Product User Guide

Z8051 Tools

PUG003305-1112

Introduction

The Z8051 family of microcontrollers is a new Zilog product line based on the 8051 microprocessor architecture. Unlike other Zilog products, proprietary Zilog development tools to support the Z8051 family are not provided, due to the 8051 processor core's being well-supported by a large assortment of development tools from third-party tools suppliers. Among these suppliers, Zilog recommends the free open-source tools from the Small Device C Compiler (SDCC) software project and the Keil tools from Keil Software. Customers can also choose to work with other excellent third-party tools that are available for the 8051 architecture.

Because these development tools are not provided by Zilog, and because their use may be unfamiliar to Zilog customers, this product user guide is provided to help users get started using these tools in this potentially new development environment. Both the SDCC and Keil tools are detailed in their own documentation, which customers should consult as the ultimate source of information about these tools. Zilog assumes no responsibility for the functioning, performance or safety of these third-party tools.

First Steps With Z8051 Tools

To get started working with the Z8051 for the first time, a good place to begin is with the user manual for the Z8051 development kit you are working with. Each of these manuals offers a brief, step-by-step guide to building your first sample Z8051 project in their respective *Build and Run the [sic] Demo Project* sections, as well as general instructions for installing and setting up the Z8051 software. In this product user guide, it is assumed that you have successfully set up the software and built your demo project as described in these manuals.

Goal Of This Document

The purpose of this product user guide is to provide enough additional information so that you can begin to develop your own applications, learning the basics of the SDCC and/or Keil tools along the way. This document discusses the following topics:

- Header files provided by Zilog to help you write code for Z8051 peripheral hardware and to resolve code portability issues
- Memory map issues you must confront and how to solve them
- Getting these tools to deal properly with interrupt service routines
- How to build an application using the SDCC tools

- Why you must also work with the Zilog OCD Debug Tool, and how this requirement affects the build process
- How to build an application using the Keil tools
- Where to find further information about the SDCC and Keil tools

Conventions Used In This Document

Special fonts are used throughout this document to more easily distinguish normal text from file names, source code terms, and similar elements. Throughout this document, the names of all files, folders and paths are displayed in the `Courier New` typeface, as are all listings of source code or command files, example code segments parsed from such files, source code entities including keywords (such as `int` or `void`), variable names, and expressions in source code.

At times, a shorthand notation for a sequence of selections in a graphical user interface (GUI) is displayed in a bold typeface. Therefore, **Project** → **Options** → **Device** → **Database** means you would first select the **Project** menu item, next select **Options** from the available choices in the Project menu, then select the **Device** submenu item, and finally select **Database**.

Writing Your Code: Notes About Header Files

Before building your application, you must write the code, for which you can use any editing tool of your choice. Zilog provides several header files that should be very useful to you and save you time at developing your application. These header files will make it easier and less error-prone to write code for the Z8051 family's peripheral devices, as described in the remainder of this section. They will also take care of some portability issues for you. Zilog encourages you to include these header files in your source code.

These header files can be found in the Zilog software installation contained in the following path:

```
<Z8051 Install>\include
```

In this path, `<Z8051 Install>` is the location on your hard drive in which you installed the Z8051 software. The `C` subdirectory at this location contains C header files, and assembler header files are included in the `asm` subdirectory.

The following types of header files are provided:

- Family-specific C header files that define special function registers (SFRs) and interrupt numbers
- C header files that fix compiler compatibility issues
- Family-specific assembler include files that define the SFRs

Family-Specific C Header Files

The Z8051 product line includes a number of different families of microcontrollers, each defined by common SFRs & interrupts, and each oriented toward different types of applications. For each of these families, a unique Zilog-provided C header file defines the names and addresses of its SFRs, and also provides symbolic names for its interrupt numbers. For example, the `z51f0811.h` file provides these definitions for the Z51F0811 microcontroller family; this file is contained in the following path:

```
<Z8051 Install>\include\C\z51f0811.h
```

By including this `z51f0811.h` file in your C code, you can work with the names of the SFRs and interrupts rather than the raw addresses or numbers associated with them. As a result, your code will be easier to read and less prone to errors.

The use of these family-specific header files also makes your code more portable, in two senses. First, you can more easily move an application from one Z8051 microcontroller to one in another family. The new target family may place an SFR that both families have in common at a different address, and assign different interrupt numbers to a common interrupt source. However, by simply changing which header file you include in your code, all of these details are taken care of. Your code will not only compile, but more importantly, it will run correctly on the new part.

