiler dir%\bios\lib; ie set C_DIR=C:\SBC6x;C:\SBC6x\include\target;C:\SBC6x\lib\target;C:\ti\c6000\rt dx\include;C:\ti\c6000\rt dx\lib;C:\ti\c6000\cgtools\include;C:\ti\c6000\cgtools\lib;C:\ti\c6000\bios\include;C:\ti\c6000\bios\lib; Specified order is critical!
A_DIR	All TI Assemblers	Same as C_DIR above.

D_DIR	TI Debuggers	set D_DIR=<debugger dir> ie set D_DIR=c:\ti\c600\evm6x\lib
D_SRC	All Debuggers	set D_SRC=<source code dir1>;<dir2>;...;<dir n> ie set D_SRC=c:\SBC6x\stdio;c:\SBC6x\dsp; c:\SBC6x\periph\analog;c:\SBC6x\periph\digital;... ;c:\SBC6x\periph\bus
PATH	All II products All TI Tools	set path=%path%;%ii_board%;%ii_board%\lib\host; %compiler dir%\cgtools\bin;%compiler root%\bin ie set path=%PATH%;C:\SBC6x;C:\SBC6x\lib\host;C:\ti\c6000\cgtools\bin;C:\ti\bin

TABLE 17. Required Disk Directory Structure for II Development Tools.

The Innovative Integration, TI, C Development System for the SBC6x requires the following environment variables be set properly for correct operation:

set ii_board=C:\SBC6x

set C_DIR=C:\SBC6x;C:\SBC6x\include\target;C:\SBC6x\lib\target;C:\ti\c6000\rt dx\include;
C:\ti\c6000\rt dx\lib;C:\ti\c6000\cgtools\include;C:\ti\c6000\cgtools\lib;C:\ti\c6000\bios\include;C:\ti\c6000\bios\lib;

set A_DIR=%C_DIR% (same as c_dir)

set d_src=C:\SBC6x;C:\SBC6x\stdio;C:\SBC6x\periph\bus;C:\SBC6x\periph\digital;
C:\SBC6x\periph\misc;C:\SBC6x\dsp;C:\SBC6x\periph\analog;C:\SBC6x\periph\flash;
C:\SBC6x\periph\bus\polled

set path=%PATH%;C:\SBC6x;C:\SBC6x\lib\host;C:\ti\c6000\cgtools\bin;C:\ti\bin

SBC6x Hardware Functions

The SBC6x is a stand alone digital signal processor (DSP) card based around the Texas Instruments TMS320C6x01 processor. Implementing a modular I/O expansion system, the SBC6x is particularly well suited to data acquisition and control tasks. The SBC6x is also supported by a collection of I/O bus function cards which, provide hardware interfacing to real-world equipment.

The SBC6x's features include:

1. TMS320C6x01 processor.
2. Optional external one wait-state SDRAM memory pools.
3. FIFOPort digital I/O expansion capability.
4. OMNIBUS I/O expansion compatible (two available slots).
5. 32 bits of digital I/O.
6. Two high speed serial port connectors.
7. RS232 serial port.
8. Universal Serial Bus (USB) port.
9. External mux board control connectors (compatible with external TERM board).
10. JTAG hardware emulation support.

The following figure gives a block diagram of the SBC6x.

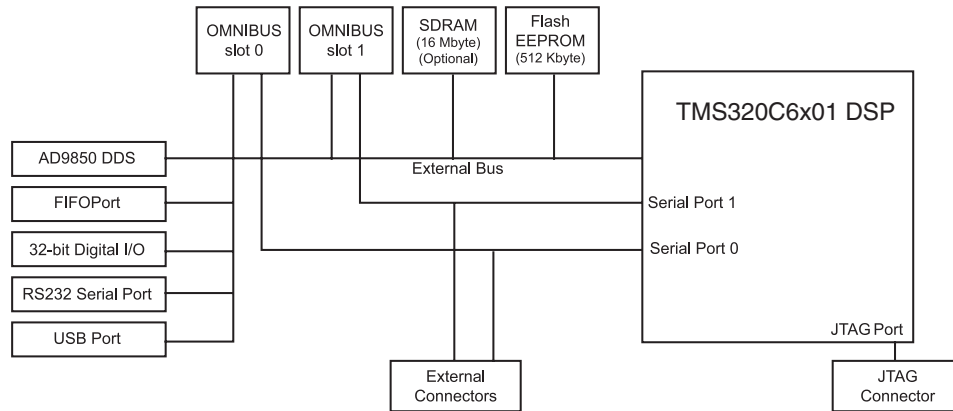


FIGURE 34. SBC6x Block Diagram

Memory Map

The SBC6x processor operates in 'C6x01 ROM boot mode with memory map type 1. In this mode, applications boot from external flash EEPROM and run from the on-chip memory.

The following figure gives the processor memory map of the SBC6x for external peripherals and memory. Please note that this table ignores any on-chip resources.

Function	Address	Memory Space
RS232 Serial Port	0x400000	CE0
USB Reset	0x410000	
USB Register Set	0x420000	
USB DMA Acknowledge	0x430000	
Application Boot Jumper Readback	0x440000	
FIFO Port Status	0x450000	
Miscellaneous Control	0x460000	
AD9850 Reset	0x470000	
AD9850 Frequency Update	0x480000	
AD9850 Write Clock	0x490000	
Digital I/O Data Register	0x4A0000	
Digital I/O Direction Control	0x4B0000	
Digital I/O Input Latch Clock Control Register	0x4C0000	
External Mux Control 0	0x4D0000	
External Mux Control 1	0x4E0000	

Timer Clock Source Control	0x4F0000	
External Interrupt Input 4 Select	0x500000	
External Interrupt Input 5 Select	0x510000	
External Interrupt Input 6 Select	0x520000	
External Interrupt Input 7 Select	0x530000	
I/O Module Strobe 0	0x540000	
I/O Module Strobe 1	0x550000	
I/O Module Strobe 2	0x560000	
I/O Module Strobe 3	0x570000	
I/O Module Strobe 4	0x580000	
I/O Module Strobe 5	0x590000	
I/O Module Strobe 6	0x5A0000	
I/O Module Strobe 7	0x5B0000	
Flash EEPROM (512Kbyte)	0x1400000	CE1
SDRAM (16Mbyte) (optional)	0x2000000	CE2
FIFOPort	0x3000000	CE3
FIFOPort Reset	0x3010000	
FIFOPort Transmit FIFO PEN Mode	0x3020000	

TABLE 18. SBC6x External Memory Map

SBC6x Hardware Initialization Requirements

The SBC6x design requires the following values to be written to its hardware control registers in order to provide access to on-board hardware:

Register	Address	Value
EMIF Global Control	0x01800000	0x0000306E
CE1 Control	0x01800004	0x48342020
CE0 Control	0x01800008	0x73E70F22
CE2 Control	0x01800010	0x00000030
CE3 Control	0x01800014	0x30800520
SDRAM Control	0x01800018	0x07117000
SDRAM Refresh	0x0180001C	0x00000618
Interrupt Polarity	0x019C0008	0x0000000F

TABLE 19. SBC6x Bus Control Register Initialization Values

These values are initialized automatically by C programs compiled under the SBC6x Development Package software libraries. Be sure to include initialization of these values whenever software is developed outside the Development Package or when a JTAG hardware assisted debugger is employed for code downloading to the SBC6x (i.e. when using Code Composer Studio or any other JTAG debugger package).

External Memory

The SBC6x offers two types of external memory: synchronous DRAM (SDRAM), and flash EEPROM. The 512Kx8 EEPROM memory comes standard with the SBC6x, while the SDRAM is optional.

The flash EEPROM memory contains the SBC6x's start-up software. Shipped standard with Innovative's TALKER program installed for use with the SBC6x Development Package, the EEPROM may be programmed with end user applications through the use of tools included in the Development Package. The EEPROM and boot process makes the SBC6x a stand-alone processor board capable of remote control and data acquisition tasks. The EEPROM operates with several wait states on the processor bus.

EEPROM memory is implemented on the SBC6x with a single AMD 29F040 flash device. In addition to its intended purpose as the boot device for application software, the 29F040 may also be used to store nonvolatile application-dependent data. The device is connected to the C6x01 processor on the least significant eight bits of the processor bus, and responds to standard AMD programming sequences. The SBC6x Development Package contains support software for erasing and programming the flash device.

The optional SDRAM memory provides a large, fast area to store copious amounts of data or program information. The 16 Mbyte SDRAM option operates at one half the processor's bus rate, for a maximum throughput of 400 Mbytes/sec.

