2

Programming the 8032 MCU Core

Objectives

- Become familiar with the 8032 microcontroller core on the Triscend CSoC.
- Learn how to instantiate and use 8032 I/O ports.
- Learn how to use some of the dedicated peripherals of the 8032.
- Learn how to use the Keil software development environment to write C programs.
- Learn how to create interrupt-driven programs.
- Learn how the Triscend CSoC handles code and data memory spaces.

Microcontroller Resources in the CSoC

Your introduction to the Triscend CSoC continues with an exploration of the built-in 8032 microcontroller core. The areas of the CSoC you will use are:

- the *8032 microcontroller unit* (MCU) with its dedicated peripherals,
- the *byte-wide system SRAM,*
- the *memory interface unit* (MIU),
- the *address mappers,*
- the *JTAG interface*,
- the *hardware breakpoint unit*.

These areas are highlighted in Figure 11.

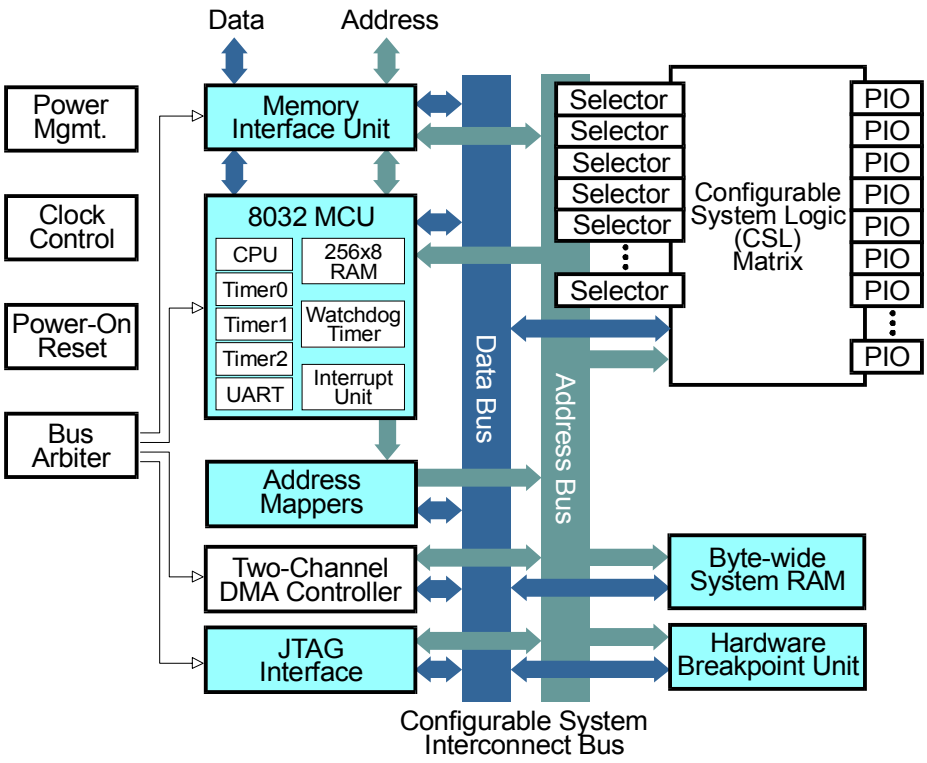


Figure 11: Areas of the Triscend CSoC used in the designs in this chapter.

The 8032 Microcontroller Unit

The Triscend CSoC contains an enhanced version of the industry-standard 8051 MCU architecture. The 8051 architecture has been described in hundreds of books over the past twenty years, so I won't try to replicate that body of work here. I will list the features of the enhanced 8032 MCU in the CSoC:

Turbo CPU: The standard 8051 uses twelve clock cycles per instruction cycle, but the 8032 MCU in the CSoC uses only four clock cycles. The 8032 MCU also has two data-pointer registers versus the single data-pointer in the original 8051.

SRAM: The 8032 MCU contains 256-bytes of SRAM for use as scratch-pad storage and to hold the program stack.

UART: A universal asynchronous receiver/transmitter (UART) is built into the MCU for serial data communications. The 8032 UART offers automatic address recognition and frame error detection in addition to the standard features of the original 8051 UART.

Timers: The 8032 MCU has three programmable timers which are used to generate pulse-trains, periodically interrupt the MCU, and set the bit rate of the UART.

Watchdog timer: The watchdog timer resets the 8032 MCU to a known state unless the MCU regularly clears the watchdog. If the program running in the 8032 MCU goes astray and the watchdog is no longer being cleared, then the watchdog will restart the program.

Interrupt unit: The 8032 MCU responds to interrupts from twelve different sources that are organized into three levels of priority.

The Byte-wide System SRAM

The TE505 on your CSoC Board contains 16 KBytes of fast static SRAM (SRAM). This SRAM can store data or programs for the 8032 MCU.

The Memory Interface Unit

The MIU controls the access of the CSoC to external memory or other devices. The set-up and hold timing for external devices can be programmed into the MIU.

The Address Mappers

The 8032 MCU uses 16-bit addresses to fetch instructions and data from internal or external memory. The address mappers expand these addresses to 32-bits and send them to the CSI address bus. At this point, the translated address will determine whether the internal SRAM, external SRAM, or CSL is accessed. The address mappers are programmable, so you can select which 8032 address ranges will activate an internal or external memory access.

The JTAG Interface

You use a PC host system to program and debug a Triscend CSoC through a JTAG interface consisting of four wires:

TCK: This clock input to the CSoC synchronizes the signals on the other JTAG wires.

TMS: The logic value on this input to the CSoC is used to steer the internal state machine of the CSoC into various states for configuring the CSoC, debugging its operations, etc.

TDI: This wire carries configuration data or other instructions to the CSoC.

TDO: This wire carries data from the CSoC back to the host system.

The Hardware Breakpoint Unit

The 8032 MCU on your CSoC Board runs at up to 25 MHz. The four-wire JTAG interface is too narrow to monitor the program and data accesses at this speed. You can program the hardware breakpoint unit to monitor the 32-bit CSI address bus, the byte-wide data bus, the control signals, and the type of memory access (instruction fetch or data read/write). When a trigger event is detected, the hardware breakpoint unit halts the CSoC and alerts the host system through the JTAG interface. Then you can examine the state of the CSoC.

You will use the 8032 MCU, MIU, internal and external SRAM, and the address mappers in the design examples that follow. You will not be directly involved with the JTAG interface and hardware breakpoint unit, but these will be used by the software tools as you do the examples.

Design 2.1 - 8032 MCU I/O Ports

Your first MCU-based CSoC design will read the settings of the DIP switch and display the lower seven-bits on the seven-segment LED digit of the CSoC Board (Figure 12). The MCU will continuously poll the DIP switch settings and then write the settings into a register that drives the LED digit.

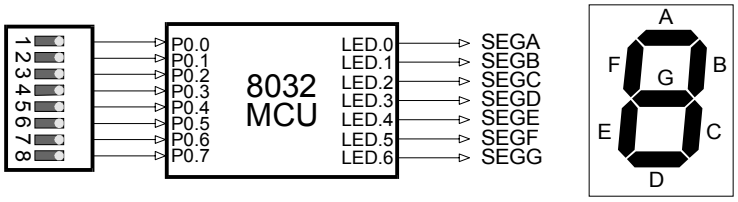


Figure 12: Block diagram of an MCU-based system that reads DIP switch settings and displays them on an LED digit.

This design example requires you to design some hardware to read the DIP switch and drive the LED digit, and then you have to write some software for the 8032 MCU that passes the value from the DIP switch circuitry to the LED driver. You will create the hardware in the next section and then go on to write the software.

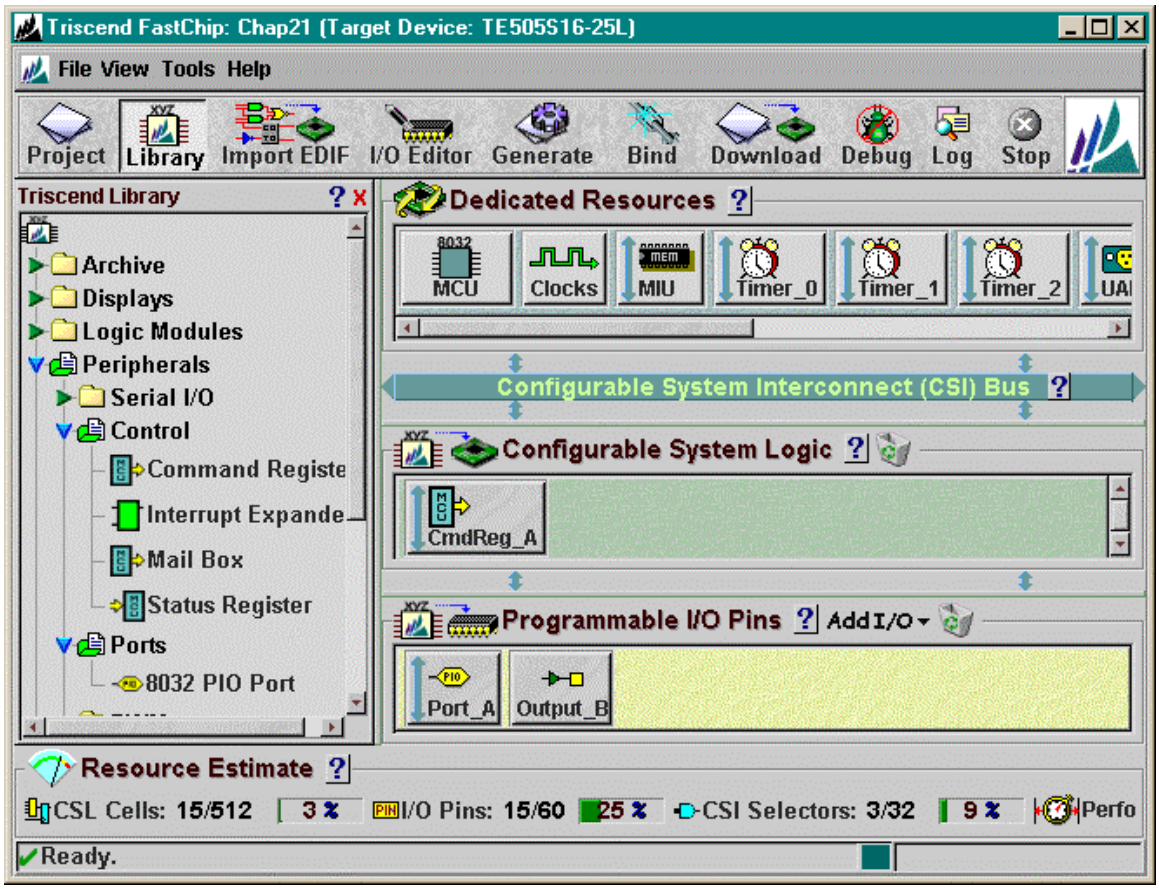
Instantiating the Hardware

To begin this design, start FastChip and create a project called ***Chap21***. Then open the library of soft modules and select Peripherals⇒Ports⇒8032 PIO Port from the menu and drop it into the Programmable I/O Pins area. This places a standard 8032 I/O port into your design that you can use to read the DIP switch settings.

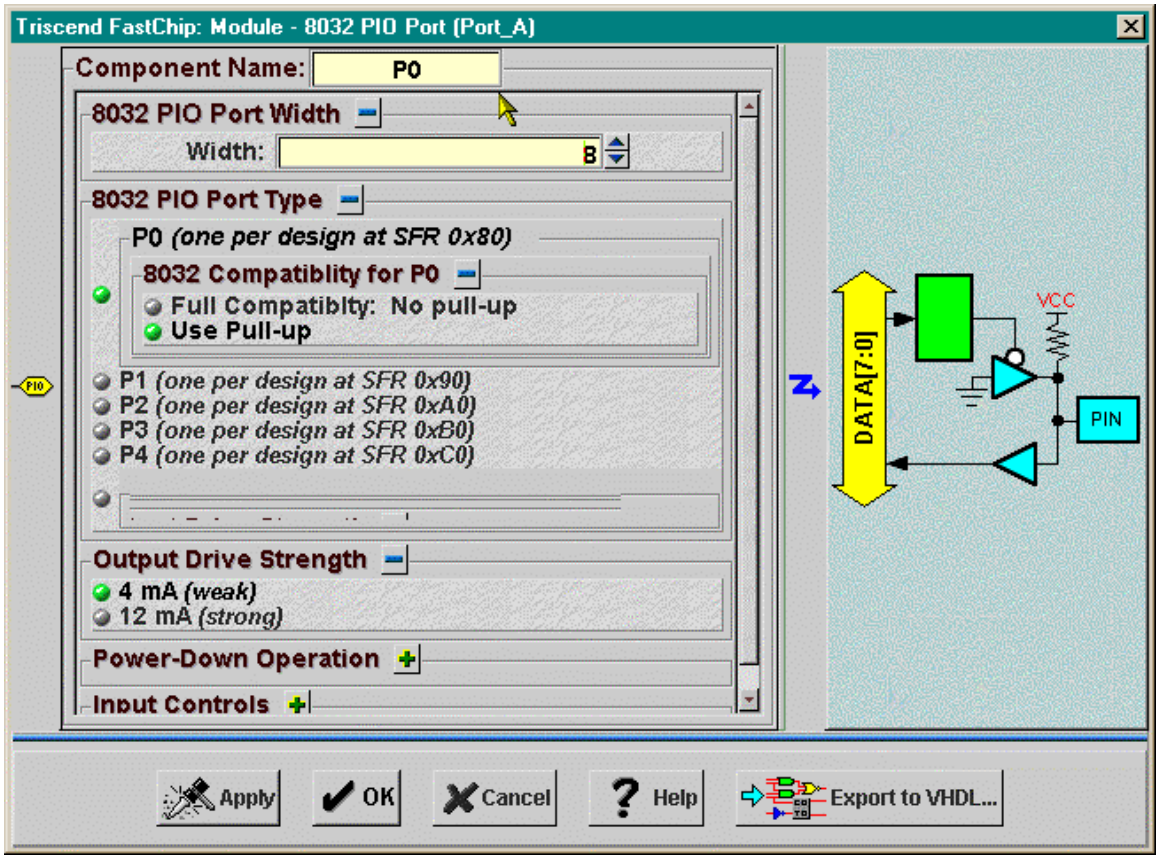
Next, select I/O⇒Output and drop an output port into the same area. You will use this output port to drive the LED. This output has to be driven with a value written by the 8032 MCU, so select Peripherals⇒Control⇒Command Register from the library and place it in the Configurable System Logic area. This instantiates a register that the MCU can write with the value to be displayed on the LED.

Why not use another 8032 port to drive the LED digit rather than the combination of a command register and an output? Because the standard 8032 port cannot source enough current to illuminate the LED segments (it is limited to sourcing less than 100 μA). But the **Output** module is capable of sourcing up to 12 mA per pin, which is plenty.

After instantiating these soft modules, your project design window will appears as follows.



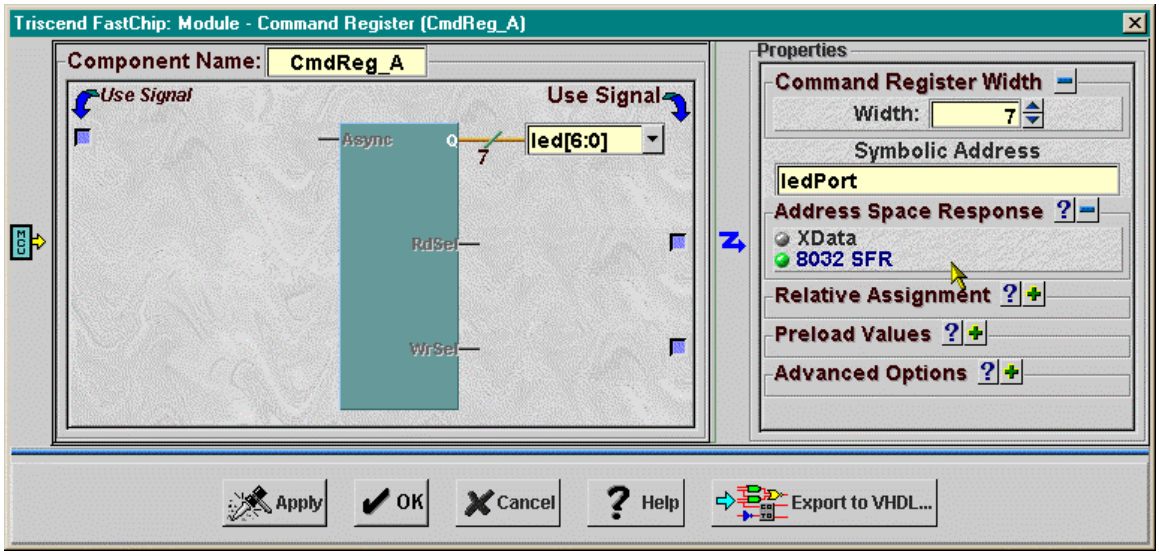
Now you need to modify the modules to suit the design. Click on the **Port_A** module to make the following window appear.



