Using Keil Development Tools with Triscend FastChip and the E5 CSoC Family

January, 2000, v1.02 Application Note (AN-07)

Abstract:

This application note describes how to configure and use Keil's µVision-51 and dScope-51 development software with Triscend's FastChip CSoC Development System to create and debug Triscend E5 applications.

Introduction

Keil's µVision-51 integrated development environment for the 8051 and their dScope simulator/debugger support the Triscend E5 Configurable System-on-Chip (CSoC) device family.

This application describes how to configure and use Keil's µVision-51 and dScope-51 development software with Triscend's FastChip Development System to create and debug Triscend E5 applications. Some of the topics include:

- FastChip's automatic header file generation, including automatic address assignment for registers used in "soft" modules. FastChip also generates initialization routines for the dedicated resources within an E5 CSoC device. FastChip conveys this information to Keil via a header file.
- How to configure Keil's compiler and linker settings for optimal results. These settings are saved within a Keil project.
- The differences between instruction-set simulation (ISS) and in-system debugging.
- How to use the Keil dScope instruction set simulator to verify logical operation.
- How to use dScope as a source-level debugger with the Triscend E5 CSoC.
- What memory spaces and unique data types are available when using Keil C51. Using only ANSI-C constructs limits the amount of optimization possible with the 8051 architecture.
- How to use typed and generic pointers with C51.
- How to declare interrupt service routines in both C51 and A51.
- How to compile programs for the 8051 architecture of up to 2 Mbytes in size. Keil's code-banking compiler and unique features within the Triscend E5 CSoC make it possible.

Communicating with Keil using Header Files

The Triscend FastChip Development System communicates with the Keil tools via a header file. This header file is created while completing a CSoC hardware design, using FastChip's Generate program.

The header file contains the following information.

- Register names and address assignments for all 8032 and Triscend E5 control registers (CRU).
- Register names and automatic assignments for all "soft" module functions that contain memory-mapped registers.
- Initialization routines for all the Triscend E5 dedicated resources such as the 8032 peripherals, DMA channels, etc.

Creating the Header File using the FastChip Generate Program

After completing the E5 hardware design in FastChip, create a header file to use with the Keil tools. FastChip automatically assigns the address values of all memory-mapped soft module registers during the Generate process.

Generate uses the symbolic names defined in the design as the register variable names. Generate then allocates the variables—in SFR or XDATA space as specified in the design—at the FastChip-assigned address location.

To create the header file, click the **Generate** button from the FastChip tool bar.

In the resulting dialog box, click the **C** option to create a 'C' header file or the **Assembly** option to create a header file for assembly-language programs. Choose a location to store the resulting header file. FastChip automatically assigns a '*.h' extension for the 'C' header file or '*.inc' for the assembly header file.

Including the Header File

A 'C' or assembly language program written for the Triscend E5 must include the header file generated by FastChip before referring to any registers in the design. The header file contains the register definitions and initialization routines required by the application.

Including the 'C' Header

To reference the 'C' header file created by FastChip, use the **#include** directive as shown below.

```
// Include the 'C' header file
// created by FastChip

#include "mydesign.h"
```

Including the Assembly-Language Header

The following options are not required for 'C' applications. If the project is written in A51 assembly language, use the $INCLUDE directive instead.

```
; Include the A51 header file
; created by FastChip

$INCLUDE (mydesign.inc)
```

Using the Initialization Routines Created by FastChip

FastChip automatically generates initialization routines for the dedicated resources available on the Triscend E5 CSoC. The top-level initialization routine is always named after the FastChip project name, not the name of the header file. For example, if the FastChip project were called "MyDesign", the top-level initialization routine would be called **MyDesign_INIT( )**.

Table 1. Initialization routines for the E5 dedicated resources, generated automatically by FastChip.

Function	Description
<design>_INIT()	Executes the initialization routines for all dedicated resources, in the indicated order.
Timer_0_INIT()	Timer 0 initialization routine
Timer_1_INIT()	Timer 1 initialization routine
Timer_2_INIT()	Timer 2 initialization routine
UART_INIT()	UART initialization routine
Interrupt_INIT()	Interrupt controller initialization routine
Watchdog_INIT()	Watchdog Timer initialization routine
DMA_0_INIT()	DMA controller, channel 0 initialization routine
DMA_1_INIT()	DMA controller, channel 1 initialization routine
Power_INIT()	Power management initialization routine

Calling the top-level function created by FastChip initializes all of the dedicated resources on the Triscend E5 CSoC, according to the settings specified by the designer during hardware design. The top-level function merely invokes the individual initialization routines, detailed in Table 1.

Calling the 'C' Initialization Routines

```
/***********************************
* MAIN FUNCTION
***********************************/

void main () {

    // Call the main initialization
    // routine defined in the
    // included header file.

    MyDesign_INIT();

}
```

Calling the A51 Initialization Routines

The following options are not required for 'C' applications. If using A51 assembly, call the FastChip-generated initialization routines as a subroutine at the beginning of the application program.

```
      cseg ; absolute segment at 0h

      org 0000h
      ljmp MAIN

; Interrupt routines go here.

; Call the main initialization
; routine from the header file.

MAIN: lcall MyDesign_INIT
```

Other Header File Options

A designer could choose to write his or her own initialization routines instead of using those automatically generated by FastChip. Likewise, the individual initialization routines for the dedicated resources can be invoked as stand-alone functions. For example, invoking UART initialization routine, `UART_INIT( );`, initializes just the UART and nothing else. All of the individual initialization routines are defined in the header file, including comments that document which registers are modified.

Using the Header File in Other Modules

Many software projects contain multiple modules. The FastChip header file defines the initialization routines for the E5's dedicated resources. The main module exclusively uses these functions and consequently the **PROTOTYPE_ONLY** compile-time option is left undefined. Other modules set **PROTOTYPE_ONLY** to avoid redeclaring these functions.

For 'C' modules, the **PROTOTYPE_ONLY** option is defined as shown below.

```
// 'C' modules, other than main,
// should set the PROTOTYPE variable

#define PROTOTYPE_ONLY
#include "mydesign.h"
```

For A51 modules, the **PROTOTYPE_ONLY** option is set as shown below.

```
// A51 modules, other than main,
// should set the PROTOTYPE variable

$SET (PROTOTYPE_ONLY)
$INCLUDE (mydesign.inc)
```

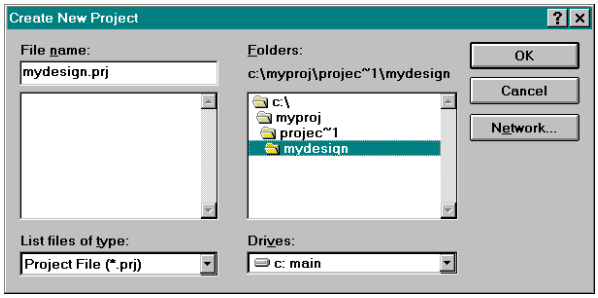
Configuring the Keil Compiler/Linker

After writing the application in Keil 'C', configure the Keil compiler and linker options to produce better code for the Triscend E5 family.

Creating a Project

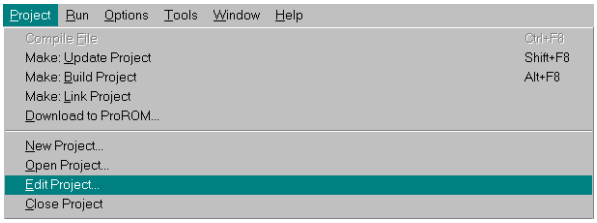
The compiler and linker options are saved as part of the Keil project. Before setting the compiler options, create a new project and add the 'C' source files.

From the **Project** menu, click **New Project**. Select the location of your project registry and enter a name for your Keil project.

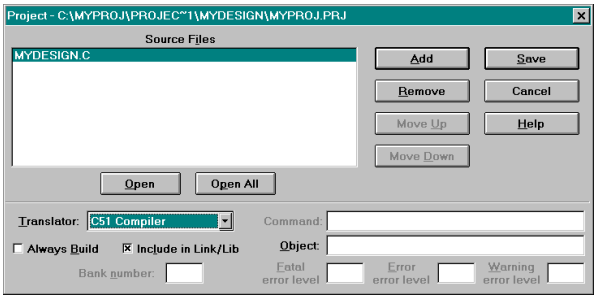


Click **OK** when complete.