SBC6x OMNIBUS

The SBC6x I/O bus provides a modular, high-speed expansion area which is directly tied to the processor's bus and is ideally suited for I/O hardware expansion. Direct memory-mapped accesses allow the processor to transfer data to and from I/O bus peripherals constructed as plug-in modules, which can be mixed and matched to suit the particular user's functional requirements.

The OMNIBUS slots are accessed as memory-mapped peripherals with the SBC6x providing four decoded chip select signals per slot. The following figure gives the memory map for the OMNIBUS slots, and shows the decode signal to slot mapping.

Function	Starting Address	Module Slot
OMNIBUS Strobe 0	0x0540000	0
OMNIBUS Strobe 1	0x0550000	0
OMNIBUS Strobe 2	0x0560000	0
OMNIBUS Strobe 3	0x0570000	0
OMNIBUS Strobe 4	0x0580000	1
OMNIBUS Strobe 5	0x0590000	1
OMNIBUS Strobe 6	0x05A0000	1
OMNIBUS Strobe 7	0x05B0000	1

TABLE 20. SBC6x I/O Bus Memory Mapping

Each module site provides a 32-bit wide data bus connection to the processor's data bus, with 12-bits of low order address signals for additional decoding beyond the four chip select signals available per slot. Each module also connects to a 'C6x01 serial port (serial port zero for slot zero, and serial port 1 for slots 1 and 2) to allow serial port driven I/O. Bus reset, RDY, R/W, and processor clock signals are available, as are power connections for digital 5V and analog +5V and +15V. Timebase connections include timer channels from both the 16-bit timers and the 9850 direct-digital synthesizer.

Each OMNIBUS slot has a 50 pin undedicated connector (JP11 on slot 0 and JP16 on slot 1) for use in providing external I/O to and from a module installed in the slot. The slot's I/O connector is in turn pinned out to a 50 pin .100" square double row header (JP12 for slot 0, JP17 for slot 1) for use in attaching cables from external hardware.

Connector pinouts for the module sites are provided in the appendices. Individual pin functions are noted in the tables, and in general the OMNIBUS pinout represents a direct connection to the 'C6x01 local bus.

SBC6x OMNIBUS Memory Mapping

Since the 'C6x01 processor is a byte addressable machine which implements its address bus based on a 32-bit transfer width (i.e. the address bus starts at A2 and separate byte enable pins are supplied to control accesses to individual bytes within the 32-bit wide location denoted by the address bus), users must take care when writing software which performs OMNIBUS accesses.

The OMNIBUS specification requires 32-bit accesses and does not support byte or half-word (16-bit) accesses. No support is included in the specification for the 'C6x01's byte enable pins. This means that software performing accesses must always perform 32-bit transactions with the OMNIBUS modules. When writing C code for the SBC6x, programmers should use only variables of type int or unsigned int (or their derived types), and all accesses should be word justified (the least significant nibble of the address must always be a multiple of four). Accesses generated using pointers to variables of type char, short, or long will cause erroneous non-32-bit accesses. Correct OMNIBUS module operation under these situations can not be guaranteed.

Please note that memory decoding within the OMNIBUS decode regions uses 32-bit addressing and that the memory map tables given in the *OMNIBUS Manual* should be treated appropriately. For example, the description of the OMNIBUS DIG module notes that the byte 3 direction control register for a module installed in site 0 is mapped to address IOMOD2 + 3. This address should be literally interpreted as 0x56000C, where IOMOD2 is equal to 0x560000 and the offset adds decimal 12 (three 32-bit words of offset). IOMOD2 + 3 should NOT be interpreted as 0x560003, since the offset is 3 32-bit words and not 3 bytes.

This addressing is most easily handled in C by using integer pointers and integer pointer arithmetic, which will always result in the required address alignment. For example, the following code defines a pointer and accesses the byte 3 direction control register with the documented offset:

```
unsigned int *pointer = 0x560000;

*(pointer + 3) = 0x0;    /* set byte 3 to output mode */
```

The actual accessed memory location is 0x56000C, due to the way pointer math is handled in C.

OMNIBUS Power

The OMNIBUS interface provides five separate power supplies for use by modules along with two separate ground return connections. The following table lists the power supplies and their power ratings. A separate digital 5V supply is provided along with separate digital grounds to minimize the digital noise present on the analog power supplies.

Pin Name	Voltage	Current Rating (max)
DVCC	5V (digital)	(System dependent)
AVCC	5V (analog)	500 mA
-AVCC	-5V	500 mA
+AV	+15V	(System dependent)
-AV	-15V	(System dependent)

TABLE 21. I/O Bus Power Ratings

Note: The SBC6x implementation of the OMNIBUS expansion slots deviates from the standard specification in the way the power supplies are handled. In order to provide for a more compact form factor for the host card, the SBC6x does not supply discrete +-12V power supplies as required by the OMNIBUS specification. Instead, it connects the +-AV rails to the +-12V OMNIBUS power pins, resulting in a 3V absolute overvoltage on those supply pins. Designers of custom OMNIBUS modules intended for use with the SBC6x should keep these revised power supply values in mind when planning circuitry, which connects to these power supplies.

Please note that the AGND and DGND busses are separated on the SBC6x and for proper ground referencing they must be tied together on modules which use the analog power supplies (any supply other than digital 5V, 12V, or -12V). Innovative Integration recommends that a ferrite bead (Panasonic EXCELSEA35V or equivalent, depending on expected ground return current from the analog power supplies) be used on custom modules to connect the two ground busses. This is to prevent high frequency digital noise on the DGND bus from polluting the clean AGND return.

FIFOPort I/O Expansion

The FIFOPort feature provides a single buffered bi-directional 16-bit interface which allows external hardware or other Innovative 'C6x01 processor boards to communicate with the SBC6x at high data rates. A single input 512x16 FIFO is provided to buffer incoming strobed parallel data, while a FIFOPort compatible output supports clocking data to external hardware or other FIFOPorts. The FIFOPort is memory mapped into the CE3 space.

The following diagram illustrates the FIFOPort's operation. The FIFO buffer memory serves to clock incoming data and store it for use by the 'C6x01 processor. Data is formatted as a 16-bit wide data bus synchronous with a rising edge strobe signal, which acts as the FIFO load clock. The output portion consists of the same two signals: output data plus the strobe signal for the receiving end of the port.

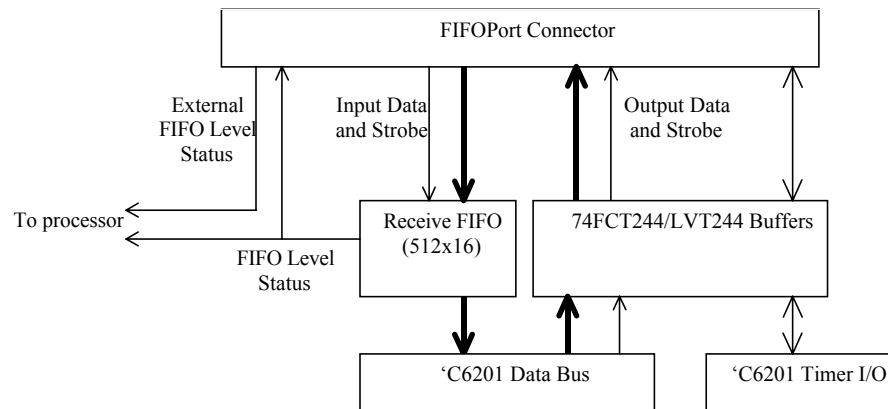


FIGURE 35. FIFOPort Block Diagram

The FIFOPort also provides external access to the receive FIFO's empty, full, and programmable almost full flags to allow hardware to monitor the FIFO's level status. The port can also receive FIFO level status from external hardware to allow the 'C6x01 processor to monitor level status of FIFOs located off the SBC6x card. Both the onboard receive FIFO level status and the off board FIFO status lines may be polled or may generate interrupts to the 'C6x01 processor.

In addition to the FIFO data management functions, access to the 'C6x01 timer I/O pins is provided to support simple bit I/O requirements. The timer I/O pins are buffered through LVT family logic buffers and driven on/off the card for use where individual bit I/O control is needed to external hardware.

Also available on the FIFOPort connector is an external interrupt input which is connected to the processor's interrupt switch matrix which allows the processor to receive an active low interrupt from external hardware.

Transmitting and Receiving FIFOPort Data

Data is transmitted and received on the FIFOPort by means of processor address location 0x03000000. EMIF read and write accesses (either due to CPU or DMA activity) causes read and write strobes to be generated to the FIFOPort circuitry only when this address is accessed.