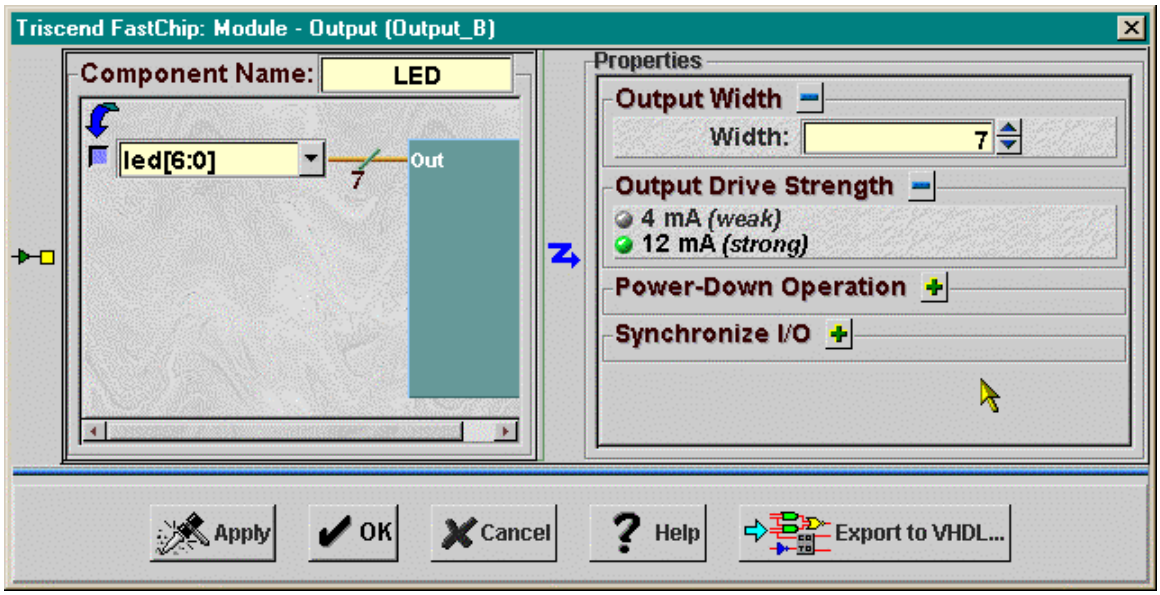
Within the window, select **P0** in the 8032 PIO Port Type area. Then type `P0` into the Component Names box to reflect the function of this port. You will use this port to read the DIP switch settings, so the output drive strength is irrelevant. Also, since the DIP switch circuitry on your CSoC Board already has external pullups, you can enable or disable the P0 pullups without affecting the operation of your design. Then click on OK to close the window.

Next, click on the **CmdReg_A** module. When the **Module** window opens, change the register width to seven bits (since there are seven LED segments) and change the name of the register outputs to **led**. The 8032 MCU will write values into this register that will appear on the led outputs, so the MCU needs a way to reference the register. In the Symbolic Address box, type the name by which you will reference this register (`ledPort`) in the program code for the 8032. You can select whether the MCU accesses this register as if it were in the *external data memory space* (Xdata) or as if it were a *special function register* (SFR). For external data accesses, the 8032 uses a data pointer that is initialized to the register address. This takes a bit more code and a bit more time than would be the case if the register was accessed as an SFR. SFRs have addresses in the range [0x80,0xFF] so there are not very many of them. (A lot of them are already used by the MCU's dedicated peripherals like the UART and timers.) In this example, code size and execution time are not critical and there is plenty of

space in the SFR address region, so you can select either method. (I chose to make the register an SFR.) The click on OK to finalize your modifications.



Click on the **Output_B** module to bring up the window that will configure the LED digit drivers. Change the name of the module to **LED** so its function is easily discernable. Adjust the output width to seven bits and connect the outputs to the **Command Register** module by typing `led` into the <output> field. Change the drive strength to 12 mA so the LED segments will be brightly lit. Then click on OK to set the changes.



Now that the modules have been instantiated and their connectivity has been specified, you can assign the inputs of the **P0** port and the outputs of the **LED** port to the pins of the CSoC using the I/O Editor. As shown below, the **LED** port consists of outputs only (signified by red, unidirectional arrows) while the **P0** port is capable of input and output

functions (represented by blue, bi-directional arrows). Click and drag the **LED** and **P0** I/Os to the pins listed in Table 6.

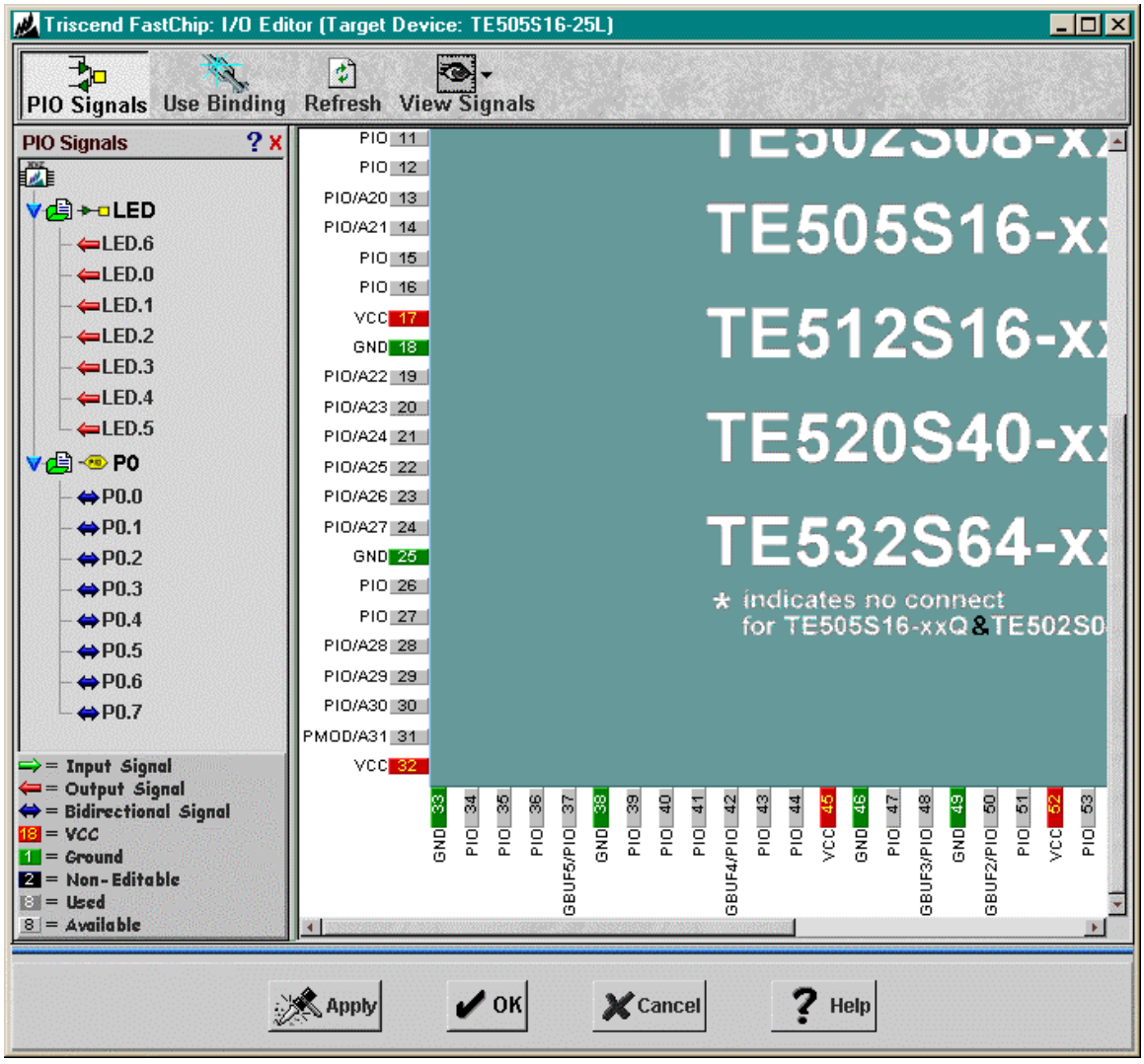
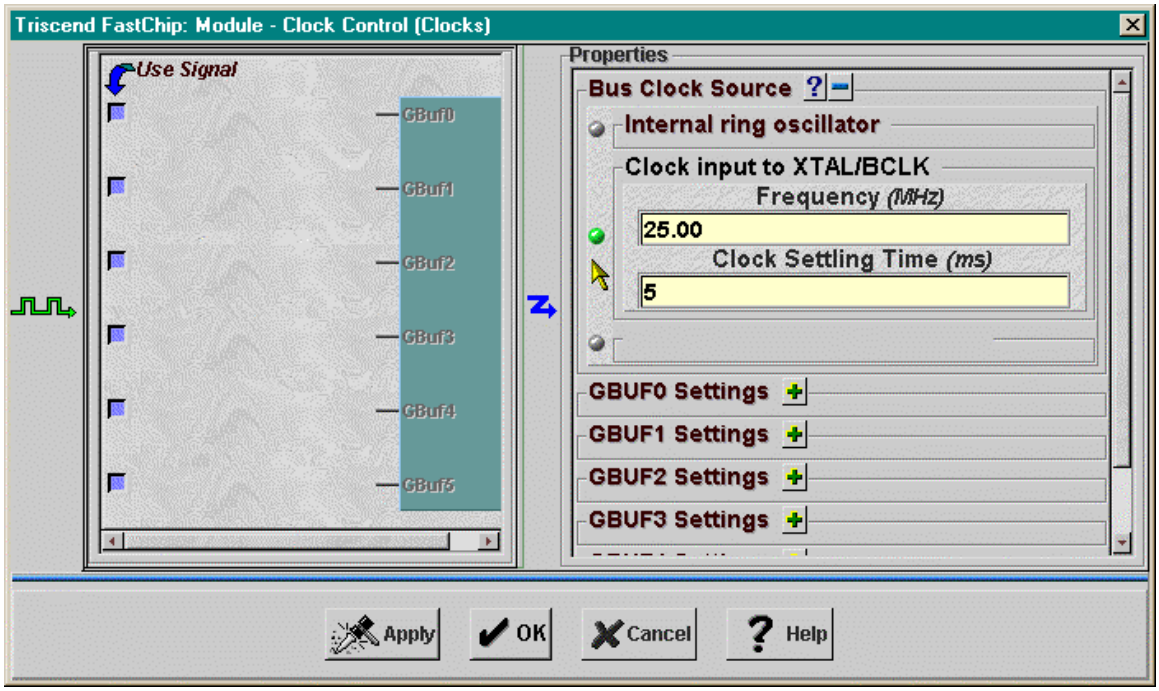


Table 6: Pin assignments and functions for the 8032 MCU I/O port design.

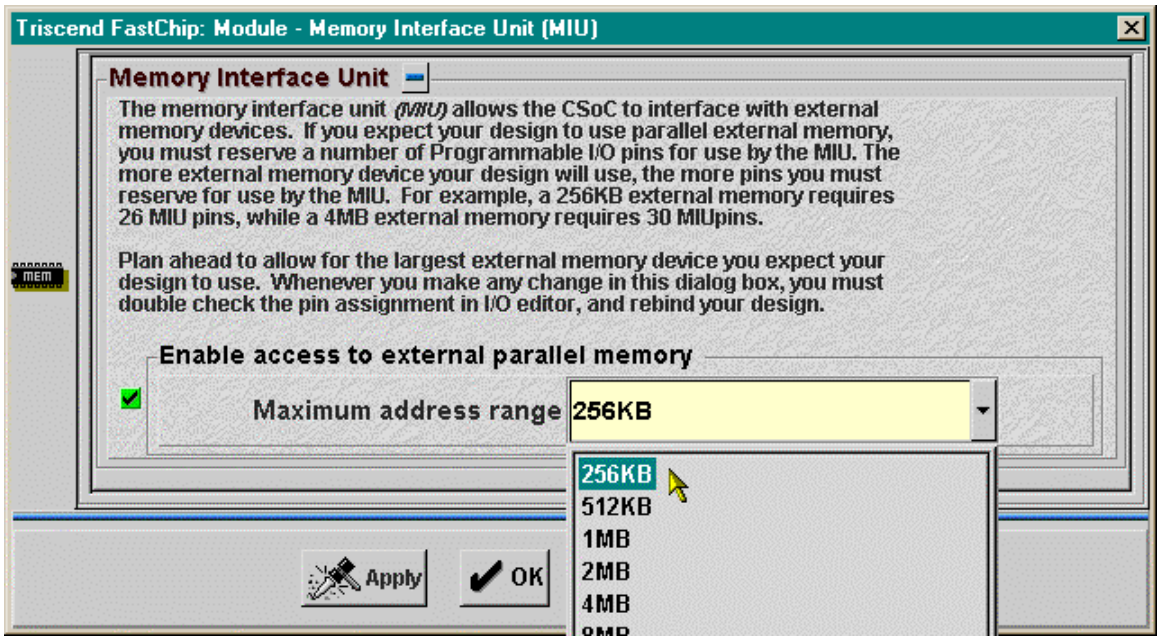
Signal	Pin	CSoC Board Resource
P0.0	53	DIP switch position #1
P0.1	54	DIP switch position #2
P0.2	55	DIP switch position #3
P0.3	58	DIP switch position #4
P0.4	59	DIP switch position #5
P0.5	60	DIP switch position #6
P0.6	62	DIP switch position #7
P0.7	63	DIP switch position #8
LED.0	35	LED digit segment A
LED.1	39	LED digit segment B
LED.2	43	LED digit segment C
LED.3	41	LED digit segment D
LED.4	40	LED digit segment E
LED.5	34	LED digit segment F
LED.6	36	LED digit segment G

You need to specify a clock to sequence the operations of the MCU. To do this, click on the Clocks icon in the Dedicated Resources area of the project window. The **Clock Control** window that appears lets you select one of several sources for the **BusClock** signal. Click the button for the second option: Clock input to XTAL/BCLK. This selects the dedicated clock input to the Triscend CSoC as the driver for **BusClock**. An external programmable oscillator is connected to this input on the CSoC Board. If you have configured your CSoC Board as described in Appendix B, then the external oscillator will output a stable 25 MHz clock frequency. Type `25.00` into the Frequency field to let the FastChip software know this is the frequency of the external clock source. Then type `5` into the Clock Settling Time field. (This value is used by the Triscend CSoC to keep the chip in a reset state until the main clock is stabilized after power-up. This parameter isn't very important in your current environment since the CSoC Board is already

powered before the chip is ever configured with your timer circuit.) Then click on OK to close the window.



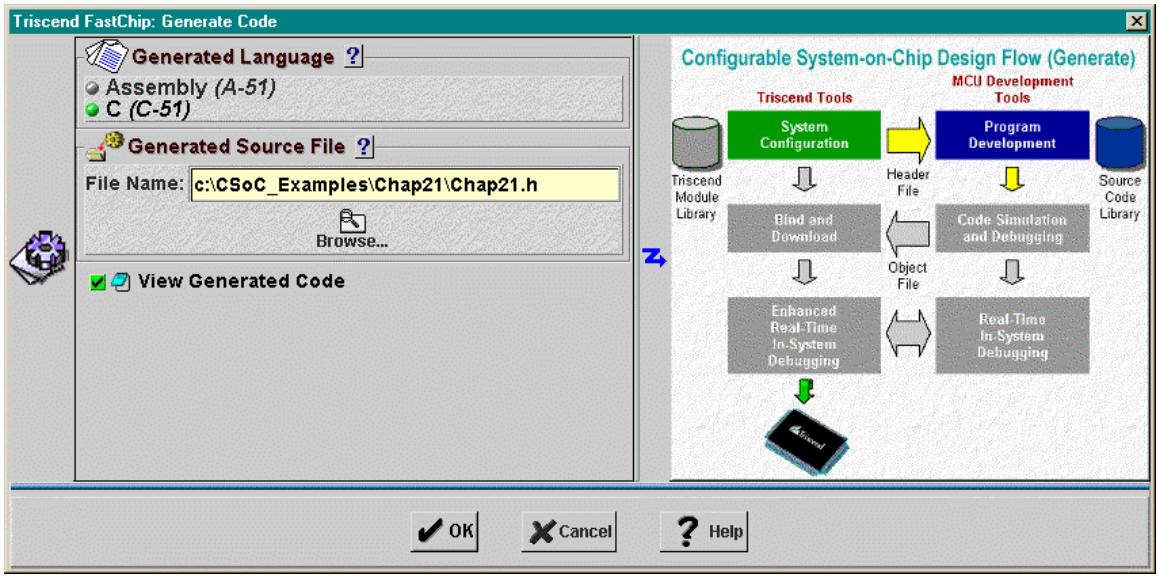
Finally, you need to set-up the memory interface unit because the instructions for your MCU program will be stored in external memory. Click on the MIU icon in the Dedicated Resources area of the project window and the following window will appear. This window lets you select how many address bits will be used by the MIU to access external memory. There are only 128 KBytes of external SRAM on your CSoC Board, so select the smallest address range in the drop-down list (256 KBytes). This wastes one address bit and the SRAM contents will be replicated twice within the 256 KByte address range, but this won't cause any problems. Click on OK and move to the next step.