To add 'C' source files to the project, select **Project** and then **Edit Project** from the menu.

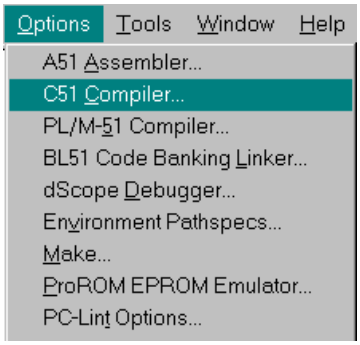


Once the dialog box appears, select the source files and click **Add**. Click **Save** when finished.



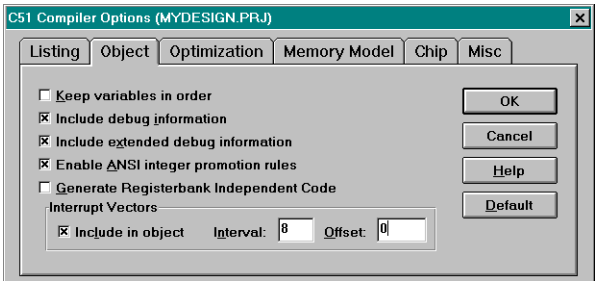
Setting the 'C' Compiler Options

To set options for the 'C' compiler, click the **Options** menu item and select **C51 Compiler...**.



Object File Options

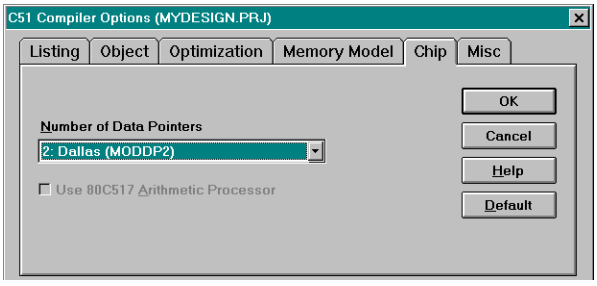
Including debugging information in the object file simplifies debugging when using Keil's dScope instruction set simulator/debugger. To set options for the object file, select the **Object** tab.



For improved debugging visibility, choose the **Include debug information** and the **Include extended debug information** options. Do not click the OK button just yet.

Chip Options

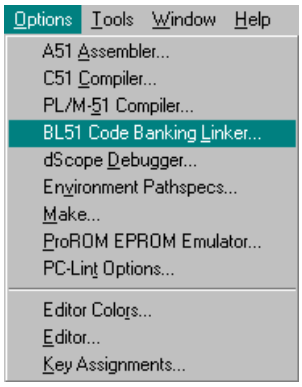
To set options for device-specific compiler optimization, choose the **Chip** tab from the dialog box. The 8032 "Turbo" microcontroller embedded within each Triscend E5 CSoC device has two data pointers, similar to the Dallas 80C320 devices. Choose **2: Dallas (MODDP2)** from the selection drop list.



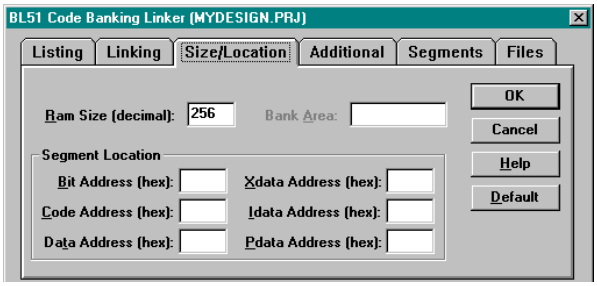
Click **OK** when finished.

Linker Options

The 8032 "Turbo" microcontroller has 256 bytes of RAM, just like other 8052/8032 derivatives. By default, the Keil linker assumes only 128 bytes. To change these setting, select the **Options** menu item, then **BL51 Code Banking Linker...**.



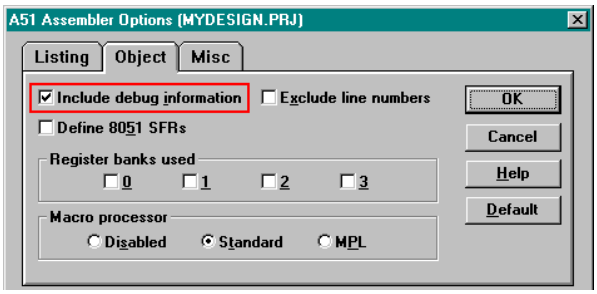
In the resulting dialog box, click the **Size/Location** tab. Update the **Ram Size (decimal):** setting from 128 to **256**. Click **OK** when finished.



A51 Assembler Options

The following options are not required for 'C' applications. Though this application note is written primarily for designers using Keil's 'C' compiler, there are a few Keil options when using assembler.

From **Options** menu, select **A51 Assembler...**. From the resulting dialog box, click the **Object** tab and clear the **Define 8051 SFRs** option.

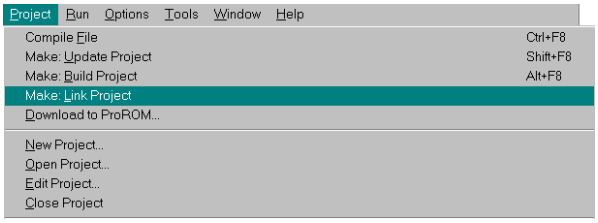


This prevents Keil from inserting SFR definitions that might conflict with the ones from the FastChip generated header file.

Also, click **Include debug information** for improved visibility during instruction-set simulation and debugging.

Compiling the 'C' Program

To compile the 'C' program, select **Project** then **Make: Link Project**. This step compiles and links your 'C' program, and creates the .Hex file and associated object files.



Alternatively, click the **Build All** button from the tool bar.

Potential Pathname Problems

The currently available version of Keil, including the version on the Triscend FastChip CD-ROM, is a Windows 3.1 application. Consequently, the Keil software has problems with long path names, specifically those containing space characters.

Keil issues the following error during compilation if the project files are located in the FastChip project directory.

```
FATAL ERROR 202: INVALID COMMAND
LINE, TOKEN TOO LONG.
```

The problem occurs because the full path name to a 'C' source file saved in the default FastChip project directory contains a space, which is an illegal character for Windows 3.1 programs.

Fortunately, there is a work-around to this problem.

1. Move the Keil source files (*.c, *.asm, *.h, *inc) to another directory, where the path name no longer contains a space.
2. Modify the Keil project file to reflect the directory changes.

Table 2. Comparing Instruction-Set Simulation and In-System Debugging.

	Instruction-Set Simulation	In-System Debugging
Best verification use	During software development, verify logical operation	During hardware/ software integration
Observe program flow, set breakpoints, set register values	✓	✓
Access all memory-mapped resources	✓	✓
Requires working hardware		✓
Easy access to 8032's peripherals	✓	
No extra hardware required, PC only	✓	
Soft modules function beyond any memory-mapped control or status registers		✓
Provides true, accurate model of device		✓

Instruction-Set Simulation versus In-System Debugging

The Keil dScope interface provides two potential views for validating application code.

- **Instruction-set simulation (ISS)** offers full logical debugging of an application, without requiring physical hardware. The application code executes on a model of the 8032, simulated on the PC.
- **In-system debugging** provides true hardware/software integration and testing. The application code is integrated with real physical hardware and code executes in the actual target environment. In-system debugging requires the JTAG download cable, connected between the PC's parallel printer port and the dedicated JTAG pins on the Triscend E5 device.