Use of the Zilog family-specific header files also makes your code more portable between the SDCC and Keil compilers because you can begin your project using the SDCC tools, which are free and therefore a good choice for preliminary exploration. If you should switch to the Keil tools at a future date, your SFRs will continue to be properly defined. This process does not occur automatically, because while both the SDCC and Keil compilers support a special class of C variables called SFRs, (which have the property of being assigned a fixed address), the two compilers each use a different syntax.

Working Around Compiler Compatibility Issues

This difficulty of incompatible syntax is handled by a header file called `compiler.h`, which is part of the SDCC distribution but works equally well when used with the Keil tools (that, indeed, is its purpose: to make your code more compiler-independent). For convenience, we have placed an extra copy of the `compiler.h` file into the following path so that you are not required to add another *include* path in your project.

```
<Z8051 Install>\include\C
```

The `compiler.h` header file takes an SFR declaration in a particular, preprocessor macro format and translates it into the appropriate syntax for either the Keil or SDCC Compiler. The Zilog family-specific header files (e.g., `z51f0811.h`) define the SFRs using this macro format, and therefore do not incorporate the legal syntax for either the SDCC or Keil compilers, unless the `compiler.h` file has previously been included.

Upon inspection of the `compiler.h` file, either the `SDCC` preprocessor symbol or `__CX51__` for the Keil tools must be defined to select the proper definitions. The SDCC Compiler automatically defines the `SDCC` symbol, but when using the Keil tools, you must ensure that `__CX51__` is defined in the **Target Options → C51 → Preprocessor Symbols** setting. This symbol contains a double underscore (__) at its beginning and end; CX51 or C51 are Keil's names for its 8051 C Compiler.

Another useful header file, `ExtKeywords.h`, is also located in the path noted above. Its purpose is, again, to provide portability between the Keil and SDCC tool chains. These two compilers each add a number of extension keywords to the C language, primarily to define specific address spaces used in the Z8051 architecture such as `idata`, `xdata`, and similar spaces. Additional keywords apply specifically to the 8051 CPU, such as `using`, `reentrant`, and `interrupt`. Keil and SDCC use very similar, but not identical, names for these keywords. The `ExtKeywords.h` file automatically translates each keyword to its correct form for the compiler you are using.

For added convenience, the Zilog header file, `portable.h`, includes both `<compiler.h>` and `<ExtKeywords.h>`, as well as some useful bit definitions and a few other definitions of general utility. This file is located in the following path:

```
<Z8051 Install>\include\C\portable.h
```

You can address all of these issues of SFR definitions, compiler compatibility and cross-family portability by simply adding the following two lines to each C source code file:

```
#include <portable.h>
#include <z51f0811.h>         // or appropriate family
```

Assembler Include Files

Finally, we should briefly discuss the assembler include or header files which are located in the following path:

```
<Z8051 Install>\include\asm
```

Two subdirectories are contained in the `asm` directory: `Keil` and `SDCC`. In each of these subdirectories, a set of `.inc` files are specific to each of the Z8051 microcontroller families. These files apply the SFR definitions, just like the C header files do, but use a syntax that is appropriate for the Keil or SDCC assemblers (as opposed to the compilers). These `.inc` files can be included in your assembly code, as required.

Controlling The Memory Map

There are two issues related to your application's memory map that the build tools do not manage automatically. You must explicitly set up the build parameters so that these memory map requirements are respected in the build process. These issues must be managed no matter which tool chain you use, though the details of how this management task is applied are different between the SDCC and Keil tools. This section describes the issues and their solutions in general terms. These issues are:

- Vector table allocation

- Application-specific hardware memory mapping

Vector Table Allocation

All 8051 processors reserve a portion of program space for an interrupt vector table, which by default begins at address `0x00`. Both the Keil and SDCC compilers place entries in this vector table for interrupt service routines (ISRs) defined with the `interrupt` or `__interrupt` keyword. Because most applications will only use a fraction of all of the interrupt sources defined for the MCU they are using, gaps in the vector table are likely. Additionally, the vectors that are actually used may not extend to the end of the memory space that is, in principle, set aside for the vector table.

As a concrete example, consider an application for the Z51F0811. This MCU defines 31 interrupt numbers whose vectors are at addresses `0x00` to `0xFB`. Allowing 3-byte addresses for all interrupt vectors including the last one, this vector table would occupy addresses `0x00` to `0xFD`, and executable code could begin at address `0xFE`.