In the case of a write access, an active high output strobe is generated on the external connector and 16-bit bus data is driven out to the output bits. This data should be latched by external hardware on the rising edge of the FIFOPort output strobe. Write accesses do not affect the current state of the receive FIFO.

In the case of a read, an input read strobe is generated to the receive FIFO and its output data latched by the processor. If the data item being read in the current cycle is not the last item stored in the buffer, the next data item is clocked out by the FIFO and held ready for the next read access by the processor. Read accesses do not generate output strobes to the external connector.

Please note that the data returned by the FIFO on a read access is present on the least significant 16-bits of the processor's data bus. The most significant 16-bits are not driven and are not defined. If 32-bit CPU accesses are being used to read data from the FIFO, then the upper 16 bits of the result should be masked off before the resultant value is used. The DMA controller may be programmed for 16-bit access width and will automatically perform 16-bit to 32-bit data translation such that each stored 32-bit wide data item retrieved will be the concatenation of two 16-bit values read from the FIFO.

If the receive FIFO grows empty, the last data item's value will be output on any subsequent read accesses.

Monitoring FIFO Status

The FIFOPort provides a FIFO level monitoring feature which allows software to read the receive FIFO's level indicators as well as FIFO level data from external hardware (if connected). The receive FIFO's empty, full, and programmable almost full flags can be read at any time by the CPU. Alternatively the interrupt selection matrix may be programmed to notify the CPU of level events via an interrupt (see Interrupts section for more information). The same functionality is provided for the external FIFO, allowing the CPU to read back or be interrupted by any of six different level state conditions.

The FIFO level status register resides at address 0x0450000 in the processor memory map with the bit configuration shown in the following figure.

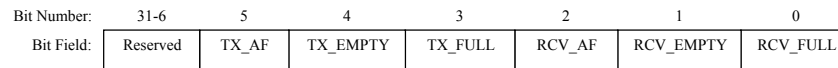


FIGURE 36. FIFOPort Level Status Register

Bit Field Name	Function
RCV_FULL	Receive FIFO Full Flag (1 = full, 0 = not full)
RCV_EMPTY	Receive FIFO Empty Flag (1 = empty, 0 = not empty)
RCV_AF	Receive FIFO Almost-full Flag (1 = almost-full, 0 = not almost-full)
TX_FULL	Transmit FIFO Full Flag (1 = full, 0 = not full)
TX_EMPTY	Transmit FIFO Empty Flag (1 = empty, 0 = not empty)
TX_AF	Transmit FIFO Almost-full Flag (1 = almost-full, 0 = not almost-full)

TABLE 22. FIFOPort Level Status Register Definition

The receive FIFO level bits are read directly from the FIFO hardware on the FIFOPort, while the transmit FIFO bits are read from the level input pins on the FIFOPort connector. If no external status is being reported by the hardware connected to the FIFOPort, then these bits will read as zeros (onboard 10K pullup resistors hold the transmit full, empty, and almost-full input pins high). If external FIFO level reporting is not desired, the level inputs may be used for application specific bit inputs to report

other hardware status conditions or trigger interrupts on the SBC6x processor. Note that this is in addition to the dedicated timer I/O pins and the processor interrupt input pin on the FIFOPort connector, which remain available regardless of the use of the FIFO status inputs.

By the appropriate programming, the FIFO levels may also be monitored using processor interrupts. The FIFO status bits are available as sources to the processor interrupt selection matrix. This technique is typically used to drive DMA transfers to and from the FIFOPort. Where one FIFO status interrupt triggers one or more transfers using DMA synchronization, or for CPU interrupts where the target CPU in a transfer wants to be interrupted when data (or space) is available in the FIFO. This would be typical of “one-shot” FIFO transfers, where a single full FIFO’s worth of data is transferred at once and the receiving processor needs to be notified when the FIFO reached the full state so that a read operation on the other side of the FIFO may commence. For more information on using the FIFO levels to trigger interrupts to the ‘C6x01 processors, see the Interrupts section.

FIFOPort Reset

The receive FIFO may be cleared and its condition reset at any time by accessing the FIFOPort reset register at address 0x3010000. The data written to the register is not critical: a write access of any data will result in the FIFO being reset. Upon reset, the FIFO levels are cleared and the flags change to reflect the FIFO empty status, and the programmable almost full control variables are reset to default values (see below for more information).

Controlling the FIFOPort Programmable Almost-full Flag

In addition to the fixed function empty and full flags, the FIFOPort provides a programmable almost-full flag, which can be used to enable notification on partial FIFO transfer lengths. This feature is particularly suitable to DMA block transfers on the FIFOPort because it maximizes the transfer rates on both sides of the FIFO by keeping the buffer partially filled.

The almost-full flag operates as follows. Given two initialization bytes (X and Y), the FIFO outputs an almost-full/almost-empty flag function, which is active whenever the FIFO contains X or less words of data or 512-Y or more words of data. By programming the X value equal to the almost-full level and the Y value to zero, the FIFO’s programmable flag effectively becomes a variable partial full indicator. For example, programming the X variable to 128 and the Y variable to 0 yields a quarter-full output function.

The programmable almost-full flag control variables for the transmit half of the FIFOPort are initialized by enabling PEN mode, then writing the variables to the FIFOPort. The PEN mode is enabled by writing a zero to the transmit FIFOPort PEN mode register at address 0x3020000. The X variable is then written to the FIFOPort, followed by the Y variable, with both data values being 8-bits wide and right justified on the bus. The default values for the X and Y variables are both 64 (the FIFO reverts back to these values on a reset). Following the completion of the Y variable write, PEN mode should be disabled by writing a 1 to the PEN mode register. Please note that the almost-full flag variables may only be written immediately after a FIFO reset has been issued to the transmit side FIFO and before any data is written to the transmit FIFO.

Note: the above description of the PEN mode register operation was a change to the SBC6x control logic made in April 1999. Boards purchased earlier than this date should be returned to Innovative for an update. Please contact Innovative with questions concerning this feature.

Please note that this initialization operation only affects the transmit FIFO (i.e. the FIFO on the external hardware or other FIFOPort compatible card). The FIFOPort architecture does not allow the onboard processors to initialize the programmable levels of the FIFOPort receive FIFOs. This initialization is always performed by the external hardware prior to writing data to the receive FIFO.

Timer I/O and the FIFOPort

The FIFOPort provides a connection to the processor's timer I/O pins. This allows designers of hardware connecting to the FIFOPort easy access to four bits of unidirectional I/O for control purposes and status reporting. The on-chip timers of the 'C6x01 may be programmed to generate or receive clock and count events on the pins, or the pins may be used for general purpose bit I/O.

The SBC6x implements LVT logic family buffering between the timer I/O pins and the FIFOPort connectors. Output and input levels are TTL compatible, but the outputs will not drive beyond 3.3V on the high side, and are tolerant of input voltages of up to 5V. This feature makes the FIFOPort timer I/O pins suitable for direct interfacing to 3.3V or 5V TTL compatible logic. Such logic families as HCT, LSTTL, FCT, ABT, and ACT may be directly connected to the FIFOPort timer I/O pins.

Designing External Hardware for use with the FIFOPort

Use caution when designing external hardware, which is to be connected to the FIFOPort. The signals present on the interface connector are extremely high speed and failure to handle them appropriately can cause functional problems with the FIFOPort as well as the SBC6x's onboard components. Innovative Integration does not recommend driving cables directly as capacitive load and ringing issues can cause corruption of the transmitted data. FIFOPort connector pinouts are given in the appendices at the end of this manual.

The following diagrams give timing information for the FIFOPort circuitry. This data is derived from device specifications and is not factory tested.