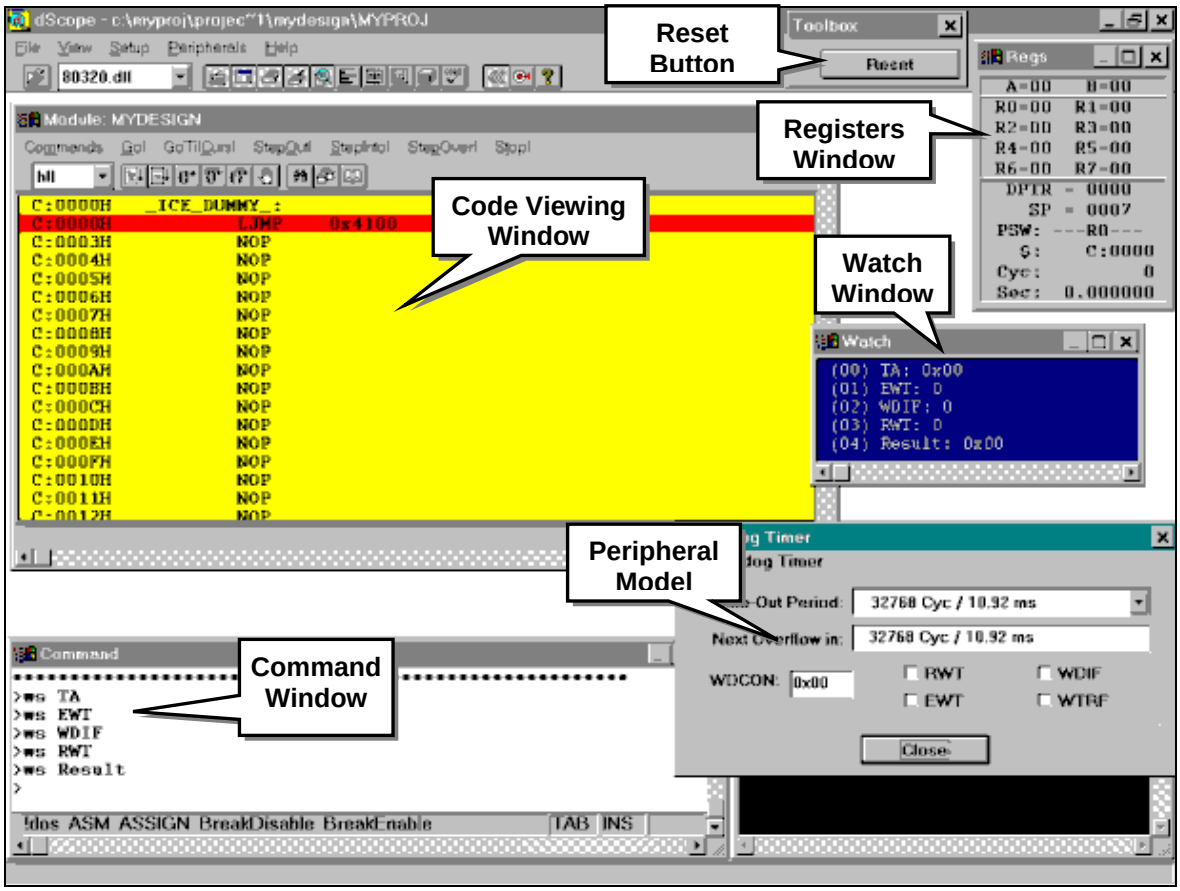


Figure 1. Keil dScope instruction-set simulator and debugger interface.

Table 2 outlines the major differences between instruction-set simulation and in-system debugging. Simulation is easier during code development because no working hardware is required. Debugging is more useful, once hardware is available, to eliminate hardware/software integration problems.

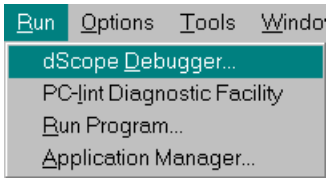
With older 8051 devices, in-system debugging was performed by attaching an in-circuit emulator (ICE) to the target board. This often proved expensive and clumsy. The E5 has built-in debugging hardware that provides superior debugging capabilities via four dedicated JTAG pins on the device, connected to a PC operating as the debugging host via the JTAG download cable.

Simulating an Application using dScope

The Keil 8051 development tools include an instruction-set simulator called dScope. The dScope program does not simulate the functionality of any "soft modules" implemented in the E5's CSL logic. However, all memory-mapped registers are visible.

Starting the Debugging Session

To launch dScope from inside µVision, click **Run** and then **dScope Debugger...**. A separate window appears for dScope.



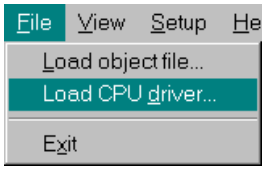
Various windows, shown in Figure 1, display the status of the application and help during debugging. Occasionally, not all of dScope's windows appear on the screen. To display all dScope windows, right-click the mouse button on the dScope task in the Windows task bar. Then, click **Maximize**.

Configuring dScope

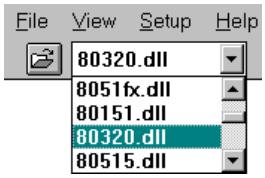
The dScope program requires two pieces of information to accurately simulate application code.

1. A functional model of the specific 8051 variant used in the design.
2. The object file (OMF) containing the code, variable names, and other relevant information from the application code.

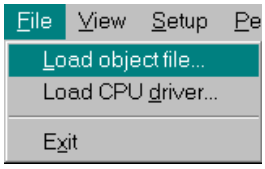
To load the functional model, click **File** and then **Load CPU driver...** from the topmost set of menus. A list of available CPU models appears.



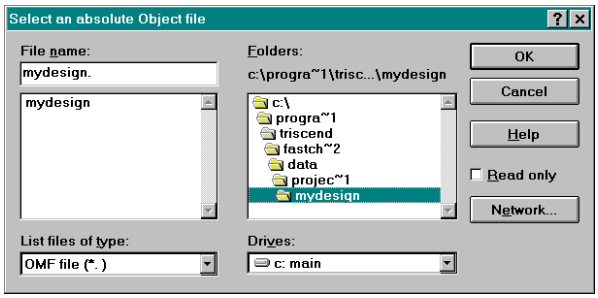
Currently, there is no exact model for the Triscend E5 "Turbo" microcontroller. However, the model for the Dallas 80C320 is sufficiently similar for most applications. Scroll through the list and select the **80320.dll** CPU driver. The differences between a Dallas 80C320 and the E5's embedded processor are few. The Dallas device contains a second dedicated UART. Furthermore, some instructions on the Triscend E5 execute faster than the Dallas device. However, for validating logical execution of code, the Dallas model is sufficient.



After loading the processor model, load the object file created when the application program was compiled. From the topmost set of menus, select **File** and then **Load object file...**.



Select the object file corresponding to the application program from the dialog box.



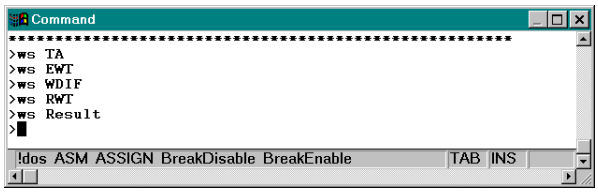
Click **OK** when finished.

Observing Register Values

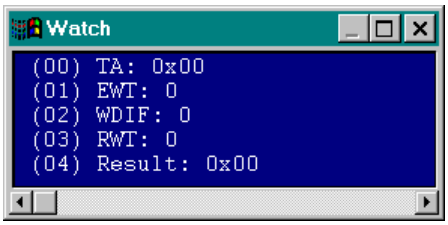
With the object file loaded, you can specify variables that dScope should watch. Any register declared in the header file can be watched, either as a byte-wide or bit-wide entity. Likewise, any variable declared with global scope is visible.

To watch a variable, use the "Command" window to type in a "watch set" command.

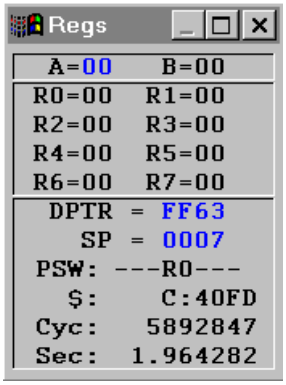
```
ws <variable name>↵
```



The "watched" values appear in the Watch window, shown below.



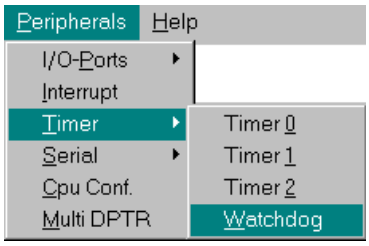
Many of the 8032's internal registers are visible in the "Regs" window, including the Data Pointer (DPTR), Accumulator (A), and the current register bank (R0-R7). Register values that changed are highlighted in blue. The "Regs" window typically only updates after the processor halts execution, reaches a breakpoint, or is single-stepped.