Note: Each Z8051 MCU product specification provides a detailed interrupt vector table in its Interrupt Controller chapter.

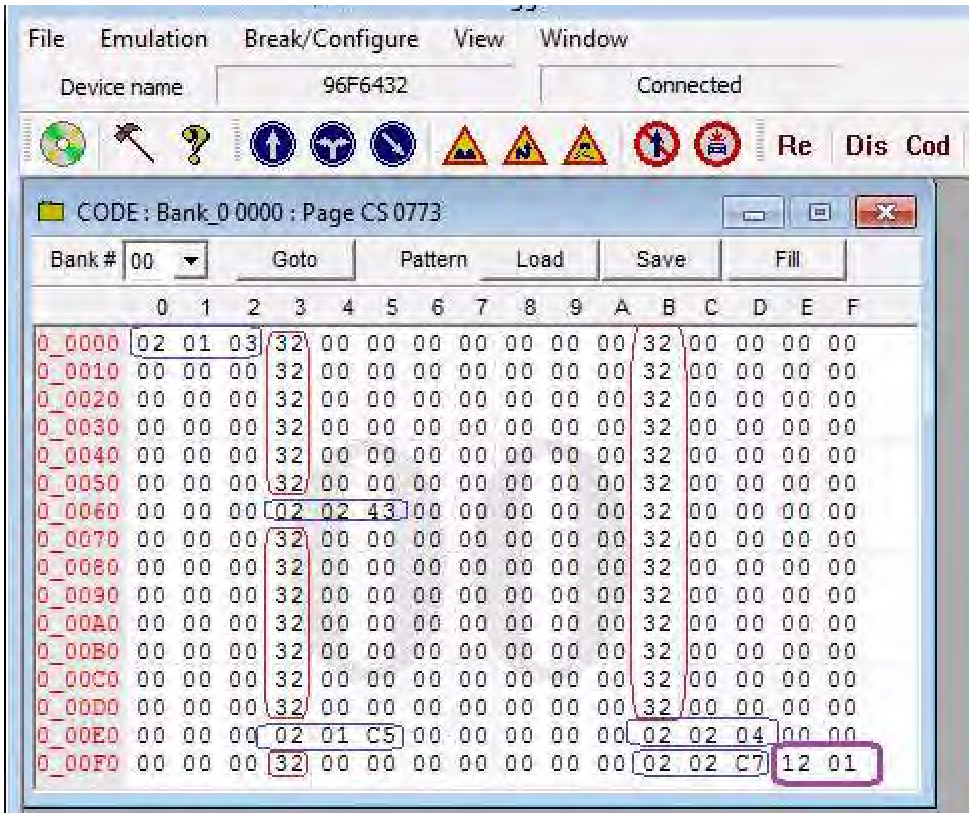
Suppose you have an application for your Z51F0811 chip in which you intend to use only four interrupts: hardware reset, timer 0, and external interrupts 4 and 5. The vectors for these interrupts can be found at addresses `0x00`, `0x63`, `0xE3`, and `0xEB`, respectively. If they are not prevented from doing so, both the SDCC and Keil linkers will begin locating executable code just after the final vector (i.e., the vector with the highest address); in this case, at an approximate address of `0xEE`. For a number of reasons, this practice is not a good one, and it is best to make the Linker avoid the entire vector table for executable code.

The arguments for enforcing this rule include:

- You may experience unexpected interrupts occurring due to noise on the input pins, or other causes
- Your code could mistakenly enable an incorrect interrupt pin
- You might later add another interrupt source which would then conflict with code storage unless you remember to change the Linker settings at the same time

All of these problems might damage your system before you can find and correct the error, if you have executable code in the vector table. It is much safer if the unused vector locations are filled with a RETI or NOP instruction. Additionally, the total amount of space allotted to a complete vector table is usually 256 bytes or less, and typical Z8051 parts have 8KB of program memory. Playing it safe in this regard therefore has only a minor impact on the available code size.

Figure 1 illustrates this issue in a graphical manner, in which the Z8051 OCD Debug Tool is used to capture a snapshot of code memory in the area of the vector table. The interrupts in use are the four that are discussed above. The Linker is essentially forced to avoid putting code into the vector table, by defining a dummy ISR for the last interrupt in the table, external interrupt 7 at address `0xFB`.