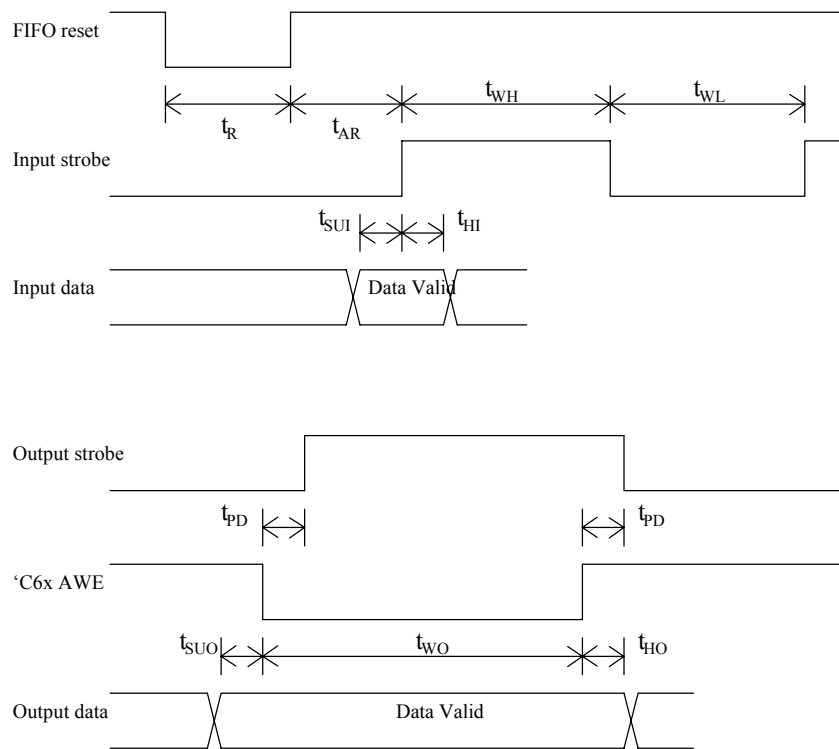


FIGURE 37. FIFO Port Timing

Parameter	min (ns)	max (ns)
t_{WH}	7	
t_{WL}	7	
t_{SUI}	5	
t_{HI}	0	
t_R	10	
t_{AR}	5	
t_{PD}		7
t_{SUO}	10^1	
t_{HO}	0^1	
t_{WO}	10^1	

TABLE 23. FIFO Port Timing Parameters

Notes ¹: Dependent on EMIF programming for CE0 space as well as processor cycle frequency. These values are determined from recommended EMIF register values.

All FIFOPort signals are TTL compatible CMOS inputs and outputs. All input pins may be safely driven with 5V logic devices. With the exception of a small subset of outputs (the timer output pins, external FIFO write clock, and AF level set control pin) all pins are driven from 5V logic: the rest are driven by LVTTL (3.3V) logic. This makes the FIFOPort compatible with devices with 5V tolerant inputs only. Do not use non-5V tolerant devices when designing hardware, which directly connects to the FIFOPort 5V signals.

'C6x01 Serial Ports

The 'C6x01's on-chip serial ports are pinned out to connectors JP6 (port 0) and JP7 (port 1) for use with external hardware. The serial ports are also connected to the OMNIBUS sites for use with modules designed to interface to the processor serially.

Pinouts for the serial port connectors are given in the appendices. Innovative Intergration recommends buffering these ports with off board hardware in order to preserve signal integrity.

RS232 Serial Port

The SBC6x implements a single asynchronous serial port channel with RS232 drivers compatible with standard PC serial port interfaces. The serial port is implemented using a Texas Instruments TL16C750 UART and Maxim MAX3223 RS232 transceiver. The following table gives the address mapping for the UART device.

16C750 Register	C6x Address
TXD	0x400000
RXD	0x400000
DLL (DLAB=1)	0x400000
DLM (DLAB=1)	0x400004
IER	0x400004
IIR	0x400008
FCR	0x400008
LCR	0x40000C
MCR	0x400010
LSR	0x400014
MSR	0x400018
SCR	0x40001C

TABLE 24. UART Control Registers

For hardware and programming details on the 16C750 UART, see the data sheet for the device (available on the Texas Instruments web site at www.ti.com).

RS232 serial port signals are present on connector JP3. Please note that the RS232 DTS pin (pin 7 on JP3) is used to control the processor reset: an RS232 high voltage signal on this pin holds the processor in reset, while an RS232 low voltage deasserts reset and allows the processor to execute code. The reset function is used by host based support code included in the SBC6x Development Package, and should be left disconnected in embedded applications, which are not using II-supplied utilities or which do not have a need to asynchronously control processor reset.

The 16C750 external reset signal is automatically asserted on power up and may be reasserted under software control via the SERIAL_RESET bit of the miscellaneous control register. Refer to the section below for additional details.

USB Port

A Universal Serial Bus (USB) compatible host interface port is included on the SBC6x to support high data rate host communications requirements. Implemented using a National Semiconductor USBN9602 bus interface device, the port allows for communications on a USB network at up to 12 Mbits/sec (USB full speed rate).

The USB interface is accessed starting at address 0x0420000. The following table gives the USBN9602 device register mappings.

USB9602 Register	Access Direction	Address
DATA_OUT	R	0x420000
DATA_IN	W	0x420000
Reserved	R	0x420001
ADDRESS	W	0x420001

TABLE 25. UART Control Registers

The USBN9602 uses an indirect addressing method whereby the address of the device register to be accessed is written to the ADDRESS register. Then a read or write operation performed on the DATA_OUT or DATA_IN registers, respectively, to retrieve or store the requisite data. Software should observe device timing constraints as defined by the USBN9602 data sheet with respect to the amount of time required between the writing of address information and the subsequent write or read operation to the DATA registers.

The SBC6x provides a byte wide interface between the USBN9602 and the 'C6x01 processor. Data read from the device is right justified into the least significant eight bits of the data bus, while the device latches the least significant eight bits of the data bus on each write to the device. The remaining most significant 24 bits are undefined on reads and should be masked off.

The USB interface uses the USBN9602's onchip 3.3V reference supply to power the required pullup on the USB D+ data line to indicate the SBC6x's presence on the bus and its ability to signal at full rate. This supply must be initialized according to the USBN9602 data sheet for proper signaling on the USB.

Please note that the USB interface does not operate at the slow USB data rate of 1.5 Mbits/sec. Host ports and hubs to which the SBC6x are to be interfaced must support the full USB rate in order to communicate with the SBC6x.

USB Interrupts

The USBN9602 supports an interrupt output to the 'C6x01 processor along with a fully programmable interrupt masking system. The flexible interrupt generation hardware allows for fully event driven USB message and state processing.

By design the USB standard is best implemented as an event-driven interface with the processor responding to state changes and packet reception by enabling selected interrupts from within the USBN9602's interrupt control system. The USBN9602 data sheet contains a complete description of the relevant registers used to control the masking of individual interrupts and the required handshake responses once an interrupt is received from the interface.

The SBC6x Development Package contains complete source code for the USB interrupt handler.

To receive an interrupt from the USBN9602, it is necessary to properly configure the SBC6x interrupt control logic such that the interrupt output from the USB interface is connected to the desired interrupt input pin on the 'C6x01 processor. For more information on programming the interrupt control logic on the SBC6x, refer to the Interrupts section.

DMA Support and the USB interface

In addition to supporting interrupt driven state reporting, the USBN9602 provides a second output pin dedicated to synchronizing DMA transfers to the device independent of the standard interrupt output. Paired with a special DMA acknowledge memory mapped register, this interface allows processor software to manage hardware synchronized data transfers to and from the USB interface.

The USBN9602 DMA request pin is available as another input source to the SBC6x interrupt control logic, allowing it to be connected to one of the 'C6x01's interrupt input pins. The DMA request line then acts as an interrupt source for the processor, which typically would be used as the synchronization event in 'C6x01 DMA programming. Configured in this way, each pulse on the DMA request pin can cause a read or write transfer to/from the USBN9602.

The target of the DMA transfer is the USB DMA acknowledge register located at address 0x430000. Accesses to this register cause a pulse to be generated at the USBN9602's DMA acknowledge pin, transferring information to and from the device via its DMA interface and bypassing the normal ADDRESS/DATA register transaction. This method streamlines data transfers and reduces the amount of CPU overhead.

When DMA transactions are to be performed between the processor and the USB interface, one end-point address should always be the DMA acknowledge register (used as the source address for reads and the target address for writes). While the DMA should be synchronized to the USB DMA request signal, programmed to be connected to one of the processor external interrupt inputs. The other end-point address can exist anywhere in processor memory, although it will most likely be in either on-chip

or external (if populated) RAM. Bear in mind when performing DMA accesses that the USB interface is only 8-bit, so the DMA controller should be programmed appropriately depending on how the resultant data is to be handled (i.e. whether or not the source/destination data is packed as bytes into 32-bit words or is stored as single bytes in each 32-bit word).

USB Software Development

The SBC6x Development Package contains extensive driver support for the USB environment. Users are strongly encouraged to make use of the standard software drivers in order to speed product development. For more information regarding USBN9602 programming and the USB standard, see the USBN9602 data sheet and the Universal Serial Bus specification.

Timers

The SBC6x provides a total of three channels of independent on board timebase generation for use in timing data acquisition, servo controls, real-time counters, and many other applications. The functionality is divided into two devices: two 32-bit timer channels on the 'C6x01 processor and a 32-bit direct digital synthesizer (DDS) channel in the AD9850 device. This section discusses the AD9850 synthesizer in detail: for more information on the on-chip timers, see the *TMS320C6x01 Peripherals Reference Guide*.