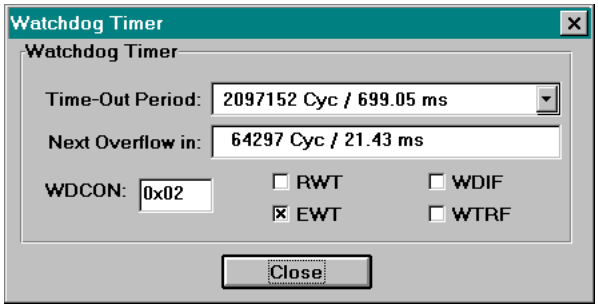
Observing Peripherals

Keil's dScope instruction-set simulator provides views into the internal operation of the microcontroller's peripherals. The correct peripheral simulation models only appear if the **80320.dll** driver is loaded. The 80320.dll model is designed for the Dallas 80C320. The Triscend E5 is sufficiently similar, though the E5 does not have a second, dedicated UART. These peripheral models are not available in the debugger interface driver, **te5-8032.dll**, used during in-system debugging.

The following example demonstrates how to observe the operation of the Watchdog Timer. From the topmost menus, select **Peripherals**, then **Timer**, then **Watchdog**.



The resulting window shows the current register settings, flag values, and overall state of the Watchdog Timer. These values can also be modified during simulation.



Observing Code Flow

The code window displays the application code and indicates the current program state. There are three possible code views to display.

1. **HLL** or High-Level Language, which usually is the 'C' source file, or the original assembly language listing if using assembly.
2. **ASM** or Assembly language, which shows the resulting compiled or assembled application code.
3. **MIXED**, indicating a mix of high-level language intermixed with the corresponding compiled or assembled assembly-level code

Choose the desired view using the drop list in the upper left-hand corner.

Examples of each view are shown in Figure 2, Figure 3, and Figure 4.

Simulating Your Code

Resetting the System

Press the large **Reset** button in the "Toolbox" window to restart the code. The reset button simulates the conditions after the 8032 microcontroller is reset. In a Triscend E5, the 8032 is actively involved in device initialization and is not reset at the end of initialization. Consequently, some of the register contents may vary from the simulation model.

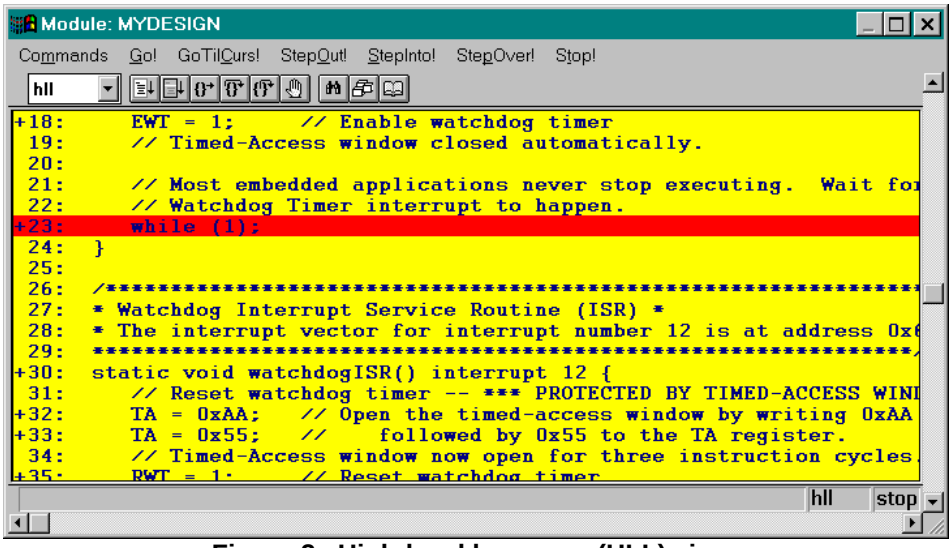


Figure 2. High-level language (HLL) view.

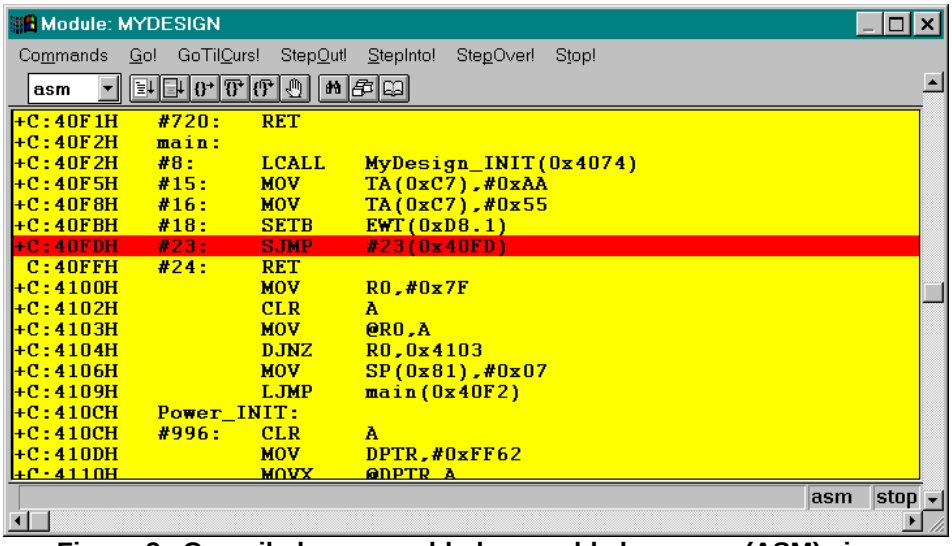


Figure 3. Compiled or assembled assembly language (ASM) view.

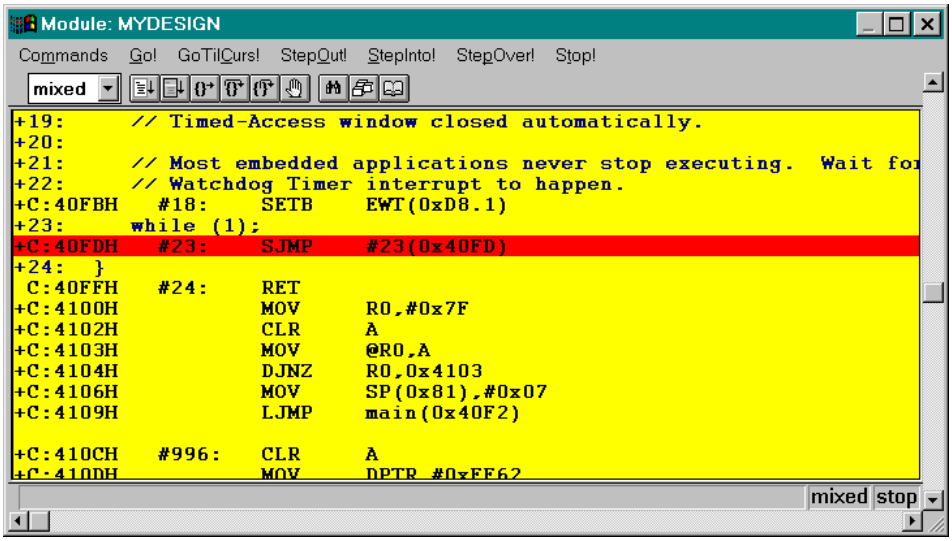


Figure 4. Intermixed (MIXED) high-level language and compiled or assembled assembly language view.

Starting Execution

To start program execution, click the **Go!** menu item or its corresponding menu bar icon.

Once the program starts running, the values change in any open peripheral window and in the Watch window. The register window, "Regs", may not update until the processor is halted.

Stopping Execution

To stop the program, click the **Stop!** menu item or its corresponding menu bar icon. The code viewing window displays the next instruction to be executed once code execution resumes.

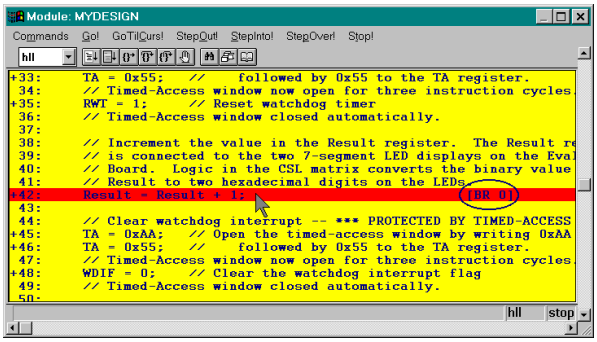
After clicking the **Stop!** button, all windows should stop updating. Any changed registers are highlighted in the "Regs" window.