Generating the Hardware/Software Interface

At this point you have defined the hardware for your design. Now you have to generate a header file that links the hardware to the software you will write for the 8032 MCU. Among other items, this header file will contain the addresses of the **P0** and **ledPort** registers you built into your hardware. Your application code will use these addresses to read the DIP switch settings and write the data to the register that drives the LED digit.

You start the creation of the header file by clicking on the Generate toolbar icon. The following window will appear.



You can generate a header file for use with either 8032 assembly or C code by clicking on the Assembly or C button in the Generated Language area of the window. I used to write 8052 assembly code in the early 80's and I have no desire to revisit those times. Now I write as much of my application software in C as possible. You are free to use either language, but I will not present any examples of assembly language coding in this text. Click on the C radio button to generate the C header file.

By default, the header file will be created in your FastChip project folder with the same name as your project. You can use the Browse button to choose another folder, but I am usually happy with the default. Once you select the language for your header file, the File Name box in the Generated Source File area will change the suffix of the header file to .inc or .h depending upon whether you chose to use assembly or C language, respectively.

Clicking on OK generates the actual header file. If you checked the View Generated Code box, then a window appears with the contents of the header file (Listing 1). The interesting part of the header file is found on lines 31–35. The **ledPort** register is placed in the SFR address space of the 8032 (as you specified previously when you customized the **CmdReg_A** module) at address 0x9a. And port **P0** is placed at address 0x80 in the SFR which is the standard address for this port in the 8051 MCU architecture.

Listing 1: Top of the header file generated by the FastChip software for the 8032MCU I/O port design.

```
1  // Generated 3/5/00 11:39 AM By FastChip Version 1999 Build 15
2
3  //////////////////////////////////////////////////////////////
4  //  ---------------------------------
```

```
5   //  ------   GENERATED CODE   --------
6   //  --------------------------------
7   //  The code in this header file was generated automatically for your
8   //  project by Triscend FastChip. Please DO NOT EDIT this header file.
9   //  It will be overwritten the next time FastChip generates code for
10  //  your project.
11  //
12  ////////////////////////////////////////////////////////////////
13
14  //======== Required symbol and macro definitions ========
15
16  #ifdef PROTOTYPE_ONLY
17  #  define CHAR_XDATA(name,location) extern volatile unsigned char xdata name;
18  #  define CHAR_ARRAY_XDATA(name,location,size) extern volatile unsigned char
19  xdata name[size];
20  #else
21  #  define CHAR_XDATA(name,location) volatile unsigned char xdata name _at_
22  location;
23  #  define CHAR_ARRAY_XDATA(name,location,size) volatile unsigned char xdata
24  name[size] _at_ location;
25  #endif
26
27  //========= BEGIN SOFT MODULE REGISTER DECLARATIONS ======
28
29  //-------------------------------- Module CmdReg_A
30      sfr ledPort = 0x9a;
31
32  //-------------------------------- Module P0
33      sfr P0 = 0x80;
34
35  //========== END SOFT MODULE REGISTER DECLARATIONS =======
```

The header file contains a bunch of other address definitions for the standard SFRs of the 8051 as well as some additional SFRs and extended data memory locations that control functions found only in the Triscend 8032 MCU. These are followed by a series of definitions for subroutines that initialize the peripherals of the 8032 such as the timers and the UART. At the very bottom of the header file is a master initialization subroutine (`Chap21_INIT()`) that calls all the other initialization subroutines (Listing 2).

Listing 2: Bottom of the header file generated by the FastChip software for the 8032 MCU I/O port design.

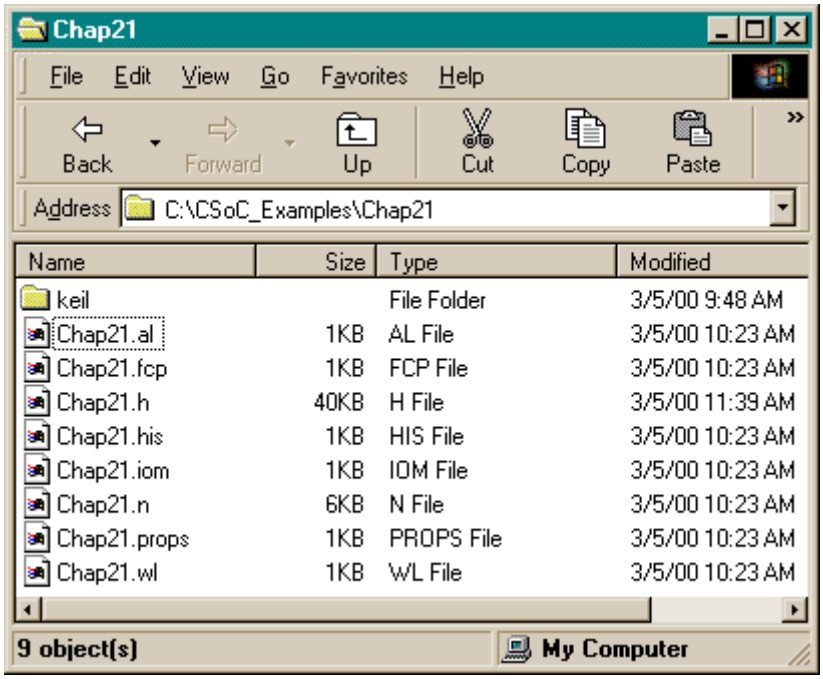
```
1   //========== PROJECT INITIALIZATION FUNCTION ======
2
3   #ifdef PROTOTYPE_ONLY
4
5   extern void Chap21_INIT();
6
7   #else
8
9   void Chap21_INIT () {
```

```
10      Timer_0_INIT();
11      Timer_1_INIT();
12      Timer_2_INIT();
13      UART_INIT();
14      Interrupt_INIT();
15      Watchdog_INIT();
16      DMA_0_INIT();
17      DMA_1_INIT();
18      Power_INIT();
19  }
20
21  #endif
```

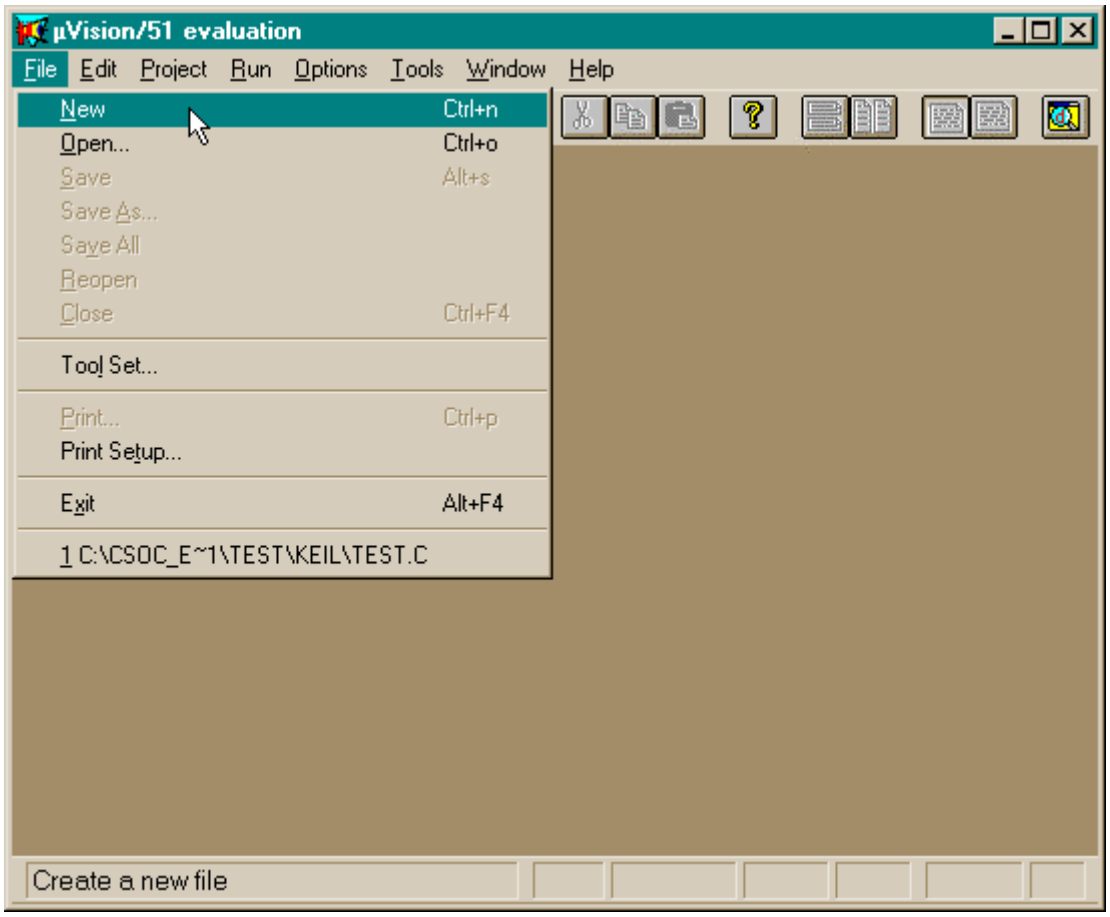
Creating the 8032 MCU Application Code

Now that you have the hardware designed and the header file that specifies the addresses of the registers in the hardware, you can begin to create the application code that will run in the 8032 MCU. To do this, you will have to leave the FastChip environment and use software development tools for the 8051 architecture. In this text I will use the Keil 8051 software tools, but you can use any 8051 compiler (although the steps in creating and compiling the code will probably be different from what I show here).

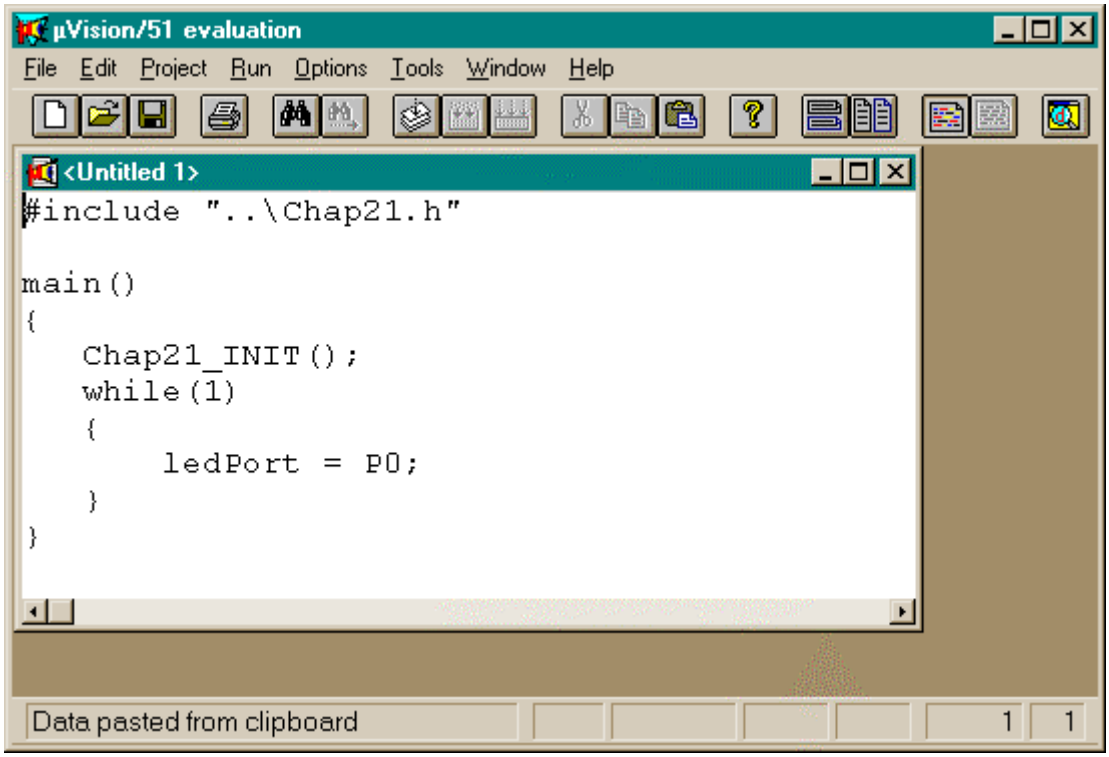
To begin, you should create a folder where you will store your application code. I usually create a folder called keil within the FastChip project folder as shown below.



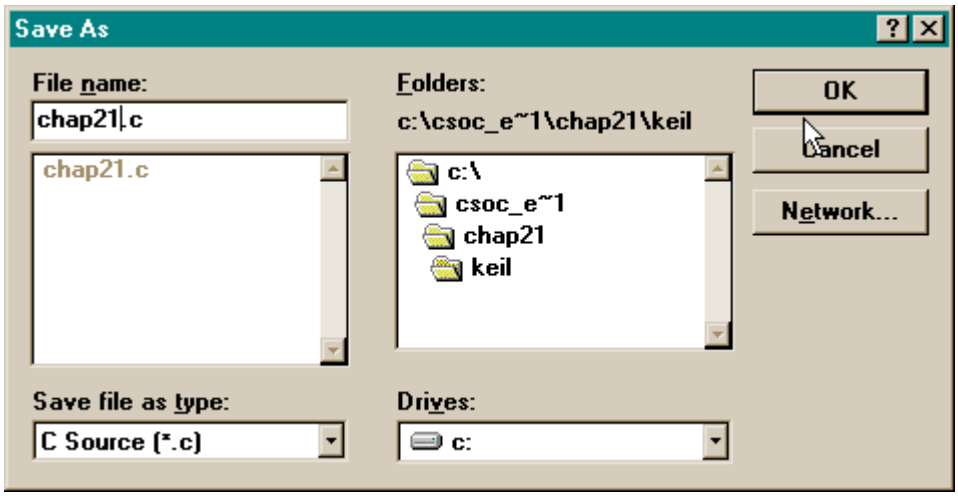
Once the folder is created, click on the icon to start up the Keil integrated development environment (IDE). In the **µVision/51** window that appears, click on the File⇒New menu item to open a subwindow.