On-chip Timers

The on-chip timers are available for use as software timebases and interrupt generators. The timer I/O signals (one input and one output for each timer channel) are available on the FIFOPort connector (JP23). The signals are buffered through LVT family drivers, and allows external clock sources to drive the timer inputs as well as giving external hardware access to the timer outputs. The timer pins may also be programmed for digital bit I/O operations in applications, which require as many as four bits of digital I/O independent of the other resources on the SBC6x.

In addition to allowing the timer input pins to be driven by external clocks, the SBC6x allows the AD9850 DDS device to directly drive the TINx processor timer inputs. The timer clock source register at address 0x4F0000 controls independent switches in the onboard logic which allow the DDS output signal to be switched on to the timer input pins. The register bit definition is given below.

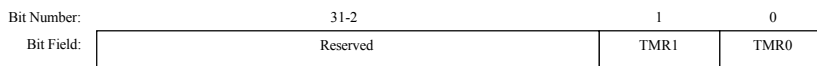


FIGURE 38. On-chip Timer Clock Source Control Register

Bit Field Name	Function
TMR0	Timer Channel 0 Input Clock Source (1 = DDS enabled, 0 = DDS disabled)
TMR1	Timer Channel 1 Input Clock Source (1 = DDS enabled, 0 = DDS disabled)

TABLE 26. On-chip Timer Clock Source Control Register Definition

When set low, the individual bits allow an external signal to be connected to the TINx pins on the FIFOPort connector. When set high, the DDS output signal is switched onto the TINx pins and acts as a source clock for the internal timer(s). The default reset state for both bits are low.

Note: Do not enable the DDS clock when an external signal is present on the TINx pins. Doing so can cause contention between the external signal source and the onboard logic, this may result in damage to the SBC6x, or the external hardware, or both.

AD9850 Direct Digital Synthesizer

The AD9850 direct digital synthesizer (DDS) is a precision programmable clock source which is capable of generating frequencies in the range of 0 to 25 MHz with a resolution of 0.019 Hz/step. Unlike a digital counter-timer chip, which uses a digital counter to divide down a high input clock rate, the DDS uses phase-locked-loop synthesizer technology to tune a sine wave oscillator based on a 32-bit digital word. This method realizes a linear output frequency over input range rather than the nonlinear one associated with counter-timer chips, whose resolution drops dramatically as the period register used to program them falls. The DDS should be used when a precise and accurate clock is required by the application.

The AD9850 is mapped into memory as shown in the table below. The device is interfaced using the parallel I/O method, with one address to write data, one to trigger frequency/phase updates, and one to control the reset pin of the device.

Function	I/O Space Address
AD9850 Reset	0x470000
AD9850 Frequency Update	0x480000
AD9850 Write Clock	0x490000

TABLE 27. AD9850 Control Registers

The write clock address latches frequency/phase data into the AD9850 one byte at a time. The least significant eight bits of the processor bus carry the byte-wide data. The frequency update address causes the output frequency and phase of the DDS clock to update to the values contained in its input latches. The reset address causes an active high reset pulse to be generated to the AD9850. All three of these registers are write-only.

The SBC6x Development Package includes support routines, which make it easy to set the AD9850's output frequency as discussed in the previous sections of this manual.

Digital I/O

The SBC6x includes 32 bits of software programmable digital I/O for use in controlling digital instruments or acquiring digital inputs. The digital I/O port controls are mapped into memory space using three addresses: one to read/write the digital I/O data as a single 32-bit word, one for direction control for each byte of the port, and one for controlling the source of the clock edge used to latch input data into the digital I/O port register. The following table lists the addresses and their functions.

Function	Address
Digital I/O Data Register	0x4A0000
Digital I/O Direction Control	0x4B0000
Digital I/O Input Latch Clock Control Register	0x4C0000

TABLE 28. Digital I/O Control Registers

The direction control register provides for software control of the drive direction of the port. The least significant four bits of the register control the four bytes available on the I/O port. Bit D0 sets the direction for the least significant eight bits of the port (port bits 0-7), D1 the next least significant bits (8-15), D2 the next least significant (16-23) and D3 the most significant (24-31). Each byte is individually controllable by writing a zero (to select output) or a one (to select input) to the respective bit in the direction control register. For example, if the value 0xC were written to the direction control register, bits 0-15 would act as inputs while bits 16-31 would act as outputs. All bytes default to input mode upon the board power-up or on reset.

The data register allows software to directly read data from port pins programmed for input, or write data to pins programmed for output. Read operations performed from the data register on port bytes programmed for output will return the current value of the digital I/O latch (i.e. the last value written to that portion of the port). For example, suppose that the direction controls were programmed to 0xC and the data register written with the data word 0x12340000. Since the most significant 16 bits are setup as outputs, those pins on the port connector would assume the value 0x1234. A subsequent read of the port would yield the value 0x1234xxxx, where xxxx is the value of the signals present on the digital I/O connector.

The input latch clock register allows the user to select from either software read clocking or external hardware clocking. Writing a zero to the register selects software clocking, while writing a one selects external hardware clocking. If software clocking is selected, then the port latches programmed for input will clock in the digital data present on the external pins at the beginning of a read cycle executed on the port data register (30-50 ns before the data is returned to the processor, depending on processor clock speed). If external clocking is selected, then the port will latch data on the falling edge of the TTL signal EXT_DIG_RD_CLK* on the digital I/O connector. The data will be held for the processor to read until the next low-going edge of the EXT_DIG_RD_CLK* signal. In the external hardware clocking mode, read operations by the processor do not affect the contents of the digital I/O latch. The latched data may be reread as many times as is required, and only another EXT_DIG_RD_CLK* pulse will cause new data to be latched into the port.

Digital I/O Timing

The following diagram gives timing information for the digital I/O port when used in external readback clock mode (see above for details). This data is derived from device specifications and is not factory tested.

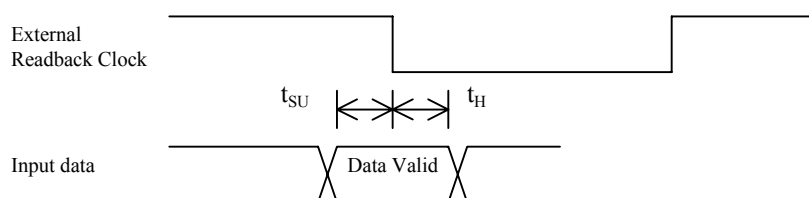


FIGURE 39. Digital I/O Port Timing

Parameter	min (ns)
t_{SU}	0
t_H	10

TABLE 29. Digital I/O Port Timing Parameters

External Mux Control

The SBC6x provides two external multiplexer control bus connectors for use with the TERM external multiplexer board. Control for the multiplexer connectors is provided at the MUX0 and MUX1 addresses (see memory map above), with the memory-mapped functions described in the following table.

TERM Module	Function	I/O Space Address
0	Mux #0 Channel Select	0x4D0000
	Mux #1 Channel Select	0x4D0004
	Mux #2 Channel Select	0x4D0008
	Mux #3 Channel Select	0x4D000C
	All Muxes Channel Select	0x4D0010
	Reset	0x4D001C
1	Mux #0 Channel Select	0x4E0000
	Mux #1 Channel Select	0x4E0004
	Mux #2 Channel Select	0x4E0008
	Mux #3 Channel Select	0x4E000C
	All Muxes Channel Select	0x4E0010
	Reset	0x4E001C

TABLE 30. External Mux Control Memory Map

The control connectors (JP20 for TERM module #0 and JP21 for TERM module #1) select multiplexer channel numbers. The first four addresses from the start of each mux control address map allow the selection of incoming signals on each multiplexer device on the TERM hardware. The fifth address location allows the simultaneous selection of the same channel on all multiplexer devices. The remaining address performs a global reset of the TERM hardware.

Interrupts

The 'C6x01 processor implements four interrupt input pins which allow external hardware events to directly trigger software activity. Processor interrupt inputs are supported on the SBC6x through a set of control registers and multiplexers which allows application software to dynamically select the source of the signal which will drive each particular interrupt input.

The available interrupt source signals are as follows:

1. External interrupt input pins 0-3 (from the I/O modules).
2. 9850 direct digital synthesizer clock.
3. Serial interrupt.
4. USB interrupt and DMA request.
5. FIFOPort FIFO level status.

The following table shows the addresses of the control registers for each processor interrupt input. A value written to the appropriate control register causes the interrupt mux to select the interrupt source given in the next table (see below).