Single Stepping

To single step through the program, click the **StepInto!** menu item or its corresponding icon on the tool bar. Again, the code viewing window displays the next instruction to be executed once code execution resumes. Likewise, any changed registers are highlighted in the "Regs" window.

Setting a Breakpoint

A program can execute until it reaches a desired line of code by setting a breakpoint. To set a breakpoint, double-click the mouse while the cursor is pointing to the desired statement. A "[BR 0]" indicator appears to the right of the statement indicating that Breakpoint 0 is set.



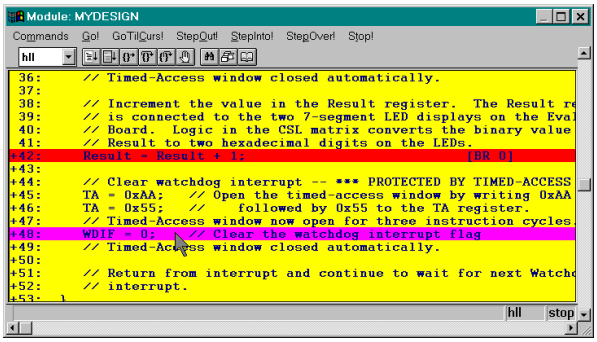
Click the **Go!** button. The program should execute until it reaches the breakpoint. Single-step the program and observe any watched variables in the Watch window. Click **Go!** to resume executing code. Again, the program executes until it reaches the breakpoint.

Execute Until Reaching a Specific Statement

A program can execute until it reaches a particular line, without setting a breakpoint. Use the Up and Down keys or the mouse to point to the desired statement. Don't double-click on this line, otherwise a breakpoint will be set.

Click on the **GoTilCursor!** menu item or the corresponding icon from the menu bar.

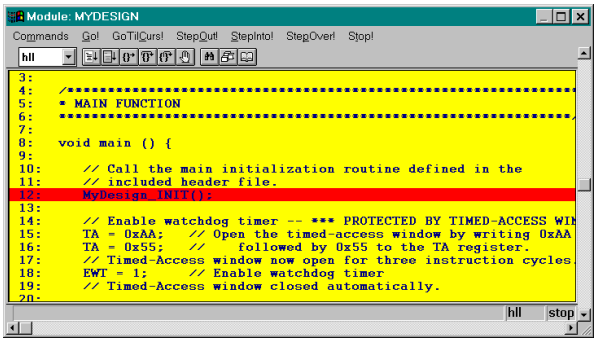
The program executes until it reaches the specified line.



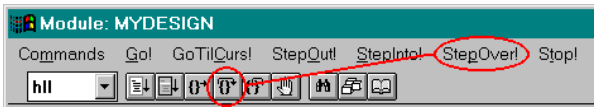
Stepping Over a Subroutine

One additional function is a bit more difficult to demonstrate. The StepOver! function steps over a subroutine or function, without actually stepping through all the statements in the underlying routine. To see how this function operates, press the big Reset button in the "Toolbox" menu.

Single-step until you reach a statement that calls a subroutine.



Skip the dreary details of the subroutine by stepping over it. Click the **StepOver!** menu item or its corresponding icon from the menu bar.



Clicking the button executes the subroutine and then advances the program flow to the next instruction to be executed, skipping the instructions within the subroutine.

Stepping Out of a Subroutine

A similarly useful function is **StepOut!**, which escapes out of subroutine. To see how this function operates, again click the big **Reset** button and single-step until you reach a statement that calls a subroutine. Single-step to enter the subroutine. To execute the remainder of the subroutine and effectively escape from it, click the **StepOut!** menu item or its corresponding icon in the menu bar.

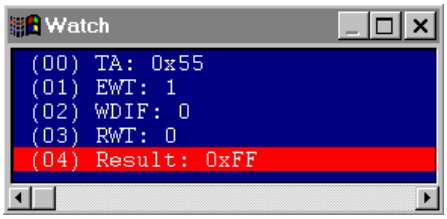
As before, the program proceeds to the next instruction to be executed.

Setting a Register Value

To set a register's value, type in a valid 'C'-style statement in the Command Window. For example, to set a register called Result to all one's, type the following statement in the Command Window. The trailing semi-colon is optional in Keil.

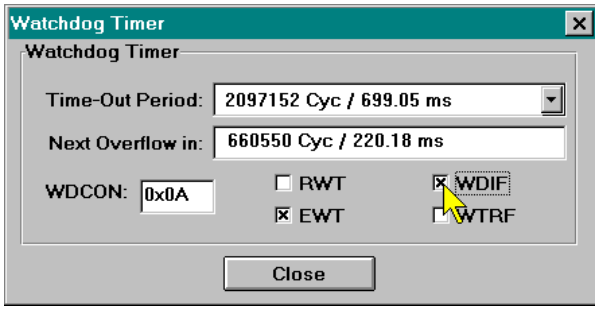
```
Result = 0xff
```

You should see the corresponding value change in the Watch window.



Forcing Interrupts

Setting an interrupt flag is helpful when debugging an interrupt service routine (ISR). For example, to invoke an interrupt service routine created for the Watchdog Timer, the Watchdog Interrupt Flag (WDIF) could be set using the Watchdog Timer peripheral window, as shown below.



In-System Debugging

After creating the hardware and writing the application software, it would usually be time to lug out the in-circuit emulator (ICE) or logic analyzer. Fortunately, the FastChip Development System provides a much more modern and self-contained, in-system debugging environment. Using Keil and FastChip, a Triscend E5 design can be debugged

- In system,
- Operating at full speed,
- With all of the other system hardware and software!

Keil's dScope, in conjunction with FastChip and the JTAG download cable, provides source-level debugging. The same dScope package used earlier to simulate an application program can also be used to talk directly with working silicon, using a different CPU driver. The debugger mode behaves slightly differently than the instruction-set simulator mode. Some of the menu items available during simulation are not available during in-system debugging, specifically the Peripheral options.

Installing Keil Debugger Support

If you install FastChip, then later install Keil from another source, you need to copy the appropriate DLL to run the Keil debugger. First, follow these steps to copy the DLL:

1. Locate the **8032-te5.dll** file in the \apps directory on the FastChip CD-ROM.
2. Copy this file to the Keil \Bin directory.

The TCP/IP protocol must be installed on the host PC in order to use the Keil dScope debugger or to download a design from FastChip.

Connecting Keil dScope with the Triscend CSoC Device

Before using the Keil debugger, recompile the source code and create a .Hex file.

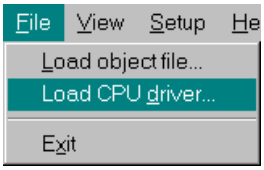
Then download the full design to the Triscend E5 device by clicking the **Download** button in FastChip.

If you don't perform these steps, the Keil object file used by the debugger may not match the design actually downloaded into the Triscend E5 device. This will cause strange behavior.

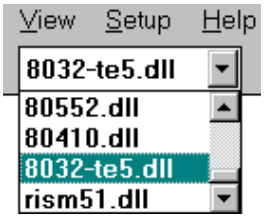
After successfully downloading the design, leave FastChip running.

Invoke the Keil dScope debugger.

To interface Keil's dScope program directly to the E5 CSoC device, load the special Triscend CPU driver included on the FastChip CD-ROM. This CPU driver turns the Triscend JTAG download cable into a real-time source-level interface connected to the operating application. From the dScope menu, click **File** followed by **Load CPU driver...**.



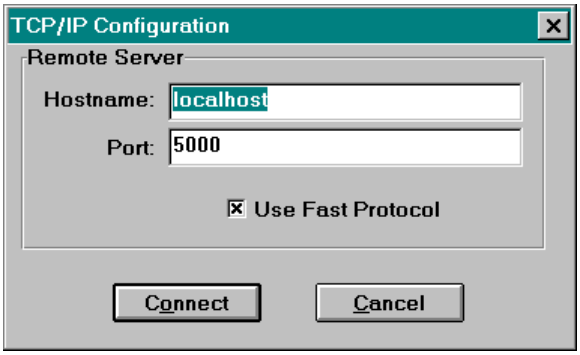
Instead of the driver used to simulate the 80C320 microcontroller select the Triscend E5 driver called **8032-te5.dll**. If this driver does not appear, see "***Installing Keil Debugger Support***".