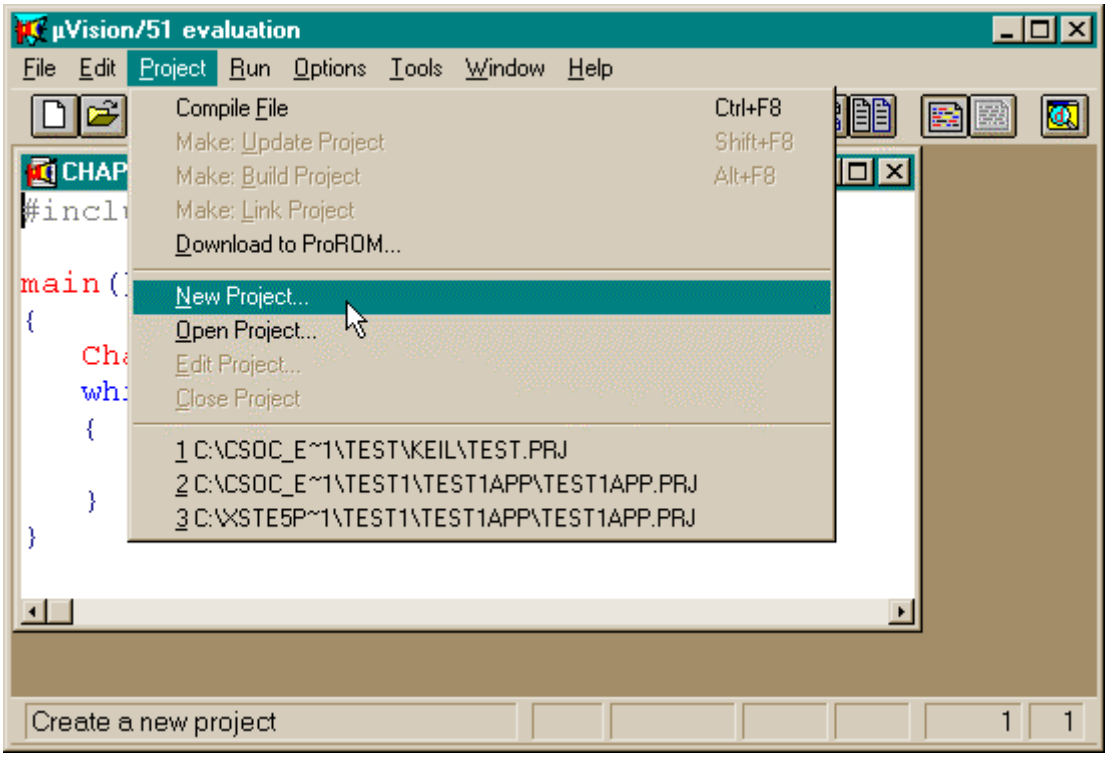
In the **<Untitled 1>** window, type the source code for the application. Start with an `include` statement that brings in all the definitions from the Chap21.h header file. (Recall that the header file was generated in the FastChip project folder directly above the keil folder.) Then create a `main()` routine whose first action is to call the `Chap21_INIT()` subroutine that initializes all the 8032 peripherals. Finally, place an infinite `while`-loop that continually reads the **P0** port and writes the value into the **ledPort** register. The header file will inform the compiler of the addresses for these registers in the SFR space. That's all there is to the application code for this example!



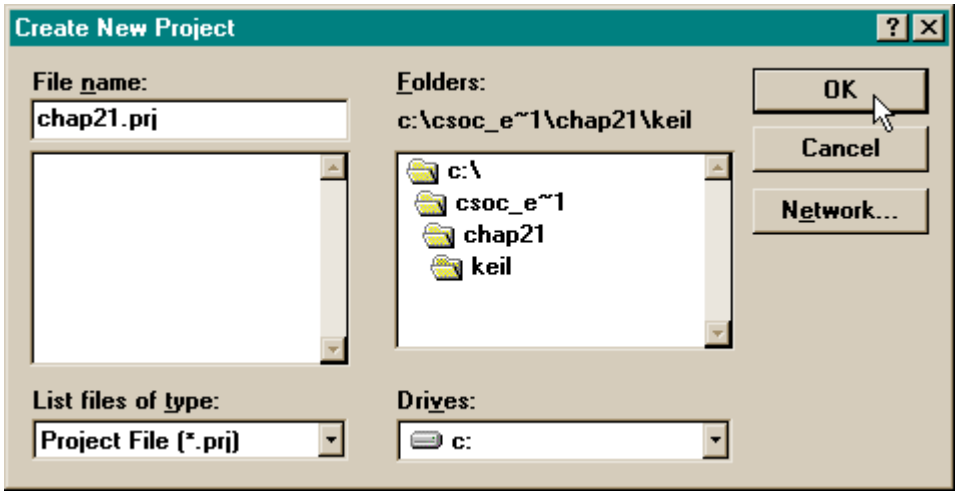
Click on File⇒Save As... to bring up the following window. Store your source code in the chap21.c file in the keil folder.



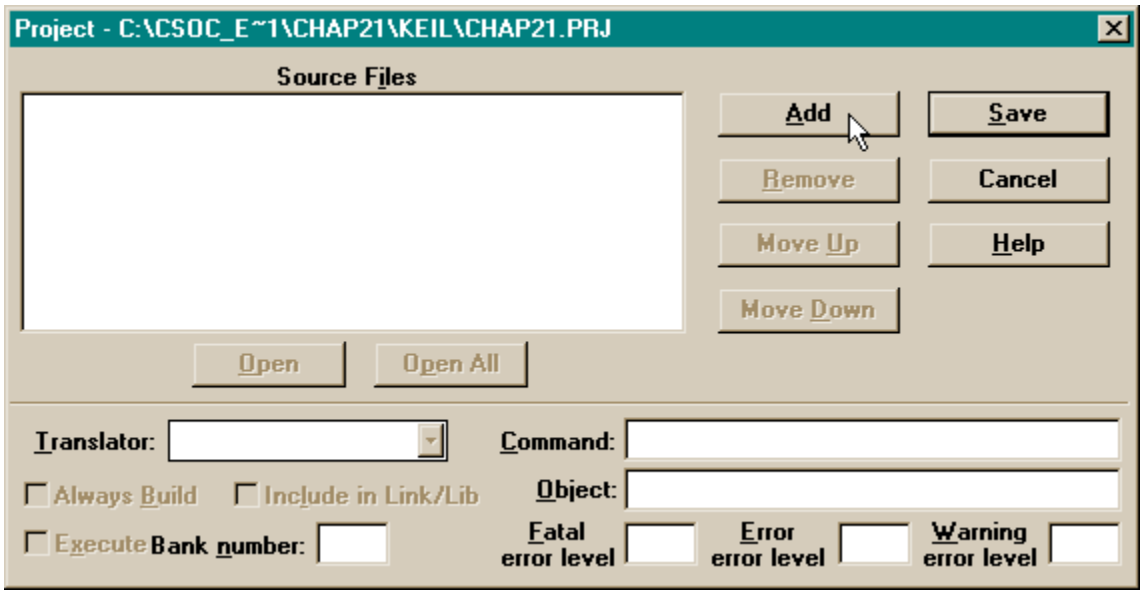
Once your code is saved in the keil folder, you can create a project within the Keil IDE. The project file will store all the settings and source files that are needed to generate the 8032 object code from the C source code. Click on the Project⇒New Project... menu item as shown below.



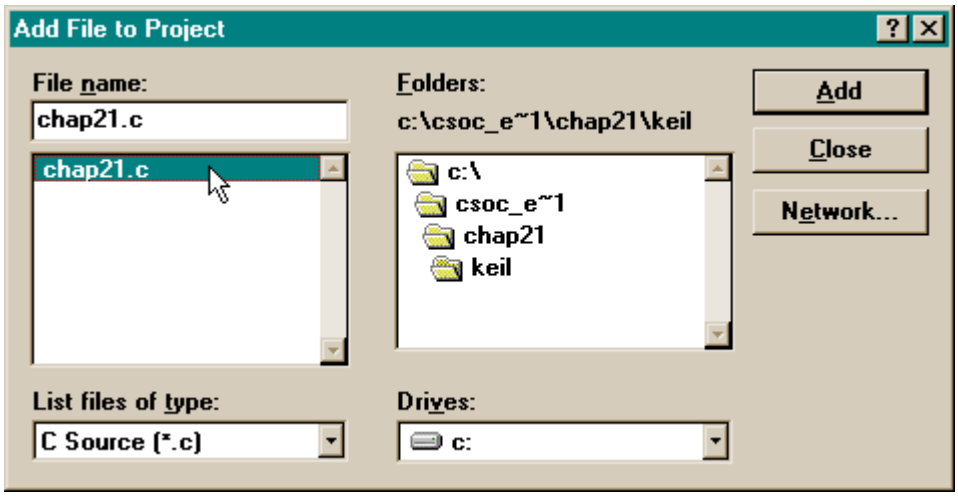
In the **Create New Project** window that appears, save your project in the chap21.prj file in the keil folder.



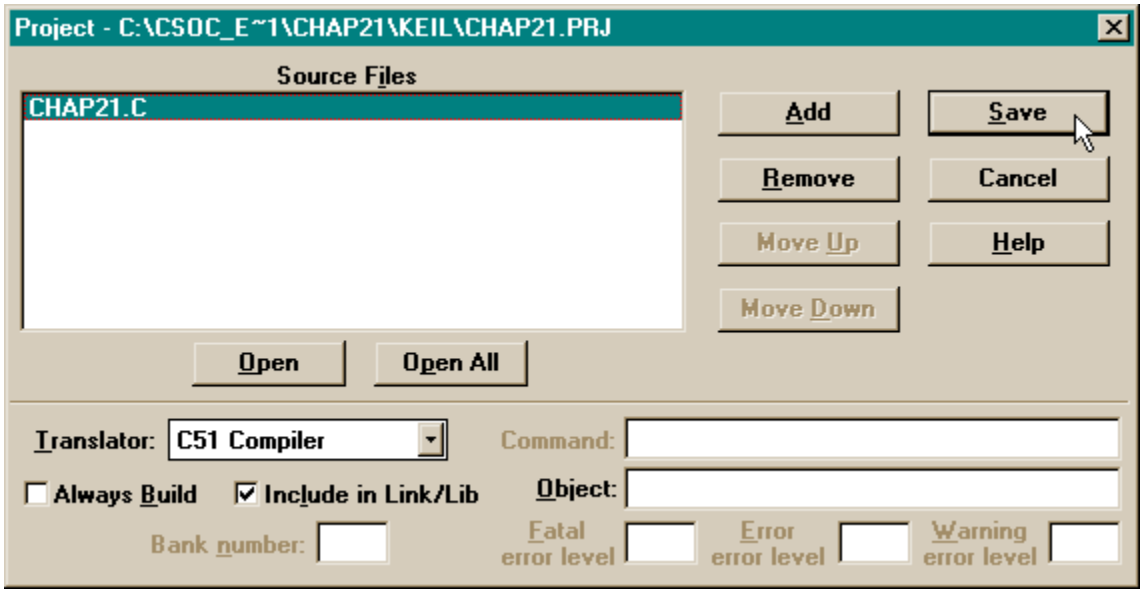
Click on the OK button and the **Project** window will appear. You will use this window to specify the source files that are compiled to create the project object code. This is done by clicking on the Add button.



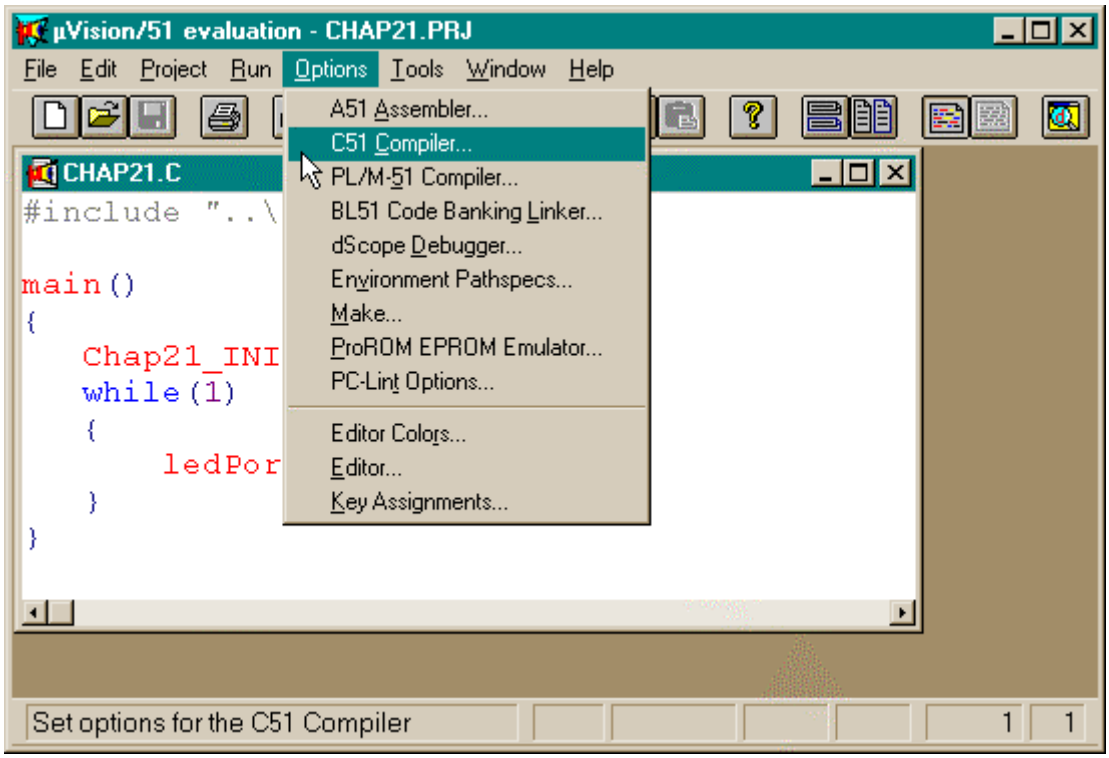
Clicking the Add button brings up the **Add File to Project** window. The C files in the keil folder are shown in the left-hand side of the window. There is only one file in this example, so click on chap21.c to select it and then click on Add.



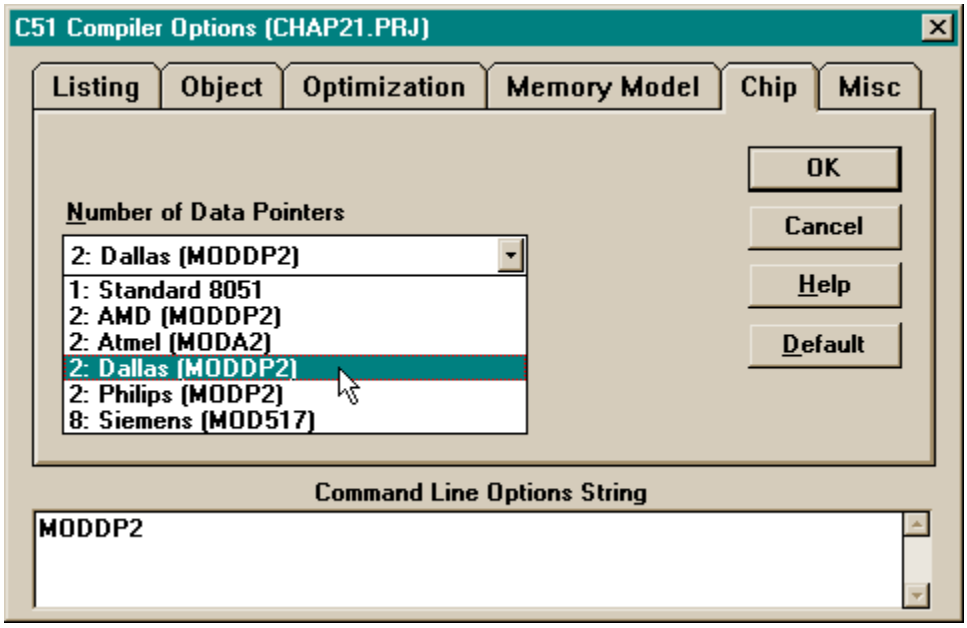
The **Add File to Project** window will disappear and the **Project** window will show that the chap21.c file has been added to your Keil project. This is the only C file needed to create the object code, so click on Save to store the list of source files in the chap21.prj file.



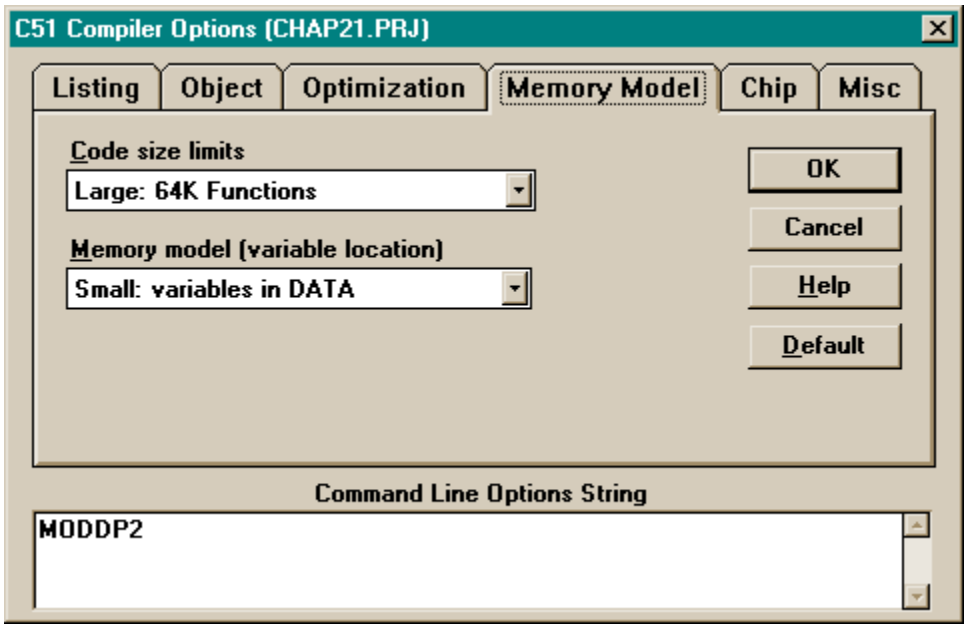
Now you need to tell the Keil compiler and linker tools about the particular features that exist in the Triscend 8032 core. That lets the compiler take advantage of any special features in the hardware and it lets the linker know the organization of the memory spaces. To set-up the compiler, click on the Options⇒C51 Compiler... menu item.