Function	Address
External Interrupt Input 4 Select	0x500000
External Interrupt Input 5 Select	0x510000
External Interrupt Input 6 Select	0x520000
External Interrupt Input 7 Select	0x530000

TABLE 31. External Interrupt Input Control Registers

Interrupt Control Register Value	Interrupt Source
0	External Interrupt Input 0
1	External Interrupt Input 1
2	External Interrupt Input 2

3	External Interrupt Input 3
4	9850 DDS Clock
5	Serial Interrupt
6	USB Interrupt
7	USB DMA Request
8	FIFOPort Receive FIFO Full
9	FIFOPort Receive FIFO Empty
10-14	Reserved (do not use)
15	Deactivated (interrupt held high)

TABLE 32. External Interrupt Input 4 Source Select Register Values

Interrupt Control Register Value	Interrupt Source
0	External Interrupt Input 0
1	External Interrupt Input 1
2	External Interrupt Input 2
3	External Interrupt Input 3
4	9850 DDS Clock
5	Serial Interrupt
6	USB Interrupt
7	USB DMA Request
8	FIFOPort Receive FIFO Almost Full
9	FIFOPort Transmit FIFO Full
10-14	Reserved (do not use)
15	Deactivated (interrupt held high)

TABLE 33. External Interrupt Input 5 Source Select Register Values

Interrupt Control Register Value	Interrupt Source
0	External Interrupt Input 0
1	External Interrupt Input 1
2	External Interrupt Input 2
3	External Interrupt Input 3
4	9850 DDS Clock
5	Serial Interrupt
6	USB Interrupt
7	USB DMA Request
8	FIFOPort Transmit FIFO Empty
9	FIFOPort Transmit FIFO Almost Full
10-14	Reserved (do not use)
15	Deactivated (interrupt held high)

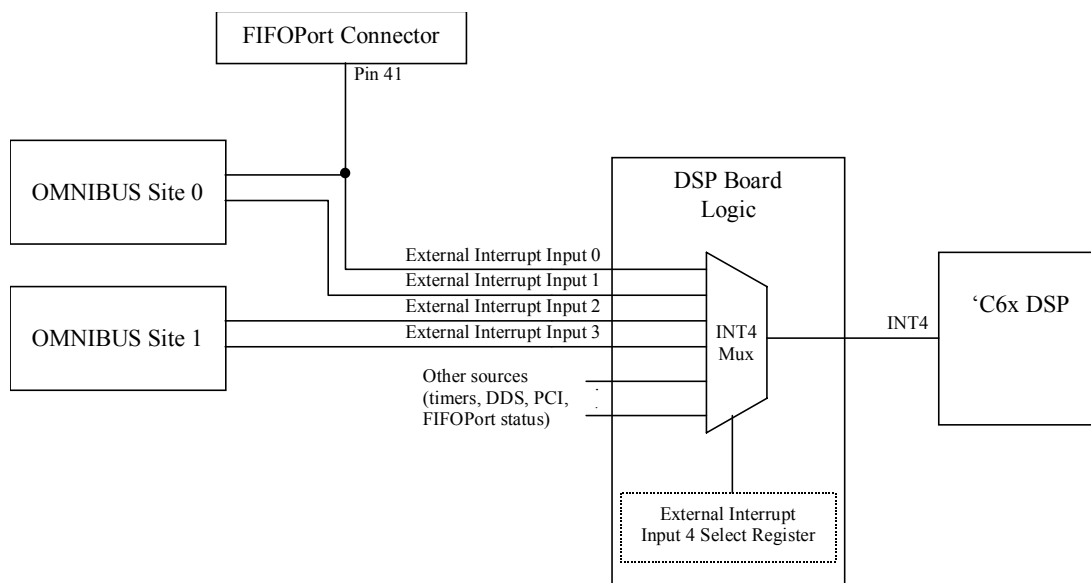
TABLE 34. External Interrupt Input 6 Source Select Register Values

Interrupt Control Register Value	Interrupt Source
0	External Interrupt Input 0
1	External Interrupt Input 1
2	External Interrupt Input 2
3	External Interrupt Input 3
4	9850 DDS Clock
5	Serial Interrupt
6	USB Interrupt
7	USB DMA Request
8-14	Reserved (do not use)
15	Deactivated (interrupt held high)

TABLE 35. External Interrupt Input 7 Source Select Register Values

For example, if the application requires the output from external interrupt input two to drive processor interrupt input five, the value two should be written to memory location 0x510000. All interrupt control registers default to 15 on power-up or board reset. Note that the processor interrupt signals generated by the logic are active low (falling edge trigger), and the ‘C6x01 interrupt polarity control register must be programmed to the value 0xF to correctly receive interrupts.

The following diagram illustrates the flow of interrupt signals from the various source locations through the logic to the processor. The diagram shows the handling circuitry for the INT4 signal only; the other three interrupt inputs are handled in similar fashion.



The diagram shows the routing of the external interrupt input signals from the OMNIBUS and FIFOPort connectors through the INT4 multiplexer to the INT4 signal pin on the 'C6x DSP. The External Interrupt Input 4 Select Register controls operation of the INT4 mux and causes it to select one of the possible input source signals and drive the signal to the INT4 pin on the DSP. For clarity, the other possible source inputs to the INT4 mux have been abbreviated.

Note that the diagram shows the connection of the External Interrupt Input 0 signal to the FIFOPort at pin 41. This connection allows the signal to be driven by offboard hardware, allowing users to send a single interrupt signal to the DSP via the INT4 mux. Note also that since the connection is a direct short to the OMNIBUS site 0 interrupt signal pin, the FIFOPort may not have a signal connected to it when there is also a module installed in OMNIBUS site 0 which is actively driving the External Interrupt Input 0 node. Since most modules make use of this interrupt signal to notify DSP software of real-time events, it is generally not possible to install an OMNIBUS module in site 0 and connect a signal to FIFOPort pin 41. In the case where a customer needs to install a module in OMNIBUS site 0 and also needs to connect an external interrupt signal to the DSP board, Innovative recommends the use of the FIFOPort transmit status signal pins on the FIFOPort, since these may be programmed as interrupt sources to the DSP (see the FIFOPort section above for a discussion of the transmit status pins). Use care when attempting to connect an external signal to pin 41 of the FIFOPort connector, as an electrical conflict between the external signal driver and an installed OMNIBUS module may cause damage to the external hardware, the OMNIBUS module, or both.

JTAG Test Bus

The SBC6x implements a JTAG 1149.1-compatible scan path loop through the onboard 'C6x01, with connector compatible with the specification provided in the *TMS320C6x01 User's Guide*. When connecting a JTAG controller card cable (from an Innovative Integration Code Hammer debugger card, Texas Instruments XDS-510, or other vendor's JTAG hardware), connector JP8 is used.

Miscellaneous Control

The miscellaneous control register implements several write-only control bits which control various features on the card. The following diagram and bit description give the definition of the register.

Bit Number:	31-4	3	2	1	0
Bit Field:	Reserved	SERIAL_RESET	MOD_PWR	RDY_RST	CLK_DBL

FIGURE 40. Miscellaneous Control Register

Bit Field Name	Function
CLK_DBL	0 = OMNIBUS clock normal mode, 1 = OMNIBUS clock speed doubled.
RDY_RST	0 = ARDY wait state generator enabled, 1 = ARDY wait state generator disabled.
MOD_PWR	Bit output to MOD_PWR pin on power connector.
SERIAL_RESET	0 = 16C750 reset deasserted, 1 = 16C750 reset asserted.

TABLE 36. Miscellaneous Control Register Definition

The CLK_DBL and RDY_RST bits are intended for Innovative test use only.

The MOD_PWR bit may be used by external power supplies as a control bit for determining low power state usage. The bit value written into the register is latched and output directly from the onboard logic to the JP22 power connector

The SERIAL_RESET bit is used to assert a hardware reset signal to the 16C750 UART device. Writing a value of one asserts the reset signal (i.e. places the 16C750 in reset), while writing a value of zero deasserts the signal to the 16C750.

Power Requirements

The SBC6x processor board requires an off-board power supply capable of delivering three power supplies: +5V DC, +15V DC, and -15V DC. Power is delivered to the card via the main power connector. Pinouts for the power connector are given in the appendices. Innovative Integration supplies linear power supplies suitable for use with AC mains powered installations (part number 80022-6 for US 120 volt input: call for information concerning European and Asian power supplies).

Caution: double-check all power wiring before attempting to start the board with custom power supplies or connections. Incorrect power supply connections will severely damage the SBC6x board. The board's hardware warranty is void if specified power connection requirements are not met by the end user.