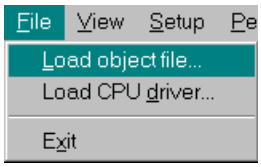
The Triscend driver connects to the JTAG Download Cable, and ultimately your hardware, via a TCP/IP connection. Because the JTAG download cable uses TCP/IP, an application can be developed on computer and then downloaded via a local network or the Internet to another PC in your lab or at a remote location.

If dScope is running on the computer connected directly to the target hardware, keep the default values for the host name and port settings. Otherwise, change the host name to the name of the computer running FastChip.

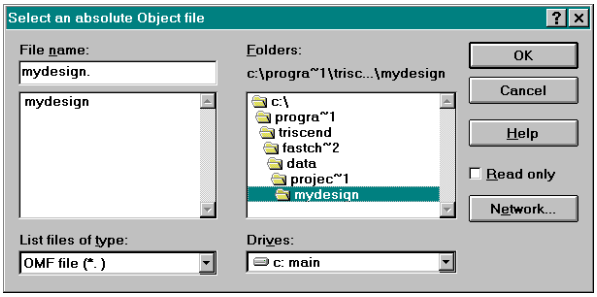
To connect, click the **Connect** button.



Load the object file for the application. Click **File** followed by **Load object file...**.



Select the object file from the resulting dialog box. Once the object file is loaded, the target application may halt operation.



After following these steps, the Keil dScope program is intimately connected to the Triscend E5 CSoC device. The dScope functions used during instruction-set simulation also operate while performing in-system debugging. The only exception is that the peripheral models available in the 80320.dll CPU driver are not available in the 8032-te5.dll driver.

Problems?

If you have problems connecting the Keil source-level debugger to the Triscend E5 development board, possible causes range from the Windows NT service pack version to whether the TCP/IP protocol driver is installed on your computer.

Investigate possible solutions by visiting the online **Triscend SupportCenter** web site.

Automating dScope

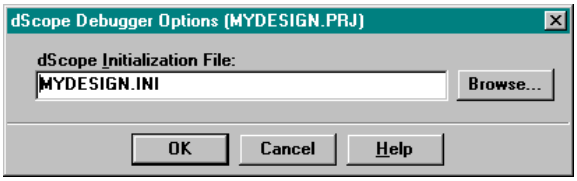
Typing commands and selecting menu items during debugging can be tedious, especially during a repetitive debug cycle. Fortunately, dScope can execute an initialization file when invoked.

The dScope initialization file below is saved as **`mydesign.ini`** and performs the following functions …

- Loads the 80C320 driver, which is similar to the 8032 "Turbo" microcontroller
- Loads the object file for 'MyDesign'
- Defines two buttons. When pressed, one button executes code until reaching main(), the other executes code until reaching the Watchdog Timer interrupt service routine
- Displays two variables, Result and WDCON, in the Watch window
- Initializes the Result register with 0x4a

```
/* Load 80C320 driver, similar to E5 */
load 80320.dll

/* Load the object file */
load mydesign

/* Define buttons */
/* Go until reaching main() */
define button "Go til main()", "g,main"
/* Go until reaching the Watchdog ISR */
define button "Go til wdt_isr()",
      "g,watchdogISR"

/* Setup Watch window */
ws Result
ws WDCON

/* Define contents of Result register */
Result=0x4a
```

An option available within µVision automatically executes the initialization file whenever dScope is invoked. To set the option, select the **Options**, then the **dScope Debugger** menu items.



Type in the name of the initialization file and click **OK** when finished.

Memory Spaces and Memory Models

The Keil compiler provides access to all memory areas associated with the E5's embedded 8032 microcontroller. The various memory areas and data types are shown in Table 3. Each variable can be explicitly assigned to one of these specific memory spaces or data types.

The **sbit**, **sfr**, and **sfr16** data types provide access to the Special Function Registers (SFRs) available on the 8051. These entities are specific to the 8051 architecture and the Keil C51 compiler. They are not a part of ANSI C and cannot be accessed through pointers.

Table 3. Memory and Data Types for the 8032 "Turbo" Microcontroller.

Memory Type	Size (bytes)	Description
sbit	16	The bit-addressable region of the 8032's Special Function Registers
sfr	128	The 8032's Special Function Registers
sfr16	64	Two Special Function Registers treated as a 16-bit integer
code	64K (per bank)	Program memory; accessed by opcode **MOVC @A+DPTR**. Effective code space can be increased using code banking.
data	128	Directly addressable internal data memory; fastest access to variables.
idata	256	Indirectly addressable internal data memory; accessed across the full internal address space.
bdata	16	Bit-addressable internal data memory; allows mixed bit and byte accesses.
xdata	64K	External data memory (64 Kbytes); accessed by opcode **MOVX @DPTR**. Used to access the E5's Configuration Register Unit (CRU), on-chip SRAM, and "soft" module registers defined in **xdata** space.
pdata	256	Paged (256 bytes) external data memory; accessed by opcode **MOVX @Rn**. This memory type is not recommended on the Triscend E5.

An **sbit** variable can be declared either using its absolute address, or using the bit's position within another declared **sfr** location. For example, if a "soft" module for 8032 port P0 is added to the design, bit 3 of the byte-wide port P0 can be declared using the following statement. The P0 SFR is already declared in the header file.

```
; Declared in FastChip header file
    ; sfr P0 = 0x80;

    sbit mybit = P0 ^ 3;
```

Registers located within most "soft" modules can either be located in the microcontroller's SFR space (sfr) or XDATA space (xdata), which is the default. FastChip's Generate utility uses the symbolic names defined in the design as the register variable names. Generate then allocates the variables in SFR or XDATA space as specified in the design, at the FastChip-assigned address location.

In the resulting 'C' header file, the symbolic address of a single-byte soft-module register is declared as a variable of type of **sfr** or **unsigned char**, as shown in Table 4. Similarly, the symbolic address of a multiple-byte soft-module register is declared as an **unsigned char** array.

In the assembly header, each symbolic address is declared as a label to the starting address. Multiple-byte registers must be accessed by incrementing the data pointer from the label, byte by byte.

Other variables used in the 8032 application program can be similarly located within specific memory spaces by including a memory type specifier in the variable declaration.

Accessing internal data memory and SFRs is considerably faster than accessing external data memory. Consequently, place frequently used variables in internal data memory and less frequently used variables in external data memory.

As with the **signed** and **unsigned** attributes, you may include memory type specifiers in the variable declaration.

Table 4. Declarations for "Soft" Module Registers from FastChip.

FastChip Address Space	Resulting Declaration in Header File
SFR	`sfr <symbolic name> = <location>;`
XDATA	`unsigned char xdata <symbolic name> _at_ <location>;`

<symbolic name> = The symbolic name specified by the user in FastChip.
<location> = The address location allocated by FastChip's Generate utility.

If the memory type specifier is omitted in a variable declaration, the default or implicit memory type is automatically selected. Function arguments and automatic variables that cannot be located within the processor's registers are also stored in the default memory area.

As shown in Table 5, the default memory type is determined by the memory model specified in the compiler options.

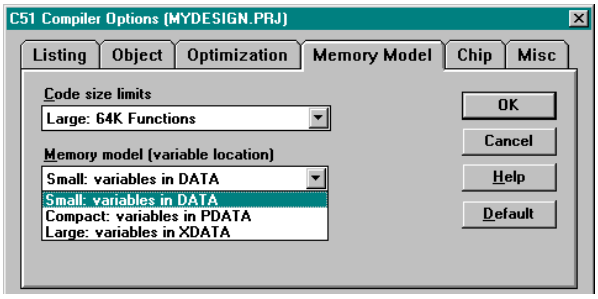
Table 5. Memory Models and Default Memory Types.

Memory Model	Default Memory Type
Small, default	**data**
Large	**xdata**
Compact	**pdata** (not recommended)

Memory Models

The memory model determines the default memory type used for function arguments, automatic variables, and variables declared with no explicit memory type. By default, Keil assumes the Small memory model, which is recommended for most applications. The default memory type can be overridden by explicitly declaring a variable using a memory type specifier.