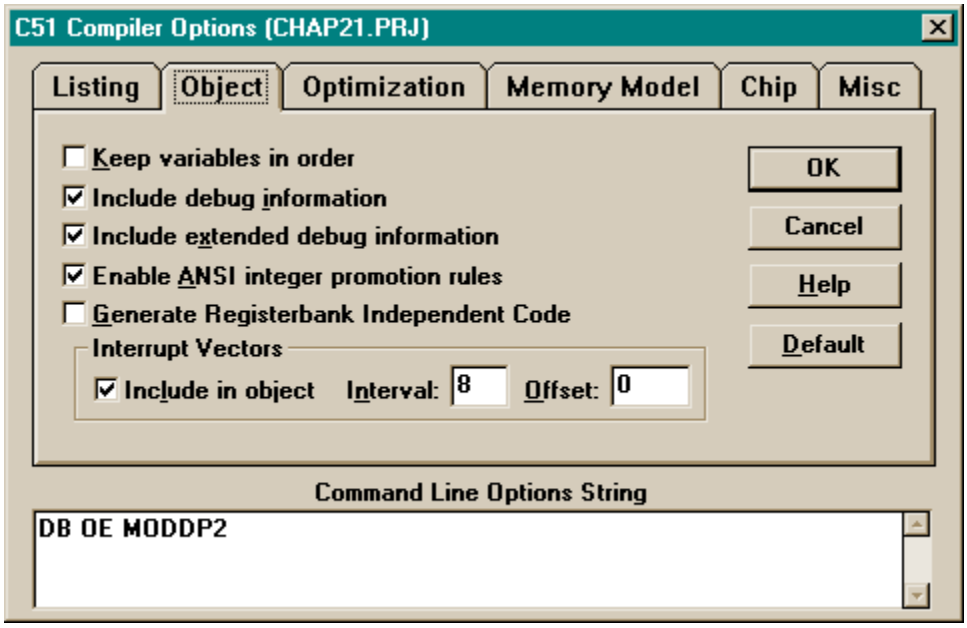
Click on the Chip tab of the **C51 Compiler Options** window in order to specify the number of data pointers in the Triscend 8032 MCU. This MCU is not included in the list, but the Dallas version of the 8051 architecture has two data pointers just like the Triscend 8032 MCU so select that one.



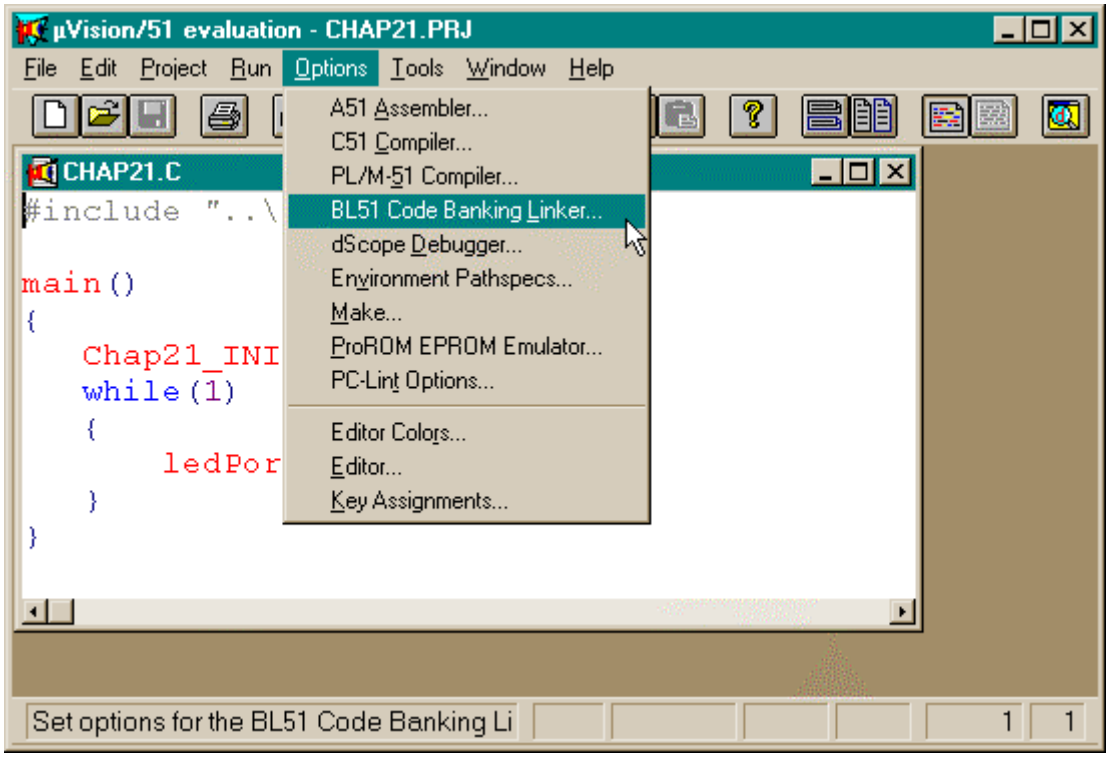
Next, click on the Memory Model tab. Select the Large limit on the code size so programs can use as much as 64 KBytes of memory (even though you won't need that much). Also select the Small memory model which stores program variables in the internal 256-byte scratchpad memory of the 8032 MCU.



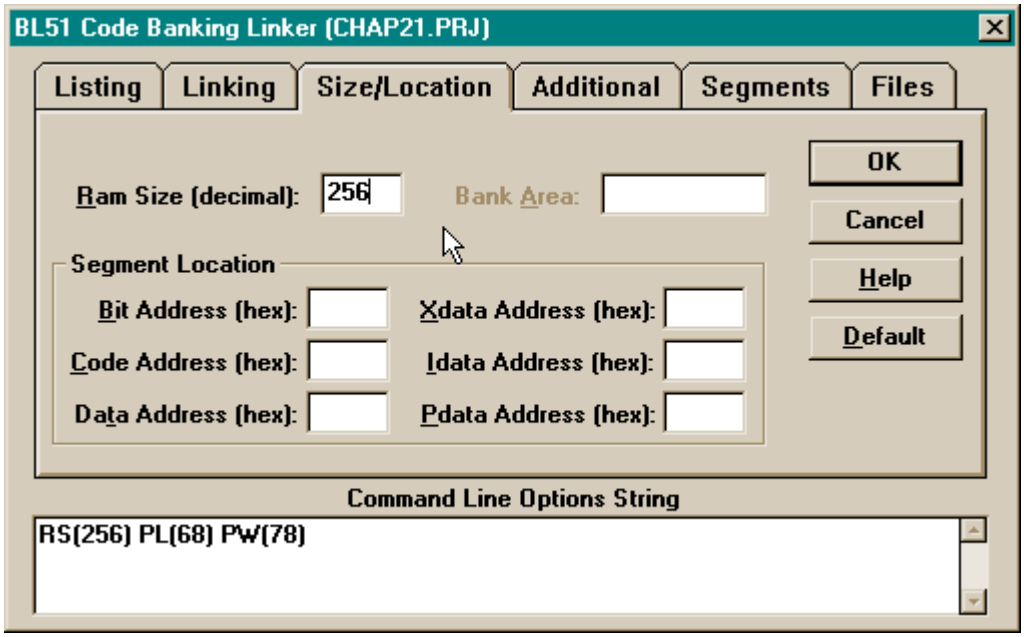
Finally, click on the Object tab and check the folloing boxes to include extra debugging information in the compiled object file. This extra information makes it easier to debug C language programs using the Keil debugging tools. Click on OK to finalize your C51 compiler option settings.



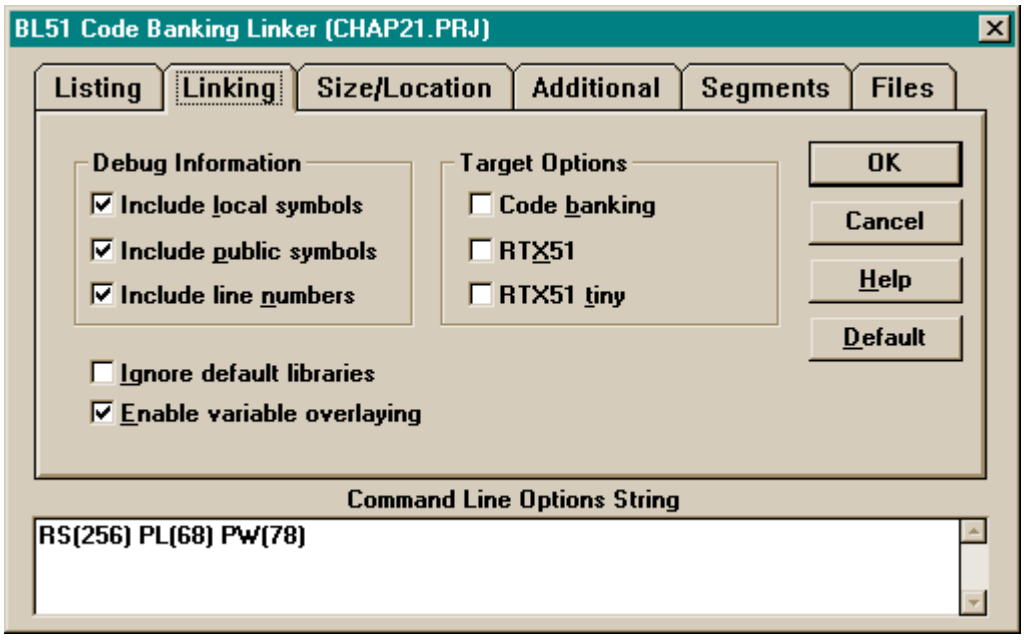
Now set-up the linker by clicking on the Options⇒BL51 Code Banking Linker... menu item.



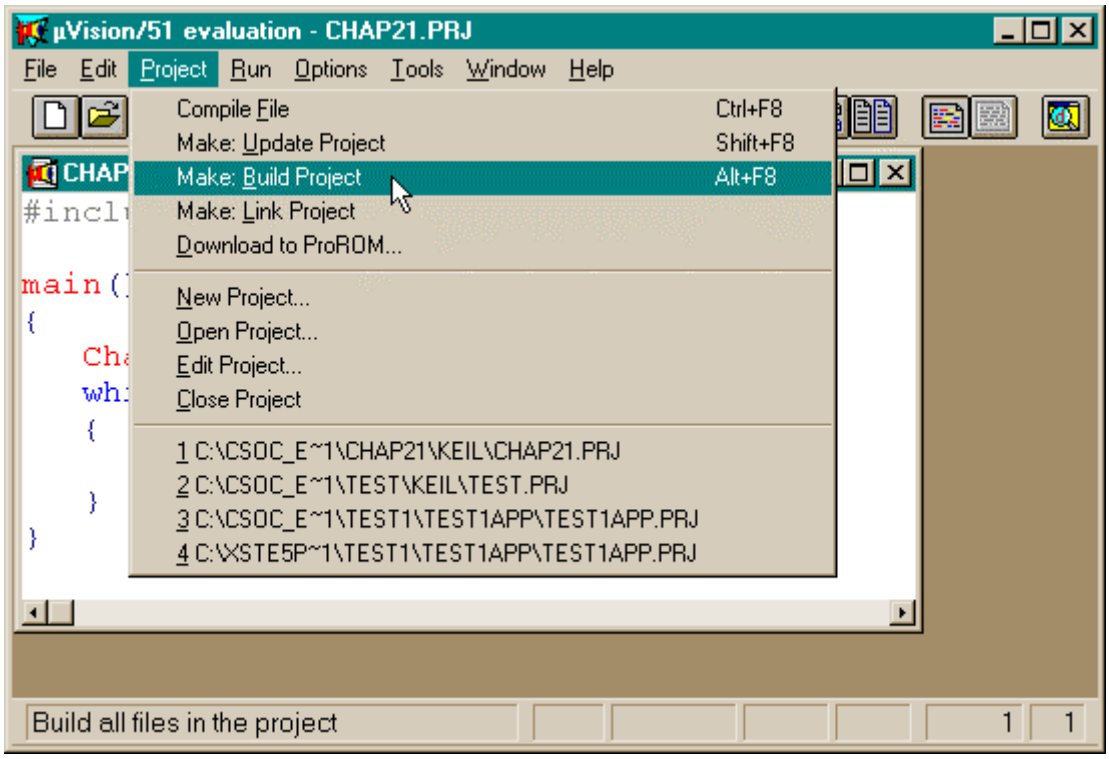
Click on the Size/Location tab in the **BL51 Code Banking Linker** window that appears. The 8032 MCU in the CSoC has 256 bytes of scratchpad RAM for storing variables, so enter 256 into the Ram Size box.



Finally, click on the Linking tab and check all the boxes for including debugging information into the object code. Then click on OK to close this window.



Now that the source files and compiler/linker options are set for your project, click on the Project⇒Make: Build Project menu item. This will compile the chap21.c file and then link it into a HEX file that can be loaded into the Triscend CSoC.



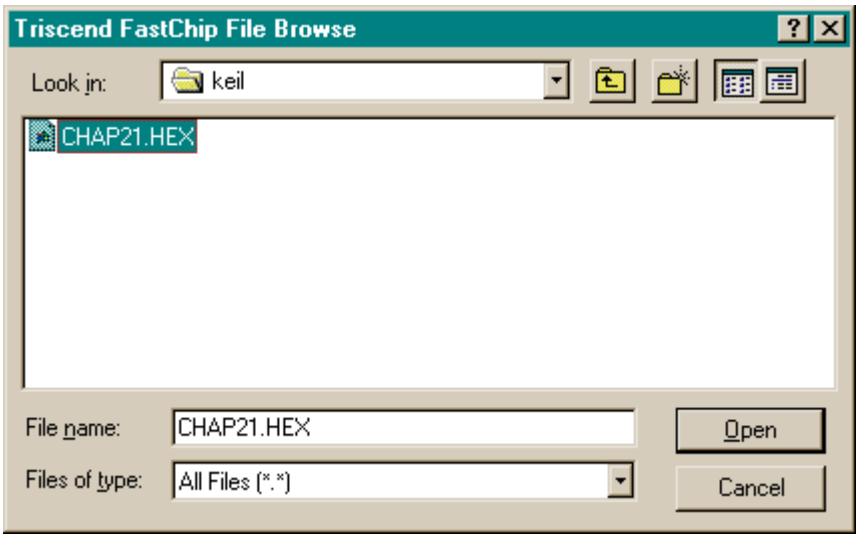
If there are no errors during the compilation or linking processes, you will see a **Project Status** window reporting that the make operation was successful. Otherwise you will get a window listing any errors and their locations in the source code. Click on OK to remove the **Project Status** window.

Binding and Downloading the Hardware and Application Code

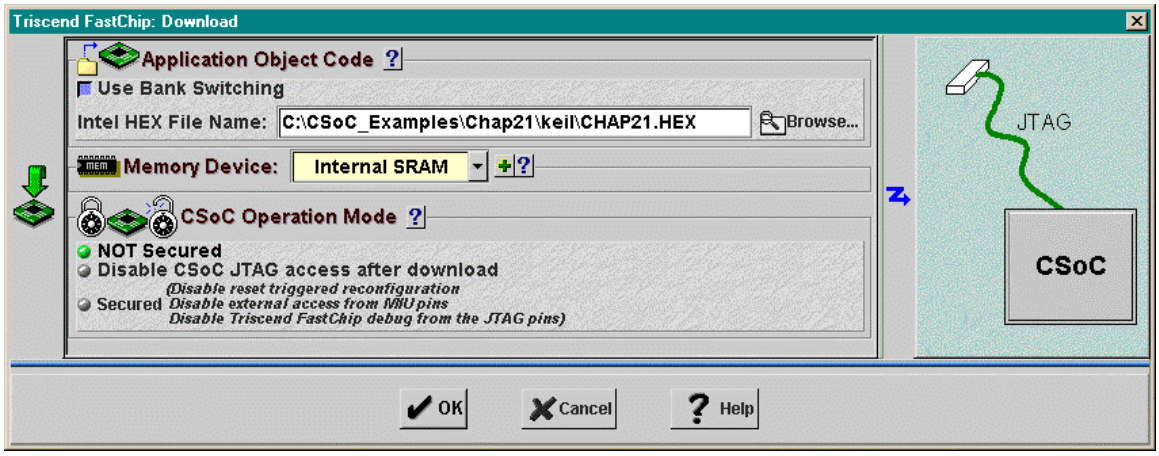
Once the application code has been compiled and linked, you can re-enter the FastChip project window and click on the Bind toolbar icon. This will map your hardware into the

CSL while making sure that the **P0** and **ledPort** registers are assigned the same addresses that are listed in the chap21.h header file.