Device name 96F6432 Connected

CODE : Bank_0 0000 : Page CS 0773

Bank # 00

```
        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0_0000 02 01 03 32 00 00 00 00 00 00 00 32 00 00 00 00
0_0010 00 00 00 32 00 00 00 00 00 00 00 32 00 00 00 00
0_0020 00 00 00 32 00 00 00 00 00 00 00 32 00 00 00 00
0_0030 00 00 00 32 00 00 00 00 00 00 00 32 00 00 00 00
0_0040 00 00 00 32 00 00 00 00 00 00 00 32 00 00 00 00
0_0050 00 00 00 32 00 00 00 00 00 00 00 32 00 00 00 00
0_0060 00 00 00 02 02 43 00 00 00 00 00 32 00 00 00 00
0_0070 00 00 00 32 00 00 00 00 00 00 00 32 00 00 00 00
0_0080 00 00 00 32 00 00 00 00 00 00 00 32 00 00 00 00
0_0090 00 00 00 32 00 00 00 00 00 00 00 32 00 00 00 00
0_00A0 00 00 00 32 00 00 00 00 00 00 00 32 00 00 00 00
0_00B0 00 00 00 32 00 00 00 00 00 00 00 32 00 00 00 00
0_00C0 00 00 00 32 00 00 00 00 00 00 00 32 00 00 00 00
0_00D0 00 00 00 32 00 00 00 00 00 00 00 32 00 00 00 00
0_00E0 00 00 00 02 01 C5 00 00 00 00 00 02 02 04 00 00
0_00F0 00 00 00 32 00 00 00 00 00 00 00 02 02 C7 12 01
```

Figure 1. Protecting the Vector Table for a Z51F0811 Application

In Figure 1, the used interrupt vectors `0x00`, `0x63`, `0xE3` and `0xEB` are highlighted in blue, as is the vector at `0xFB` for the dummy ISR. The unused vectors, which the Linker has filled with the RETI instruction (op code `0x32`), are highlighted in red. A purple box highlights the beginning of executable code at address `0xFE`.

In both the SDCC and Keil tool sets, the solution is to make the Linker avoid the full extent of the vector table for executable code.

Note: Filling a vector with a RETI instruction is typical of SDCC tools. However, NOP instructions are typically used with the Keil tools.

Application-Specific Hardware Memory Mapping

Some Z8051 MCUs map special hardware devices into a portion of the memory map. In these cases, the Linker must be told not to use that part of the memory map for normal data storage. For example, the Z51F3220 MCUs are oriented toward LCD display applications, and LCD segments are mapped to external data addresses `0x00` to `0x1A` in these devices. The Linker must be instructed to accommodate this by beginning the external data space at address `0x1B`.

Please refer to the product specification for your particular Z8051 MCU to determine if these kinds of special memory map requirements apply to your project.

Interrupt Service Routines

The Keil and SDCC compilers have completely conflicting requirements in the area of function prototypes for ISRs. The SDCC Compiler not only requires you to create a prototype for every ISR, but the prototype must either be in the module that contains the `main()` function, or in a file that is included in that module. This mechanism is one in which the SDCC Compiler creates entries in the vector table for each ISR.

The Keil Compiler, on the other hand, forbids you from declaring a prototype for an ISR anywhere at all! The interrupt keyword is explicitly not allowed in function prototypes. Instead, you simply write the function body in some module, with no prototype anywhere, and the Keil tools gather up all such ISR definitions and make the vector table entries.

These conflicting requirements present a bit of a challenge for writing code that is portable to both compilers. The easiest solution is to place all of the ISR prototypes in a header file which is conditionally included in the module that contains `main()`:

```
#ifdef SDCC
#include "isrs.h"
#endif
```

Alternatively you could use that same conditional compilation around each ISR prototype:

```
#ifdef SDCC
void TIMER0_isr (void) interrupt Z_T0_VECT;
#endif
```

Using The SDCC Tools

SDCC is a Free Open Source software project, distributed under GNU General Public License (GPL). Specifically, SDCC is a retargetable, optimizing ANSI-C compiler that targets the 8051 architecture, among others. In addition to the C Compiler, related development tools such as an assembler and linker are also provided as part of the overall

SDCC package. For more information about SDCC, refer to the SDCC - Small Device C Compiler website or search the Internet for *SDCC*.

As part of the Z8051 software distribution that you received with your Z8051 development kit, Zilog has included a copy of the latest distribution of SDCC tools. These tools can be found in the following path:

```
<Z8051 Install>\sdcc_x.y.z
```

In this path, `x.y.z` is the full SDCC version number (for example, `3.1.0`). These SDCC tools include everything you require to build applications for the Z8051, including a C compiler, assembler and linker.