The memory model option is located in the **C51 Compiler Options** dialog box. Click the **Memory Model** tab and select the desired model from the drop list.



Always use the small memory model, unless the application will not fit or operate using the small model. The small model generates the fastest, tightest, and most efficient code. Variables within the application can always be explicitly specified to reside in other memory spaces.

In the **Small** model, all variables default to the 128 bytes internal data memory within the 8032. This is equivalent to explicitly declaring all variables with the **data** memory-type specifier. Variable access is fast and very efficient, though all data objects, including the stack, must fit within the microcontroller's 128 bytes of RAM. Stack size is critical because the stack space used depends upon the nesting depth of the various functions. Typically, if the BL51 code banking linker/locator is configured to overlay variables in the internal data memory, which is the default, the Small model is the best model to use.

With the **Compact** model, all variables default to one page of external data memory, equivalent to explicitly declaring all variables with the **pdata** memory type specifier. This model is not recommend for E5 CSoC applications. Use the large memory model instead. This memory model accommodates a maximum of 256 bytes of variables. The limitation is due to the indirect addressing scheme used via registers R0 and R1. Though the compact model is faster than the large model, the compact model is awkward on the E5 CSoC.

In the **Large** model, all variables default to external data memory, equivalent to explicitly declaring all variables using the **xdata** memory type specifier. The data pointer (**DPTR**) is used for addressing, though memory accesses are inefficient, especially for variables with a length of two or more bytes. The large memory model generates more code than the small or compact models.

Pointers

The C51 compiler supports pointer declarations using the asterisk character ('*'), similar to ANSI-C. However, due to the 8051's unique architecture, the C51 compiler supports two different types of pointers:

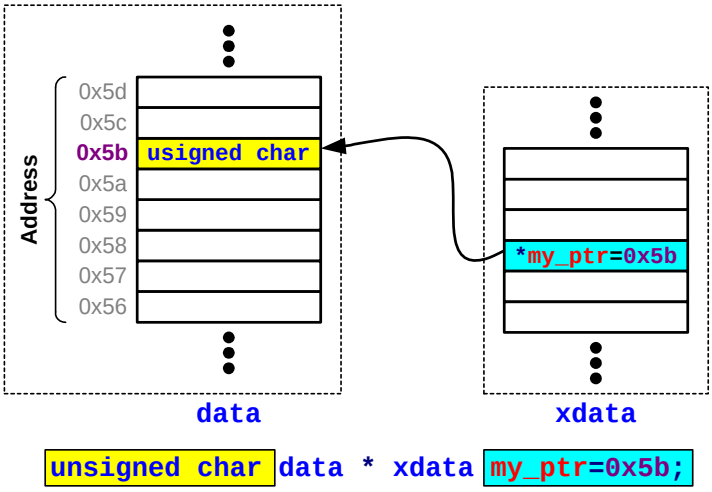
- Generic pointers
- Memory specific pointers

Generic Pointers

Generic pointers are declared similar to standard ANSI-C pointers.

```
char *s; /* string pointer */
int *numptr; /* int pointer */
long *state; /* long pointer */
```

Generic pointers are always stored as three-byte values. The first byte indicates the memory type, while the second and third bytes store the high-order byte and the low-order address byte respectively.




```
item_type mem_type_item * mem_type_ptr ptr_name [= address];
```

Figure 5. A typed pointer, located in xdata memory, pointing to an unsigned character in data memory.

Generic pointers can access any variable, regardless of its location in 8051 memory space. Consequently, many library routines use generic pointers because a function has access to data regardless of the memory type where it is stored.

Declare the memory area in which the generic pointer is stored by using a memory type specifier. The variables referenced by these pointers may reside in any memory area. However, the pointers themselves reside in the specified memory area.

```
/* string pointer in xdata*/
char * xdata s;

/* int pointer in data*/
int * data numptr;

/* int pointer in idata*/
long * idata state;
```

Memory Specific Pointers

Memory specific pointers are more memory efficient than generic pointers. Typed pointers are stored using only one byte (**idata**, **data**, **bdata**, and **pdata** pointers) or two bytes (**code** and **xdata** pointers) because the memory type is specified at compile-time. The memory type byte required by generic pointers is not necessary for typed pointers.

Memory specific pointers always define a memory type in the pointer declaration and always refer to a specific memory area.

```
/* pointer to string in data */
char data *str;

/* pointer to int(s) in xdata */
int xdata *numtab;

/* pointer to long(s) in code */
long code *longtab;
```

By default, pointers are stored in the default memory type, which depends on the memory model used during compilation. However, typed pointers themselves can have a memory type specification, as shown in Figure 5.

- **item_type** is the type of item pointed to, one of the usual 'C' data types such as **signed** or **unsigned char**, **int**, **long**, **float**, *etc*.
- **mem_type_item** is the memory type specifier defining where the referenced item resides in memory, of type **code**, **data**, **idata**, or **xdata**.
- **mem_type_ptr** is the memory type specifier defining where the pointer to the item resides in memory, of type **code**, **data**, **idata**, or **xdata**.

Table 7. Comparing Typed and Generic Pointers.

	idata Pointer	xdata pointer	Generic Pointer

Sample 'C' Program	`char idata *ip;`
	`char val;`
	`val = *ip;`
		`char xdata *xp;`
		`char val;`
		`val = *xp;`
			`char *p;`
			`char val;`
			`val = *p;`

Generated 8051 Assembly Code	`MOV R0,ip`
	`MOV val,@R0`
		`MOV DPL,xp +1`
		`MOV DPH,xp`
		`MOV A,@DPTR`
		`MOV val,A`
			`MOV R1,p + 2`
			`MOV R2,p + 1`
			`MOV R3,p`
			`CALL CLDPTR`

Pointer Size, Space	1 byte, **data**	2 bytes, **data**	3 bytes, **data**
Generated Code Size, Space	4 bytes, **code**	9 bytes, **code**	11 bytes, **code** (plus library function)
Execution Time	4 instruction cycles 16 bus clock cycles	7 instruction cycles 28 bus clock cycles	13 instruction cycles 52 bus clock cycles

- **ptr_name** is the variable name for the pointer.
- **address** is an optional initialization value for the pointer, which must be a legal value for the memory space defined by **mem_type_item.** For example, **data** space is 128 bytes and a pointer to **data** space must be between 0x00 to 0x7F.

Comparing Memory Specific and Generic Pointers

Using memory specific pointers can significantly accelerate a 'C' program targeted to the 8051 architecture. Table 7 shows the differences in code and data size and execution time for pointer declared in **idata** space (fastest, smallest), **xdata** space, and generic pointers (slowest, largest).

Interrupt Sevice Routines

The Triscend E5 has 12 interrupt sources. Writing the corresponding interrupt service routines (ISR) in Keil 'C' requires some special syntax, including the Keil interrupt number.

Triscend E5 Interrupts

Table 6 shows the interrupts available on the E5's embedded 8032 "Turbo" microcontroller. The table also indicates the Keil interrupt number and the corresponding interrupt vector address. The interrupts always appear in the 8032's 16-bit logical code space. The interrupts are listed from highest to lowest priority. The shaded interrupts are those associated with hardware debugging and not normally applied in end-user applications.

Declaring Interrupt Service Routines

C51 Declarations

To create an interrupt service routine (ISR) in Keil 'C', declare a static void function using the **interrupt** keyword and correct interrupt number.

The Keil 'C' compiler automatically generates the interrupt vector plus the entry and exit code for the interrupt routine. The **interrupt** function attribute tags the function as an ISR. Optionally, specify which register bank the ISR utilizes with the **using** attribute. Valid register banks range from 0 to 3.

Table 6. Vector locations for interrupt sources.