After the binding operation is complete, click on the Download toolbar icon. In the **Download** window, click on the Browse button in the Application Object Code area. Enter the keil folder using the **Triscend FastChip File Browse** window that appears. You will see the chap21.hex file that was generated by the Keil linker in the previous section. Select the chap21.hex file and click on the Open button.

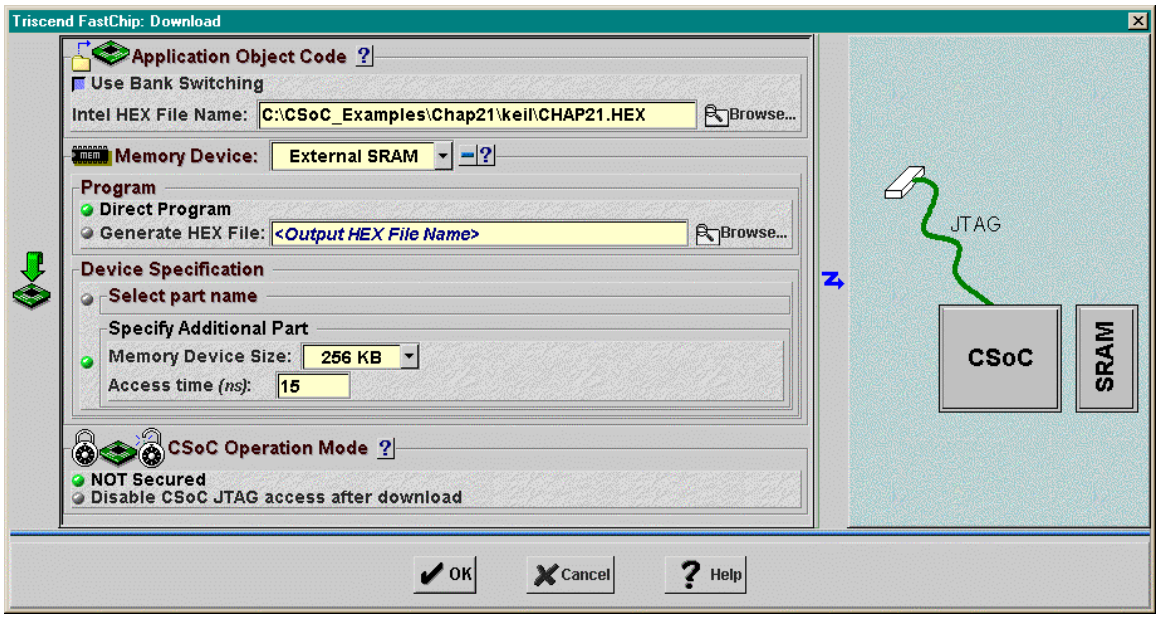


At this point, you will see the chap21.hex file is listed in the Intel HEX File Name box in the **FastChip Download** window. This 8032 object code file will be combined with the configuration file for the CSoC circuitry and the result will be loaded into the CSoC.



Now you have to decide where to load the 8032 object code. You could use the internal 16 KByte SRAM of the TE505 CSoC that exists in the 8032 address range of [0x0000,0x3FFF]. But the linker in the evaluation version of the Keil software tools will only place the 8032 object code at address 0x4000 or higher, so you can't use the internal SRAM to hold the program.

Your CSoC Board also has Flash RAM and SRAM (each stores 128 KBytes) that can hold the object code. The Flash RAM will retain the object code even if the power to your CSoC Board is interrupted. But you are just doing some quick tests on this code and you don't need to store it for a long time. Programming the code into the Flash RAM also takes a minute or so. For these reasons, you should choose External SRAM in the Memory Device area of the **Download** window as shown below.



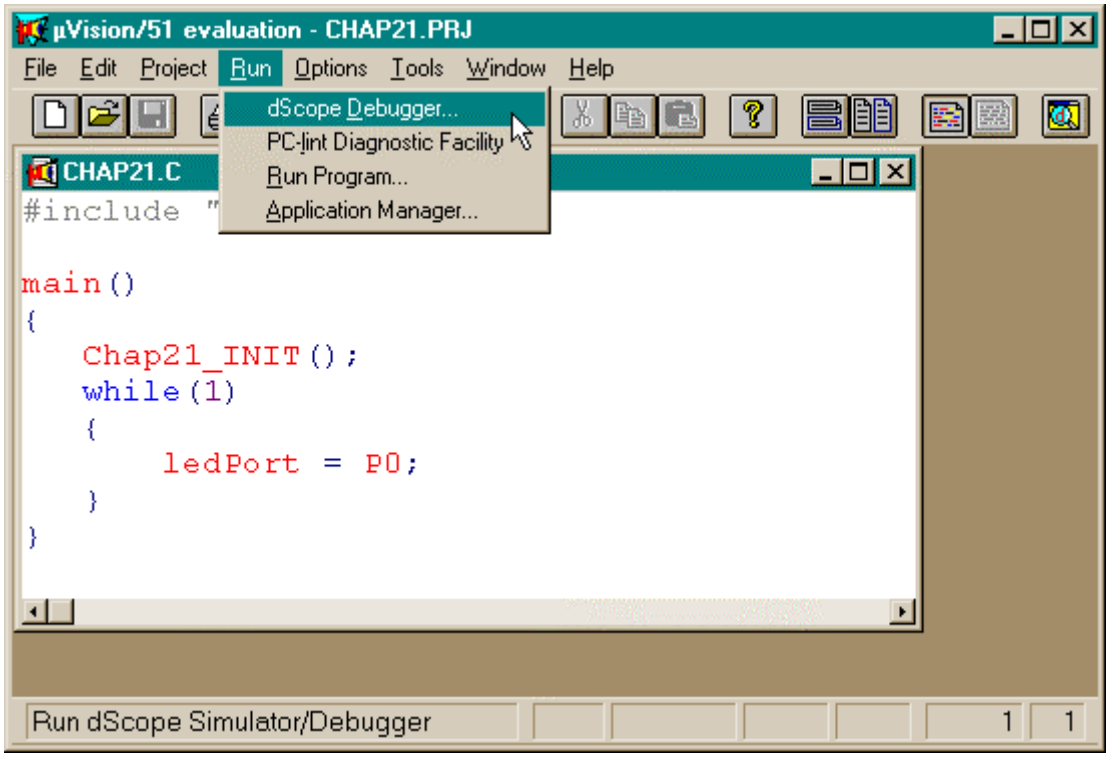
Once you select the external SRAM as the location for your 8032 object code, you have to make a few more decisions. First, you can store the object code and configuration for the CSoC into a HEX file that will be loaded into the CSoC Board SRAM at a later time, or you can elect to download the object code and configuration directly into the CSoC Board through the parallel port download cable. In this example, select the Direct Program option.

Second, you have to give the FastChip some information on the type of external SRAM chip you are using. FastChip knows the characteristics of a small set of SRAM devices, but the SRAM chip on your CSoC Board isn't one of them so you need to use the Specify Additional Part area to list its size and access time. The SRAM chip has 128 KBytes, but the smallest size listed is 256 KBytes so choose that. This will replicate the contents of the SRAM twice within the 256 KByte address range, but that won't cause a problem. The access time for the CSoC Board SRAM chip is 15 ns so type that in as well.

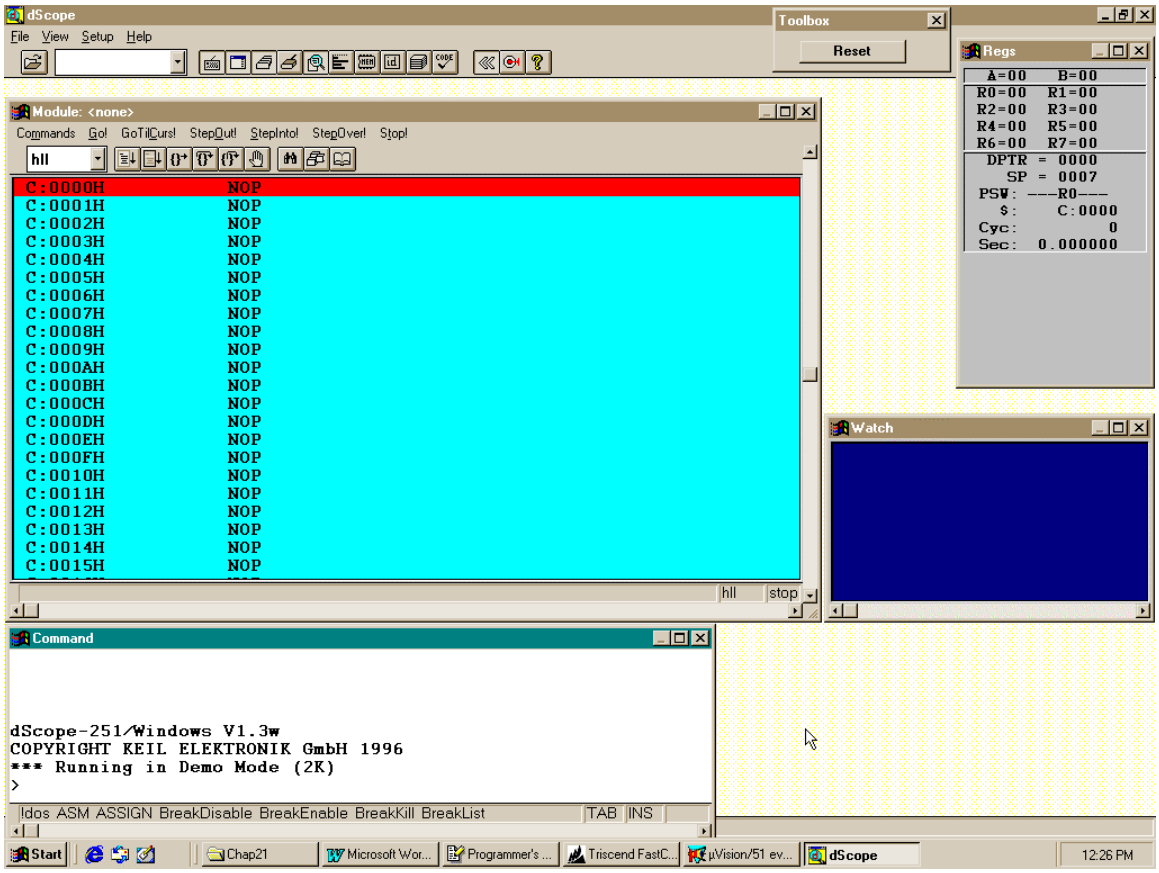
Now click on OK in the **Download** window and the 8032 object code and the CSoC configuration will download into your CSoC Board through the parallel port. *As always, . make sure your CSoC Board is attached to the 9V DC power supply and it is connected to the parallel port of your PC with the downloading cable. You should also set the shunts on jumpers J8 and J9 of the CSoC Board so that the SRAM chip-enable input is connected to the CSoC MIU chip-enable output (see Appendix A2).*

Testing Your Application Code

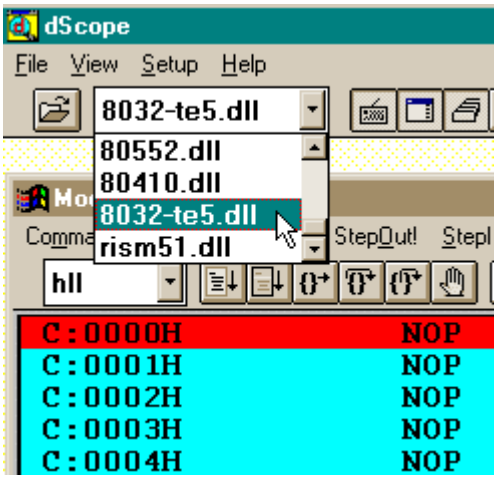
Now the CSoC Board is loaded with your circuit and application code. You will use the Keil dScope debugging tool to test the code. In the Keil IDE, click on the Run⇒dScope Debugger... menu item to start the debugger.



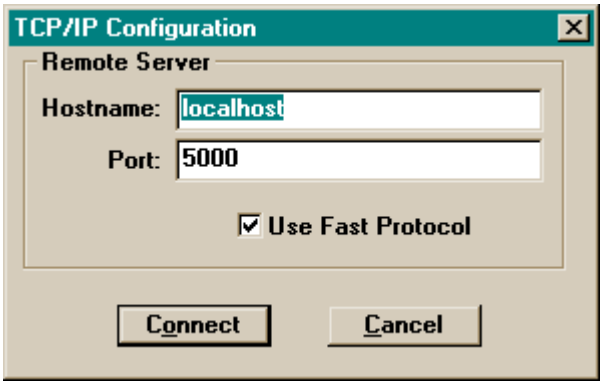
The **dScope** window will appear with several subwindows that display MCU register values and object code as well as those which allow you to reset the MCU and enter other commands.



The first thing you need to do is inform the debugger of the type of MCU you are working with. In the drop-down list near the top of the **dScope** window, select the 8032-te5.dll entry. This installs a library of routines that interface the dScope debugger to the TE505 8032 MCU and hardware breakpoint unit through the JTAG port.

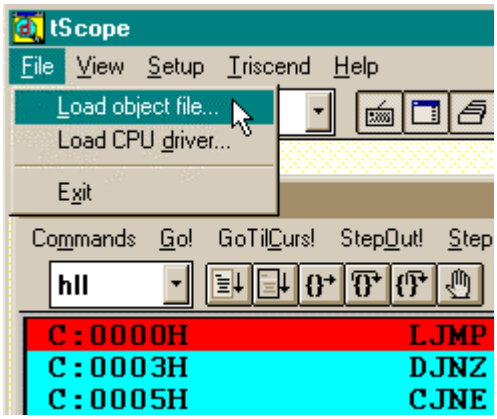


Once you select the MCU interface, the dScope debugger will initiate a connection with the CSoC Board. Just click on OK in the **TCP/IP Configuration** window that appears.

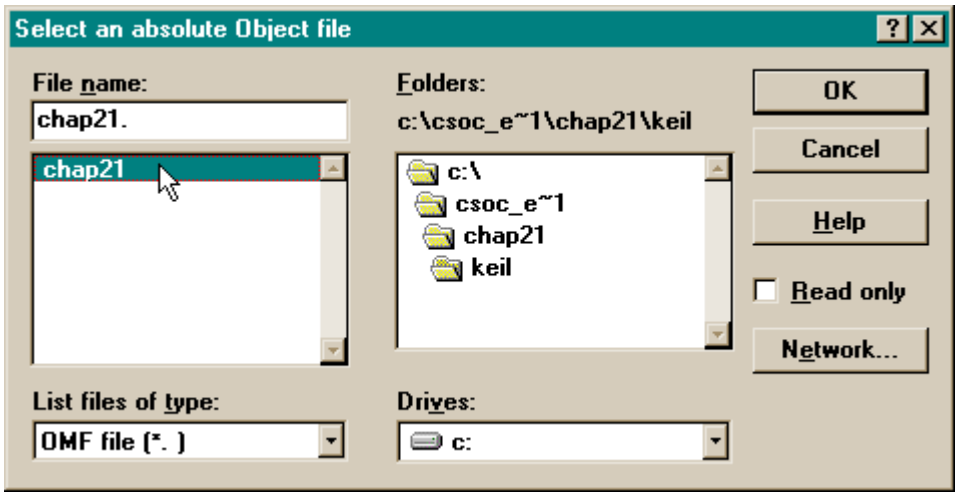


After the connection is established, you should reset the 8032 MCU in the CSoC by clicking on the Reset button in the **Toolbox** window. (If the **Toolbox** window is not visible, click on the  button in the **dScope** window.)