Any user-procured power supply hardware should meet the minimum current delivery capabilities supplied by the standard power supply. The standard supply delivers +5V 2A and +-15V 400 mA. We recommend doubling these current capabilities if high power OMNIBUS modules (SD, RF, HSA) are intended for use. All three supplies should start within 30 ms or so of each other, 5V starting first. +-15V supplies should be as quiet as possible (< 20 mV p-p ripple at frequencies < 100 kHz) as any noise can affect performance of analog hardware on OMNIBUS modules. Since power requirements can be affected drastically by software construction and applications-specific details, we recommend performing current measurements using a bench supply while running the actual application to verify actual power supply requirements.

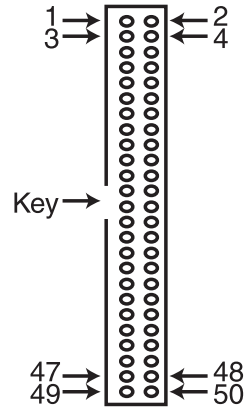
Connector pinouts

JP11, JP12, JP16, JP17 - OMNIBUS I/O Connectors

Connector types:	JP11, JP16: AMP .05 Subminiature D male JP12, JP17: .100" header
Number of pins:	JP11, JP16: 50 JP12, JP17: 50
Mating connector:	JP11, JP16: AMP 173279-3 JP12, JP17: AMP 1-746285-0

The following table shows the interconnections between the JP11 (OMNIBUS slot 0) and JP16 (OMNIBUS slot 1) module I/O connectors and their respective external I/O connectors, JP12 (for OMNIBUS slot 0) and JP17 (for OMNIBUS slot 1).

JP12, JP17 Pin Numbers
1-35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50



JP11, JP16 Pin Numbers
1-35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50

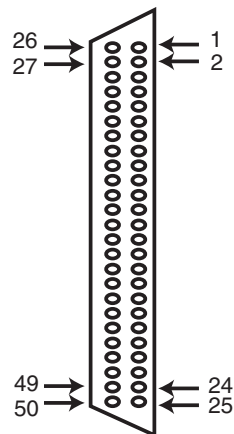


TABLE 37. OMNIBUS Connector Pinouts

JP13, 18, 14, 19 – OMNIBUS Bus Connectors

Connector types:	AMP .05 Subminiature D male
Number of pins:	50
Mating connector:	AMP 173279-3

The following table gives the pin numbers and functions for the JP13 (OMNIBUS slot 0) and JP18 (OMNIBUS slot 1) connectors. The functions for JP18 are identical to those of JP13, except where noted.

Pin Number	JP13 Function	JP18 Function	Direction (from SBC6x)
1, 19	Digital +5V		O, power
2, 20	Digital ground		O, power
3-18	Data bus 0-15		I/O
21, 43, 40, 45, 39, 26, 27	Address bus 2-8		O
28	Reset (active low)		O
29	External interrupt 0	External interrupt 2	I
30	Bus ready (active low)		I (open-collector)
31	Processor CLKOUT2 / 4		O
32	Processor timer channel 0		O
33	R/W*		O
34	9850 timebase		O
35-38	IOMOD0-3 decoded selects (active low)	IOMOD4-7 decoded selects (active low)	O
25	Analog -15V (OMNIBUS -12V)		O, power
23	Analog +15V (OMNIBUS +12V)		O, power
41,42	Analog ground		O, power
22,24	Analog -15V		O, power
44,46	Analog +15V		O, power
47,49	Analog +5V		O, power
48,50	Analog -5V		O, power

TABLE 38. OMNIBUS Bus Connectors

The following table gives the pin numbers and functions for the JP14 (OMNIBUS slot 0) and JP19 (OMNIBUS slot 1) connectors. The functions for JP19 are identical to those of JP14, except where noted.

Pin Number	JP14 Function	JP19 Function	Direction (from SBC6x)
1, 3-6	Address bus 9-13		O
2, 19, 20, 49, 50	Digital ground		O, power
7-18	Reserved	Reserved	NA
21	Processor timer channel 1		O
22	External trigger 0	External trigger 1	O
23,25	Analog +15V (OMNIBUS +12V)		O, power
24	CLKS0	CLKS1	I
26	CLKR0	CLKR1	I/O
27	FSR0	FSR1	I/O
28	CLKX0	CLKX1	I/O
29	External interrupt 1	External interrupt 3	I
30	DR0	DR1	I
31	FSX0	FSX1	I/O
32	DX0	DX1	O
33-48	Data bus 16-31		I/O

TABLE 39. I/O Module Bus Connectors

JP1 – Digital I/O Connector

Connector type:	0.1” double row shrouded header
Number of pins:	40
Mating connector:	AMP 746285-9

The following table gives the pin numbers and functions for the JP1 connector.

Pin Number	JP1 Function	Direction (from SBC6x)
1-32	Digital I/O bit 0..31	I/O
33	External Trigger 0 Input (active low)	I
34	9850 DDS Clock Output	O
35	External Trigger 1 Input (active low)	I
36	Reserved	NA
37	External Digital Readback Clock (active low)	I
38	External Interrupt Input 0	I
39	Digital +5V	Power
40	Digital Ground	Power

TABLE 40. Digital I/O Connector

JP23 – FIFOPort Connector

Connector type: 2mm header
 Number of pins: 54
 Mating connector: Samtec SQW style

The following table gives the pin numbers and functions for the JP23 connector.

Pin Number	JP23 Function	Direction (from SBC6x)
1	Digital 5V	Power
2, 44	Ground	Power
3-18	Input Data Bits 15-0	I
19	Reserved	NA
20	Input Strobe	I
21	On-chip Timer 1 Out	O
22	On-chip Timer 1 In	I
23	On-chip Timer 0 Out	O
24	On-chip Timer 0 In	I
25-40	Output Data Bits 0-15	O
41	External Interrupt Input 3	I
42	Output Strobe	O
43	Digital 3.3V	Power
45	Half-full Flag Out	O
46	Almost-full Flag Out	O
47	Almost-full Level Control In	I
48	Full Flag Out	O
49	Almost-full Level Control Out	O
50	Empty Flag Out	O
51	Reserved	NA
52	Almost-full Flag In	I
53	Empty Flag In	I
54	Full Flag In	I

TABLE 41. FIFOPort Connector

JP6, JP7 – Processor Serial Port Connectors

Connector type:	2 mm double row header
Number of pins:	10
Mating connector:	Samtec SQT style (for board-board applications)

The following table gives the pin numbers and functions for the JP6 and JP7 connectors. Pin functions of JP6 are identical to those of JP7 except where noted.

Pin Number	JP6 Function	JP7 Function	Direction (from SBC6x)
1	CLKS0	CLKS1	I
2	FSR0	FSR1	I/O
3	CLKR0	CLKR1	I/O
4	FSX0	FSX1	I/O
5	CLKX0	CLKX1	I/O
6	Digital 3.3V		Power
7	DR0	DR0	I
8	Digital 5V		Power
9	DX0	DX0	O
10	Digital Ground		Power

TABLE 42. Processor Serial Port Connector

JP8 – JTAG Debugger Connector

Connector type:	Shrouded header, pin 6 removed
Number of pins:	14
Mating connector:	AMP 746285-2

The following table gives the pin numbers and functions for the JP8 connector.

Pin Number	JP8 Function	Direction (from SBC6x)
1	TMS	I
2	TRST*	I
3	TDI	I
5	Digital +3V	Power
7	TDO	O
9,11	TCK	I
13	EMU0	I/O
14	EMU1	I/O
4, 6, 8, 10, 12	Digital ground	Power

TABLE 43. JTAG Debugger Connector

JP22 – Power Input Connector

Connector type:	6 pin locking power connector (Molex 43045-0602)
Number of pins:	6
Mating connector:	Molex 43025-0600 and contacts

The following table gives the pin numbers and functions for the JP22 connector.

Pin Number	JP22 Function	Direction (from SBC6x)
1	Digital +5V	Power
2,4	Digital ground	Power
3	Analog +15V	Power
5	Analog -15V	Power
6	MOD_PWR power control	O

TABLE 44. Power Input Connector

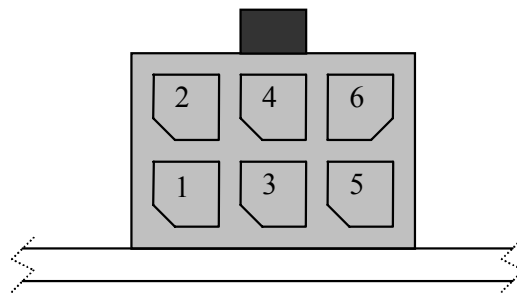


FIGURE 41. Power Connector Pin Positions (side view, from front of connector, showing connector keying and locking tab along with printed circuit board position)

Note: Mating connector may be numbered differently.