Source	Keil Intr. #	Vector Address
High-Priority Interrupt	6	0x0033
External Interrupt 0	0	0x0003
Timer 0 Overflow	1	0x000B
External Interrupt 1	2	0x0013
Timer 1 Overflow	3	0x001B
Serial Port	4	0x0023
Timer 2 Interrupt	5	0x002B
DMA	7	0x003B
Hardware Breakpoint	8	0x0043
JTAG	9	0x004B
Software Breakpoint	10	0x0053
Watchdog Timer	12	0x0063

The example below declares the interrupt service routine for the Timer 0—Keil interrupt number 1—and uses register bank 2.

```
static void T0_ISR(void) interrupt 1
using 2 {
      ...
}
```

A51 Assembly Language Declarations

In assembly, interrupts are declared by their absolute address. For example, the Timer 0 interrupt service routine starts at 0x000B, the interrupt vector shown in Table 6. An interrupt service routine always returns using the "return from interrupt" instruction, RETI, and not the instruction normally used to return from a called subroutine.

```
; TIMER 0 INTERRUPT SERVICE ROUTINE
; Timer 0 appears at 0x000B.
      org 000Bh
      ljmp T0_ISR
; Call the main initialization
; routine defined in the
; included header file.
T0_ISR:
    ; interrupt service routine
    ; goes here
      reti
```

Code Banking

Most 8051 application programs are limited to 64K bytes, the maximum address range supported by the 8051's 16-bit code space. However, Keil's BL51 code banking linker/locator supports building application programs larger than 64K bytes. Special hardware within a Triscend E5 CSoC device allows the embedded 8032 microcontroller to access up to 16M bytes of code space, though the Keil linker supports only 2 Mbytes. The special hardware, called code mappers, allows the 8032 to swap between multiple banks of code, each bank 64K bytes in size. The code mappers must be controlled by application software in a process called bank switching.

The Keil BL51 code banking linker/locator manages one common area and up to 32 separate banks of up to 64K bytes, totaling 2M bytes of bank-switched program space. Software support for bank switching hardware includes a short assembly file targeted for the Triscend E5.

To create very large and efficient applications, carefully group functions in the different banks. Locate particular program modules to specific banks in order to minimize bank switching.

Common Area

In a bank-switching program, the common area is a region of memory accessible at all times from all banks. The common area must always be available and cannot be physically swapped out or moved around. The code in the common area is duplicated in each bank.

The common area contains program sections and constants that must be available at all times. It may also contain frequently used code, to minimize bank switching. By default, the following code sections are automatically located in the common area:

- Reset and interrupt vectors
- Code constants
- C51 interrupt functions
- Bank switch jump table
- Some C51 run-time library functions

Executing Functions in Other Banks

The Keil code-banking linker—in conjunction with the Triscend-supplied **`l51_bank`** routine—selects a particular code bank by programming the E5's CMAP1 code mapper with the base address of the target code bank. The Keil code-banking linker automatically generates a jump table for functions located in other code banks.

When the application code calls a function in a different bank, the program switches to the other bank and jumps to the desired function. When the function completes, the program restores the previous bank and returns execution to the calling routine.

The bank switching process requires approximately 50 instruction cycles and consumes two additional bytes of stack space. To improve system performance, group interdependent functions in the same code bank. Functions frequently invoked from multiple banks should be relocated to the common area.

Code Banking on the Triscend E5

FastChip includes a modified version of the Keil code-banking library that takes advantage of the E5's code mappers. The Keil code-banking

scheme supports up to 32 code banks, each 64 Kbytes in size. The common code area is repeated in each bank. Please refer to the Keil documentation for more information about the Keil code banking support. Additional information is available from the comments in the Triscend code-banking library source-code.

Configuring the code banking library

The code-banking library has two constants that might require modification, depending on the project.

- **?B_NBANKS** — the number of banks supported (default is 16, maximum of 32).
- **?B_RTX** — set to 1 when using Keil's RTX-51 FULL operating system; otherwise set to 0 (default is 0).

?B_NBANKS must be set equal to or greater than the number of code banks, otherwise link errors will occur. Setting **?B_NBANKS** to a value larger than the number of banks required results in some wasted code because the library generates code to switch to the unused banks. If the application is highly sensitive to code size, then set **?B_NBANKS** to exactly match the number of banks required.

Using the code banking library

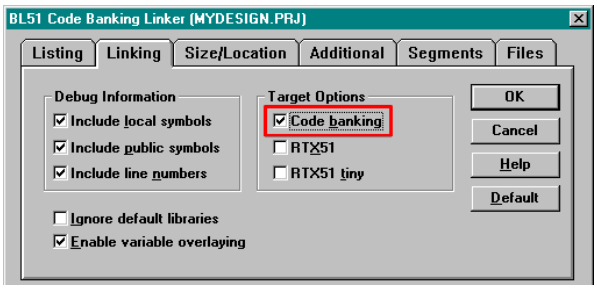
Follow these steps to use the code-banking library:

1. Archive the existing **`l51_bank.a51`** file found in the C51/LIB directory to a different file name or another directory.
2. Copy the Triscend **`l51_bank.a51`** and **`l51_bank.obj`** files from the FastChip1999/Keil/Codebank directory to the C51/LIB directory.
3. If necessary, edit the **`l51_bank.a51`** file to change the configuration settings. Then, assemble the changes using the following command to create a new 151_bank.obj file:

   `a51 l51_bank.a51`

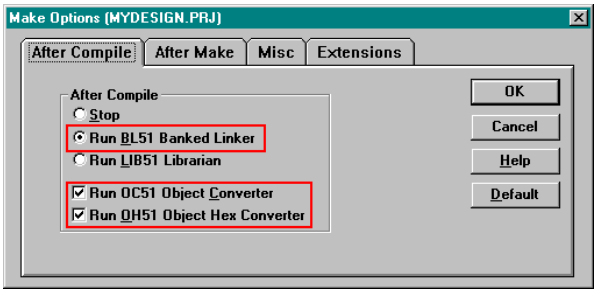
The library is now ready to use when linking the application with the BL51 Code Banking Linker. Set the following options in Keil's µVision development environment. Select **Options**, then **BL51 Code Banking Linker...** from the menu.

From the resulting dialog box, select the **Linking** tab. Choose the option **Code Banking** under Target Options.



Click **OK** when finished.

Now select **Options**, then **Make...** from the menu. Click the **After Compile** tab from the resulting dialog box. Select the **Run BL51 Banked Linker** option, and enable the **Run OC51 Object Converter** and **Run OH51 Object Hex Converter** options.



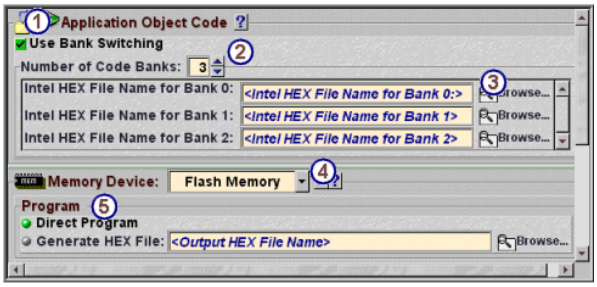
Click **OK** when finished.

Downloading Banks using FastChip

After compiling and linking a code-banked application, Keil creates multiple .Hex files, one for each 64K-byte bank.

To download multiple code banks to the Triscend E5, invoke FastChip and click on the **Download** button.

Once the download dialog box appears, perform the following steps.



1. Enable the **Use Bank Switching** option.
2. Specify the **Number of Code Banks** to be downloaded.
3. Select the .Hex files for each bank. Bank 0 is always the bank executed at start-up. Each bank is loaded at a successive 64K boundary in memory.
4. Choose the appropriate **Memory Device** for download. Only external byte-wide memories are applicable for code banked applications.
5. Choose the **Program** download method. Either program the E5 CSoC device directly, or save the image as a new, combined .Hex file to be programmed into an external memory device using an external programmer.
6. Click **OK** when finished to start the downloading process.

Summary

Triscend's FastChip development system intimately supports features available in Keil's 8051 development software.

Using the full capabilities of the Keil compiler, code-banking linker, and dScope debugger, a designer can create fast and efficient application programs for the Triscend E5 CSoC devices.

Revision History

Revision	Date	Comment
1.00	22-DEC-1999	First release.
1.01	6-JAN-2000	Minor formatting and wording changes.
1.02	11-JAN-2000	Corrected various spelling errors.

Triscend Corporation
301 N. Whisman Rd.
Mountain View, CA 940403-3969

Tel: 1-650-968-8668 x166
Fax: 1-650-934-9393

E-mail: **contact_us@triscend.com**
Web: **www.triscend.com**

AN07

Using Keil Development Tools with Triscend FastChip and the E5 CSoC Family

— Table of Contents —

Triscend