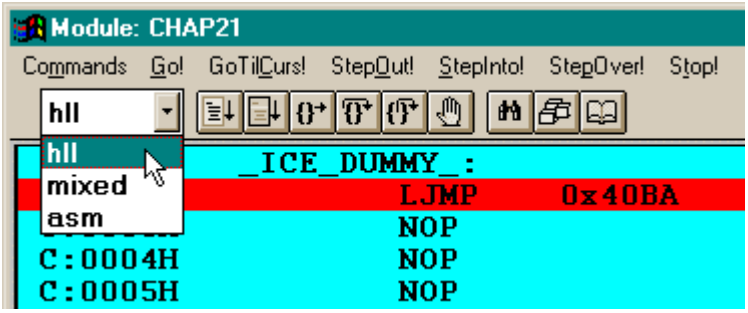
Once the MCU is reset, you should select File⇒Load object file... to load the debugging information for your application into dScope. (This does not initiate another download to the CSoC Board.)



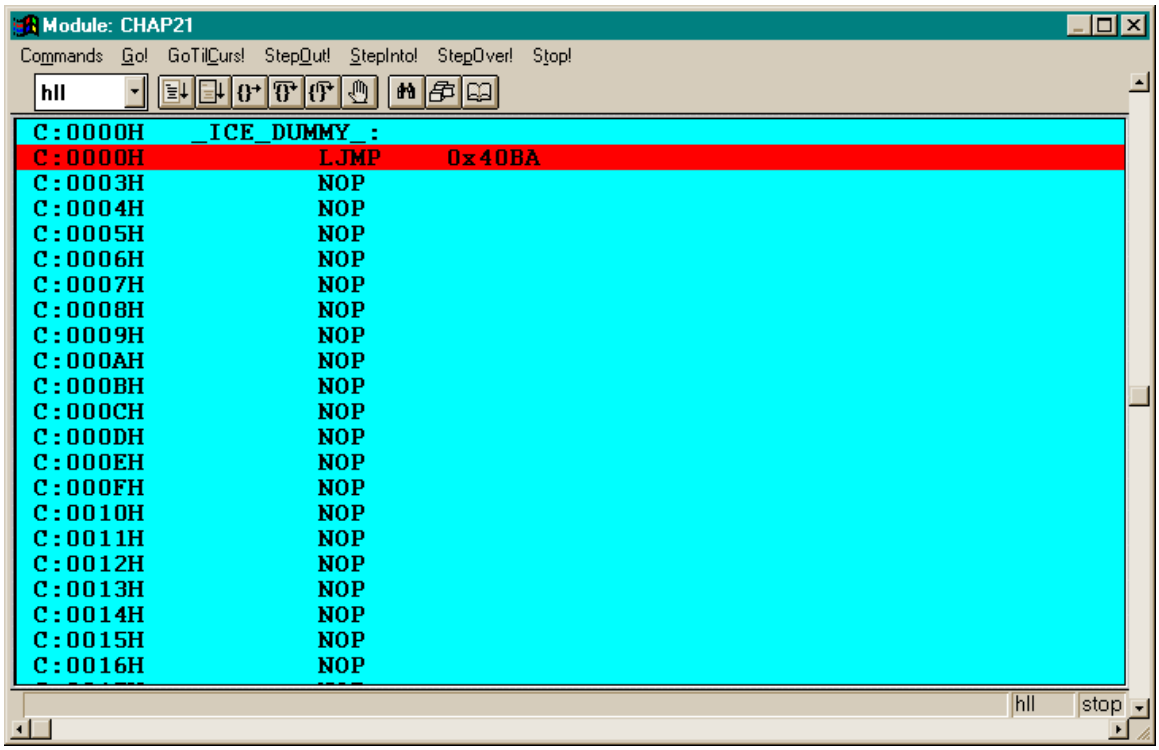
A window will appear that you will use to select the file with the debugging information. Go into the keil folder within your FastChip project folder and select the chap21 OMF file (this file does not actually have any suffix). Click on OK and the debugging information is loaded into dScope.



With the debugging information now available in dScope, you have the choice of viewing your application software as assembly language, C high-level language, or a mixed mode which shows each line of C code and its associated assembly language instructions. Tracing through the your code's assembly language as it executes gives you the most detailed view of what it is doing, but this level of detail is overkill for this application. Instead, select the hll option from the drop-down list in the **Module** window so you can view your code as C language statements.

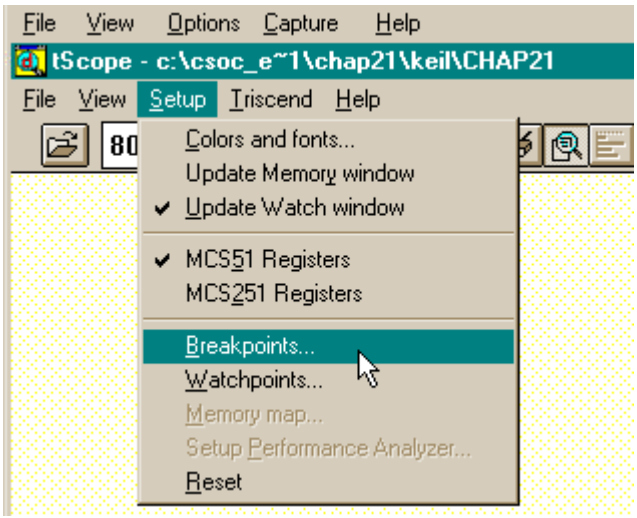


Now your Module window looks like the one below. Where is the C code? The **Module** window shows code starting at the beginning of memory, but your C code starts at address 0x4000 so it isn't visible.

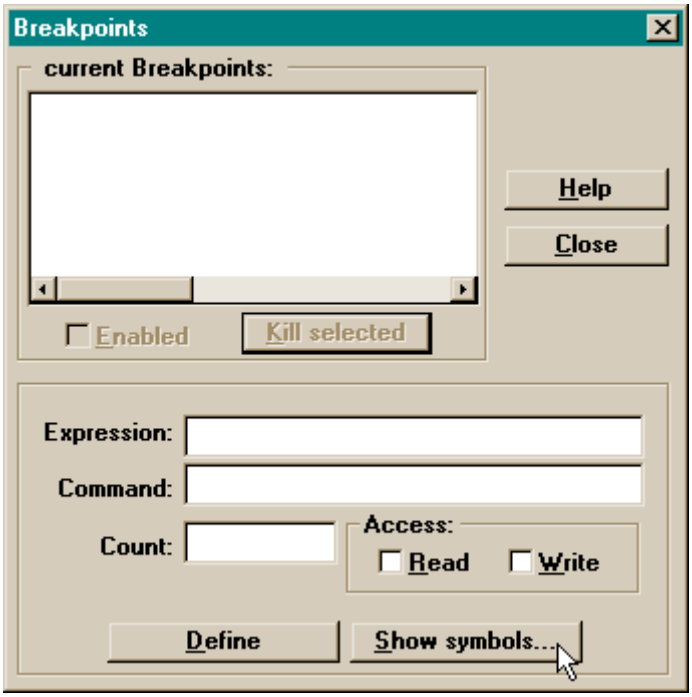


There are two ways to get to the beginning of your C code. The first way is to click on the slider bar on the right side of the Module window and slide it upwards until you see Address = C:4000H at the bottom of the window. Release the slider and you may see your C code displayed in the window. I say "may" because sometines this works for me, and other times it doesn't.

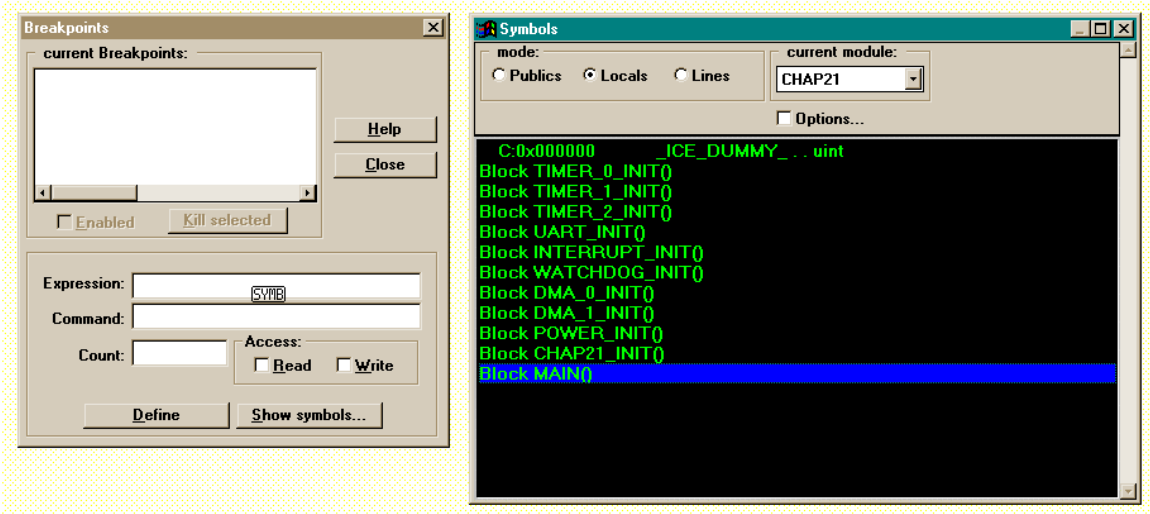
The second way to get to the start of your C code is more reliable, but takes a few more steps. It involves setting a breakpoint at the start of the `main()` routine and then letting the application program execute until it hits that breakpoint. This causes the program to stop execution at the beginning of `main()` and the C code at that point will appear in the **Module** window. To set the breakpoint, click on Setup⇒Breakpoints... in the menu bar of the **dScope** window.



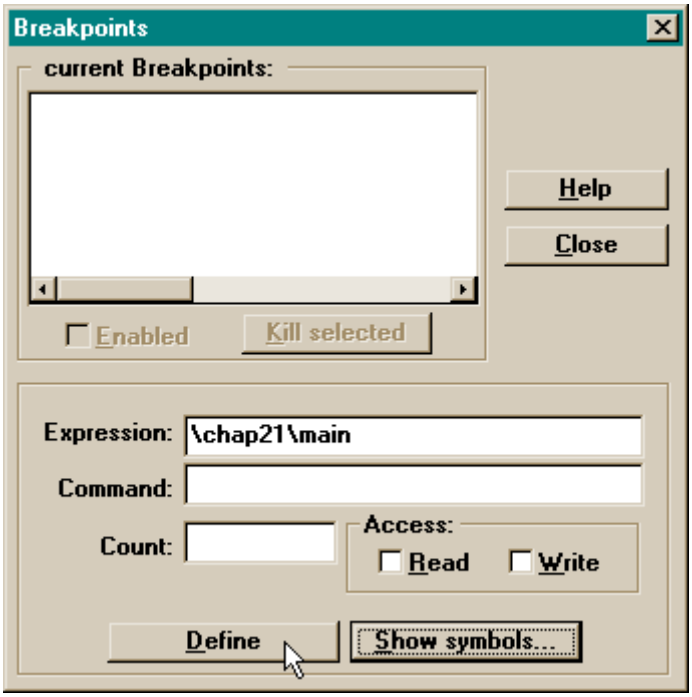
The **Breakpoints** window will appear. It has an area which shows the current set of breakpoints defined for your program (which should be empty at this time). Below that is an Expression box where you can enter an expression that defines an address within your program. This address is loaded into the hardware breakpoint unit of your CSoC. The 8032 MCU will be halted when the breakpoint unit sees a combination of read and/or write accesses to that address which equals the number entered into the Count field in the **Breakpoints** window.



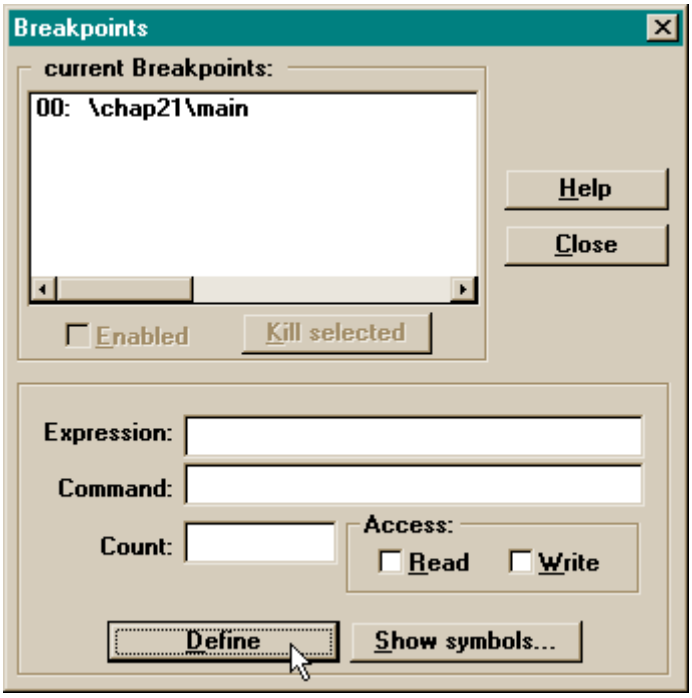
But how do you write an expression for the beginning address of the `main()` routine? It turns out you don't have to. Just click on the Show symbols... button in the **Breakpoints** window. This brings up the **Symbols** window where you can see all the variable and subroutine names for your program. Click on the Locals radio button and a list of the subroutines will appear. Click on the MAIN() line in the **Symbols** window and drag it into the Expression box in the **Breakpoints** window.



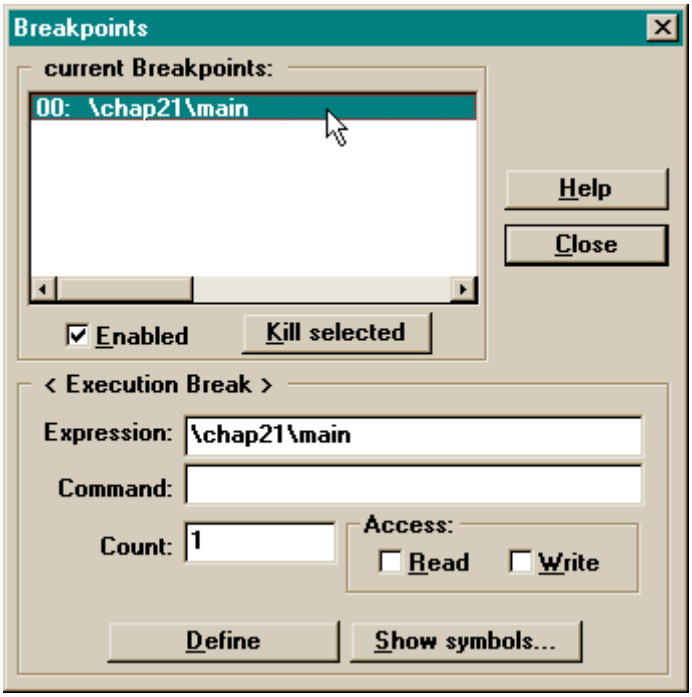
After you drop the MAIN() symbol into the **Breakpoints** window, it will appear in the Expression box.



Next, click on the Define button and the symbolic address in the Expression box will be converted to a breakpoint that appears in the current Breakpoints area.

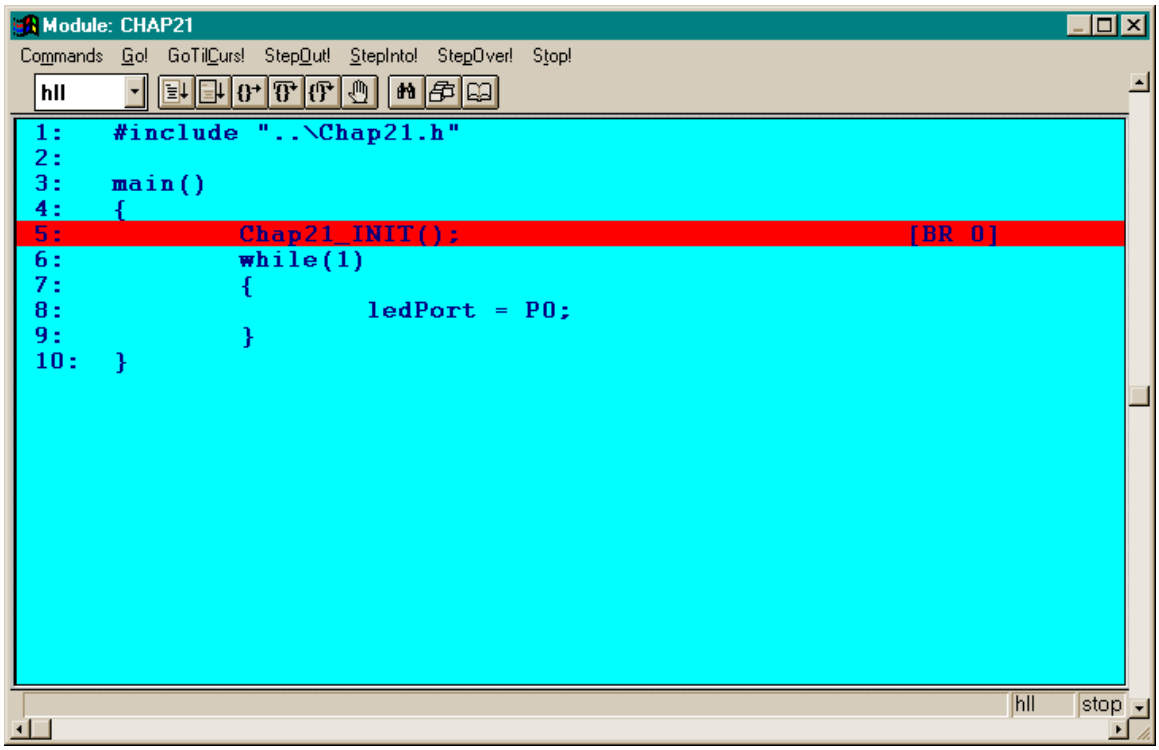


At this point you have a single breakpoint, but you can have more than one in the list. If you want to examine or change a breakpoint, just select it in the list as shown below. You can enable or disable the checkpoint using the Enabled checkbox. Or you can delete the breakpoint completely by clicking on the Kill selected button. You can also increase or decrease the number of times the address is accessed before the breakpoint becomes active by changing the value in the Count box. Or you can use the Command box to enter a command in a C-like syntax that will be executed when the breakpoint occurs. But you don't need to change anything about this breakpoint right now. Just click on Close to accept the current breakpoint and remove the window.