JP20, JP21 – External Multiplexer Control Connectors

Connector type: Shrouded header
 Number of pins: 14
 Mating connector: AMP 746285-2

The following table gives the pin numbers and functions for the JP20 and JP21 connectors. JP20 implements multiplexer control port 0, while JP21 implements port 1.

Pin Number	Function	Direction (from SBC6x)
1	Mux D0	O
2	Mux A0	O
3	Mux D1	O
4	Mux A1	O
5	Mux D2	O
6	Mux A2	O
7	Mux strobe (active low)	O
8, 13	Reserved	NA
9	Digital +5V	Power
10	Digital ground	Power
11	Analog +15V	Power
12	Analog -15V	Power
14	Analog ground	Power

TABLE 45. Multiplexer Control Connector

JP3 – Asynchronous Serial Port Connector

Connector type:	Shrouded header
Number of pins:	10
Mating connector:	AMP 746285-1

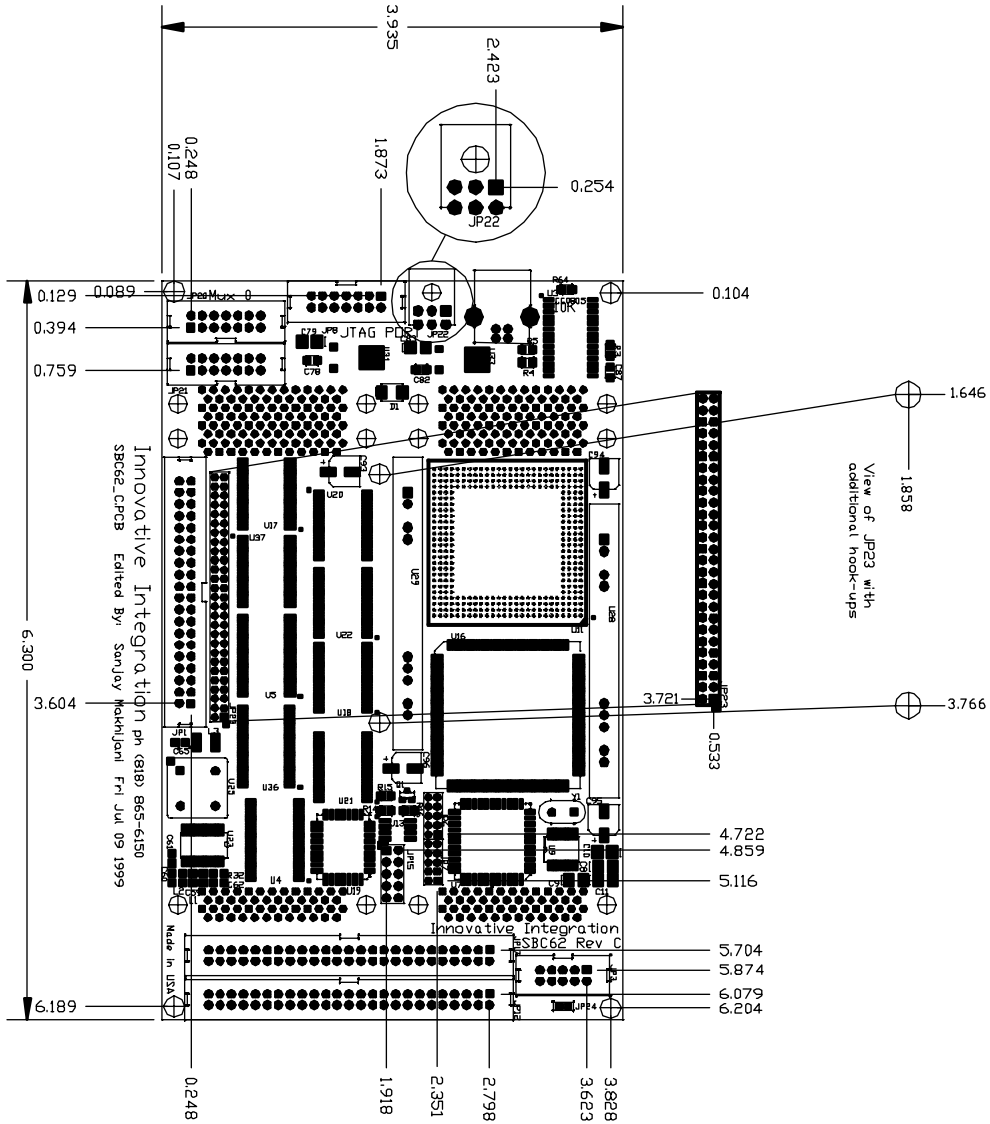
The following table gives the pin numbers and functions for the JP3 connector.

Pin Number	Function	Direction (from SBC6x)
1, 2, 8, 10	Reserved	NA
3	RS232 Transmit Data	O
4	RS232 CTS	I
5	RS232 Receive Data	I
6	RS232 RTS	O
7	RS232 DTR (Board Reset)	I
9	Digital ground	Power

TABLE 46. Asynchronous Serial Port Connector

Board Layout

A schematic of the board layout is displayed on the following page. Please review this schematic to familiarize yourself with the circuit board's configuration.



TMS320C6x01 Limitations and Errata

As of this writing, the TMS320C6x01 processor has several limitations and errata that can affect the maximum clock rate at which the processor can successfully run and which may impede the proper operation of certain software applications. This section discusses limitations discovered in Innovative Intergration's testing of early SBC6x prototype cards, as well as errata announced by Texas Instruments regarding the current 2.0 revision silicon.

This information is being supplied to current customers and potential users of the SBC6x in an effort to keep you informed of the state of 'C6x01 processor development and any performance limitations imposed on the SBC6x hardware design. Innovative Intergration will continuously update this information as new data becomes available and particularly when new silicon revisions are released by Texas Instruments to us for testing.

Processor Speed Limitations and External Memory

The current revision 2.0 silicon 'C6x01 devices have bus timing issues which prohibit the use of full speed external synchronous burst SRAM (SBSRAM) and synchronous DRAM (SDRAM) devices. These limitations affect the maximum processor speed Innovative will be able to ship with current processors, dependent on the external memory configuration of the SBC6x hardware as ordered by the customer.

Specifically, Texas Instruments has announced that SDRAM support is limited to approximately 90MHz operation (180 MHz processor speed), while SBSRAM operation is limited to 133 MHz (133 MHz processor speed). These figures were determined through board level testing at the Texas Instruments facility.

Testing of hardware at Innovative Integration shows that SDRAM is supported at least through 80 MHz (160 MHz processor speed), while SBSRAM has been tested up to a rate of 80 MHz (160 MHz processor speed with SBSRAM in half-rate mode). Testing at a processor speed of 133 MHz with SBSRAMs running in full rate mode has shown a read/write failure rate of about 10-ppm. Texas Instruments has specified that not all SBSRAMs are reliable in their tests, so it is possible that the devices Innovative is currently using may not be up to spec. Innovative is in the process of procuring SBSRAM samples of the manufacturer and type used by Texas Instruments and will continue testing of the external memory interface to determine which devices are more reliable. In addition, Innovative will be procuring additional clock source devices, which will allow the testing of intermediate processor speeds (180 MHz, for example).

Current processors are capable of 200 MHz operation and have been tested at this rate on the SBC6x design. These tests involve running software strictly from on-chip memory. The external peripheral interfaces (I/O bus sites, serial ports, FIFO ports, onboard peripherals, async SRAM, PCI interface) are unaffected by the memory interface issue as they use less aggressive bus timing.

Innovative Integration's policy on processor speed is to deliver the fastest possible speed consistent with the requested external memory configuration on the board. In situations where customers order external memory, Innovative must downgrade the processor speed to match the memory interface limitations. Current processor speeds available versus memory requirements are as follows:

External Synchronous Memory Type	Delivered Processor Speed
None	200 MHz
SDRAM	160 MHz
SBSRAM	160 MHz (SBSRAM operating in half-rate mode)

As Innovative continues testing the memory subsystems of the SBC6x, these rates may change to improve the memory access and processor speeds.

Texas Instruments Device Errata

The current Texas Instruments device errata for the most current revision silicon TMS32C6x01 devices is attached below. At this time Innovative Integration does not consider these errata to be significant to the overall operation of the card design.

Insert Texas Instrument errata files here:

6201SIERAT4

Insert Texas Instrument errata files here:

6701SIERAT1_18_00.PDF



Kane Computing Ltd
7 Theatre Court, London Road,
Northwich, Cheshire, CW9 5HB, UK.
Tel: +44(0)1606 351006
Fax: +44(0)1606 351007/8
Email: sales@kanecomputing.com
Web: www.kanecomputing.co.uk