Now the application code is loaded into the 8032 MCU and the debugger, the MCU is reset, and the breakpoint at the start of `main()` is defined. You can start the MCU program running by clicking on Go! or the  button in the Module window. The 8032 will start to run, but it will immediately stop as soon as it gets to the beginning of the `main()` routine.

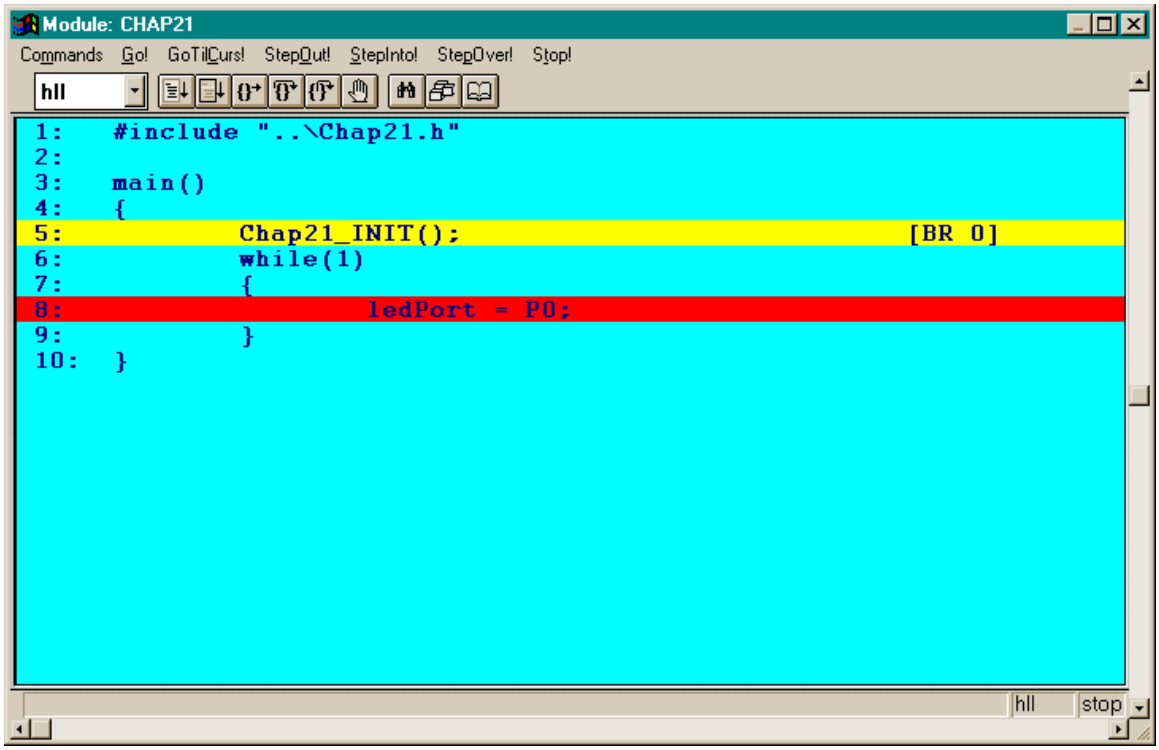
The **Module** window now shows the C code for your program. The next C statement that will be executed is highlighted in red. This is a call to the subroutine that initializes the various 8032 peripherals.



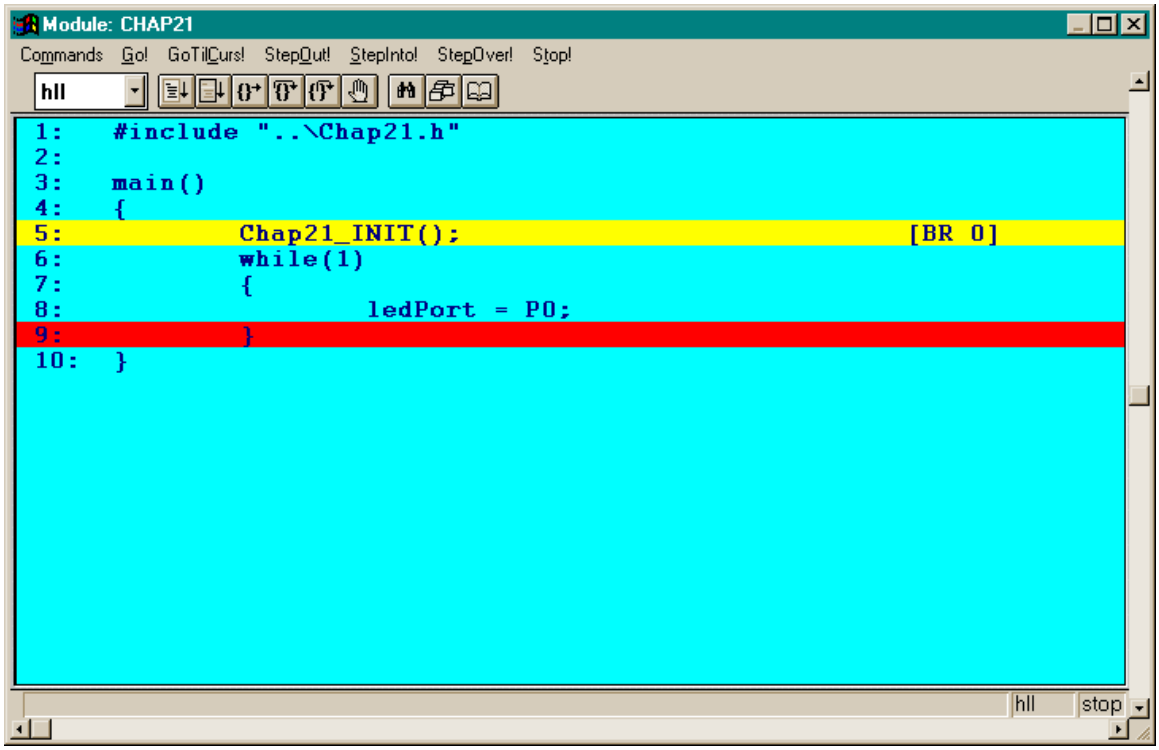
Now you can execute the C code statement-by-statement and see how the program works. The StepOver! command or the  button is used to execute a single C statement. In this case the next statement is a subroutine call, so activating StepOver! will enter the subroutine, execute all the statements in the subroutine, and then return to the next statement in the calling routine and halt.

If you want to see the statements within the `Chap21_INIT()` subroutine as they are executed, use the StepInto! command or the  button to drop down into the subroutine. Then use the StepOver! command to trace the execution of statements at that level, or move deeper into the nested subroutines by using the StepInto! command. When you trace to the end of a subroutine, you will automatically return to the next statement following the subroutine call in the calling routine. If you have seen all you need to in the current subroutine, you can issue the StepOut! command or press the  button and the rest of the statements in the subroutine will be executed, control will return to the calling routine, and program execution will stop.

After the `Chap21_INIT()` subroutine completes, the program enters the `while` loop. The next statement that will be executed will read a value from the **P0** port and transfer it to the **ledPort** register. At this point, you should change some of the DIP switch settings so you can see if the pattern on the LED digit changes.



Clicking the StepOver! command or the  button now moves the execution to the end of the `while` loop. You should see the LED digit change to reflect the DIP switch setting as shown in the examples in Figure 13.



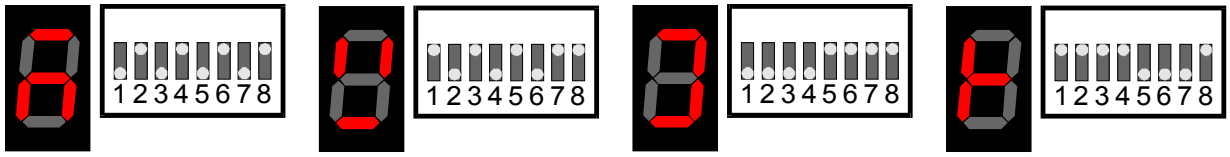


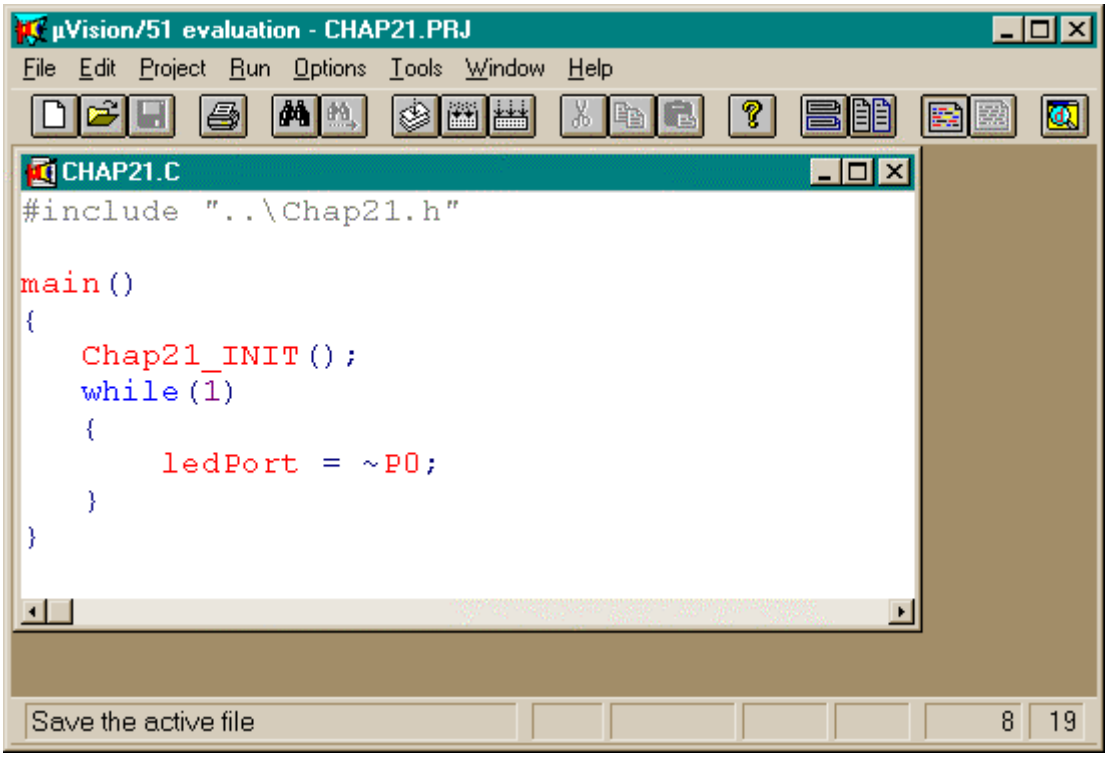
Figure 13: The LED digit activation for several DIP switch settings.

At this point, you have to click the  button twice before the LED digit segments change to reflect the DIP switch settings. If you want your program to run at full speed without interruption, just press the Go! command or the  button. The program will not return to the start of the `main()` routine so it will never hit the breakpoint and halt. Once the program starts running, any changes to the DIP switches will appear instantaneously on the LED digit. To halt the program, just click on the Stop! or  button.

Changing Your Application Code

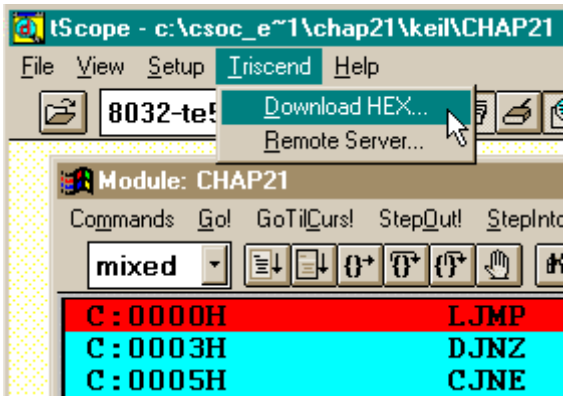
What if you discover an error and want to correct your application code? For example, suppose you want the LED segments to light up when their associated DIP switch is in

the up position instead of the down position. This is easy to handle in the C source code: just add an inversion operator (~) before the `P0` variable. This makes the statement read the DIP switch settings from port **P0**, do a bitwise inversion of the value, and then store the altered value into the **ledPort** register.

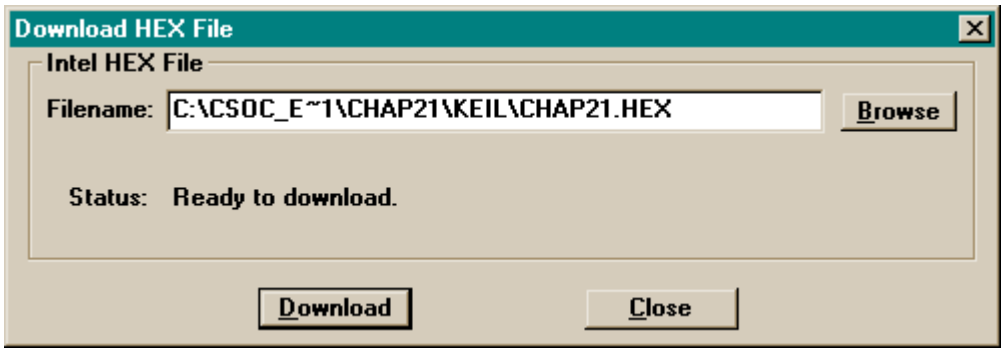


Now you have to recompile your program to reflect this change. Select Project⇒Make: Update Project or click on the  button in the **µVision/51** window. The C source code is recompiled and linked to create a new chap21.hex file.

With the updated HEX file in hand, the next step is to load it into your CSoC Board. You could use the downloading features of the FastChip software, but the dScope debugger also has this capability. First, you should halt any program that is running in the 8032 MCU by pressing the Stop! command in the **Module** window. Then in the dScope window, click on the Triscend⇒Download Hex... menu item as shown below.



This brings up a **Download HEX file** window where you can specify the name or browse to the location of the updated HEX file. Click on Download to reload the CSoC with the updated application code.



After the CSoC has been updated, you still need to load the debugging info for the new source program into dScope. Do this by clicking on the File⇒Load Object File... menu item and select the chap21 OMF file.

Once the updated debugging information is loaded, click on the Reset button in the **Toolbox** window and then click on Go! in the **Module** window. With the program running, you should see that the DIP switch buttons have the opposite effect on the activation of the LED digit segments.

After verifying the operation of your modified program, click Stop! to halt the program. Then select File⇒Exit in the dScope window to close the debugger. You can also exit from the Keil IDE and the FastChip program.

Design 2.2 - UART Loopback

Your next MCU-based CSoC design will use the UART in the 8032 to transmit and receive bytes of test data (Figure 14). The MCU will continuously load the UART transmitter and poll the receiver to see if the received byte is identical to the transmitted byte. An AND gate will be placed in the loopback path so the transmitted byte can be blocked or passed through based on the setting of DIP switch 1. The MCU will display an 0 on the LED digit as long as the loopback path is not broken (**gate** = 1). When the path is broken (**gate** = 0), the transmitted and received bytes will no longer match and the MCU will display an E.