In this document, we will discuss the use of SDCC outside of an integrated development environment (IDE). The SDCC tools do not themselves include an IDE or other GUI for driving the tools. Instead, the basic approach for developing applications using SDCC is through a command-line interface or equivalent batch execution files which embed the commands inside the batch file. Some instances of tool chains that have integrated the SDCC tools have been developed, or are under development, using the Eclipse or other IDEs. You may wish to search for these tool chains on the Internet, because the availability and status of these tool chains is often fluid.

In this section, we will discuss the following topics:

- How to build an application with the SDCC tools
- How to solve memory mapping issues (vector table allocation and hardware memory mapping) with the SDCC tools
- Limitations and oddities of the SDCC Compiler
- Debugging an SDCC-compiled application with the Z8051 OCD Debug Tool
- Where to obtain further information about the SDCC tools

Building Your Application With SDCC Tools

The process for building your application with the SDCC tools is essentially a very simple one, as follows:

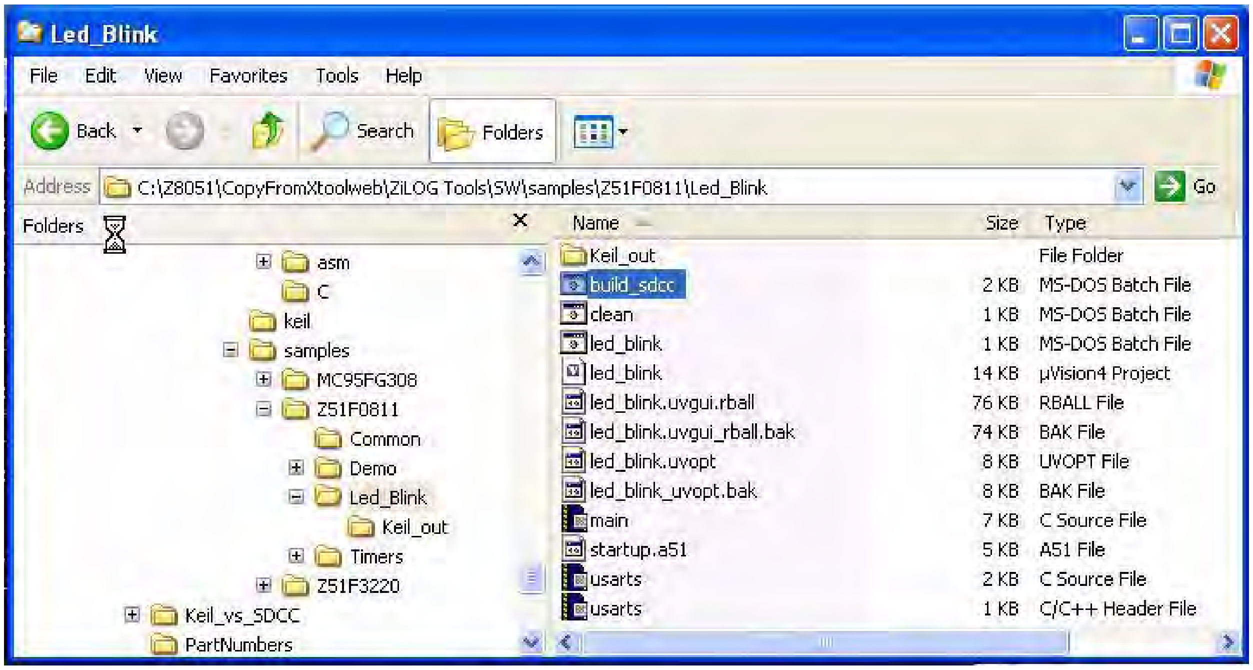
1. Open a command window.
2. Enter your commands with the correct options to compile each source code module in your application; enter these commands one by one.
3. Enter a command with the correct options to link all of the compiled modules into an executable file.

However, complications can arise upon specifying the *correct options* noted in steps 2 and 3. Additionally, it is unnecessarily laborious to literally type in commands one by one. It is much more convenient to place all of the commands together in a batch file, then execute the batch file. There are other conveniences we can place in the batch file to make creating the batch file less tedious.

The remainder of this section discusses the details and the complications of such batch files, some of which are included in the Z8051 software installation. However, it is useful to remember that the purpose of these batch files is to first compile the separate modules, then link them.

Running A Batch File

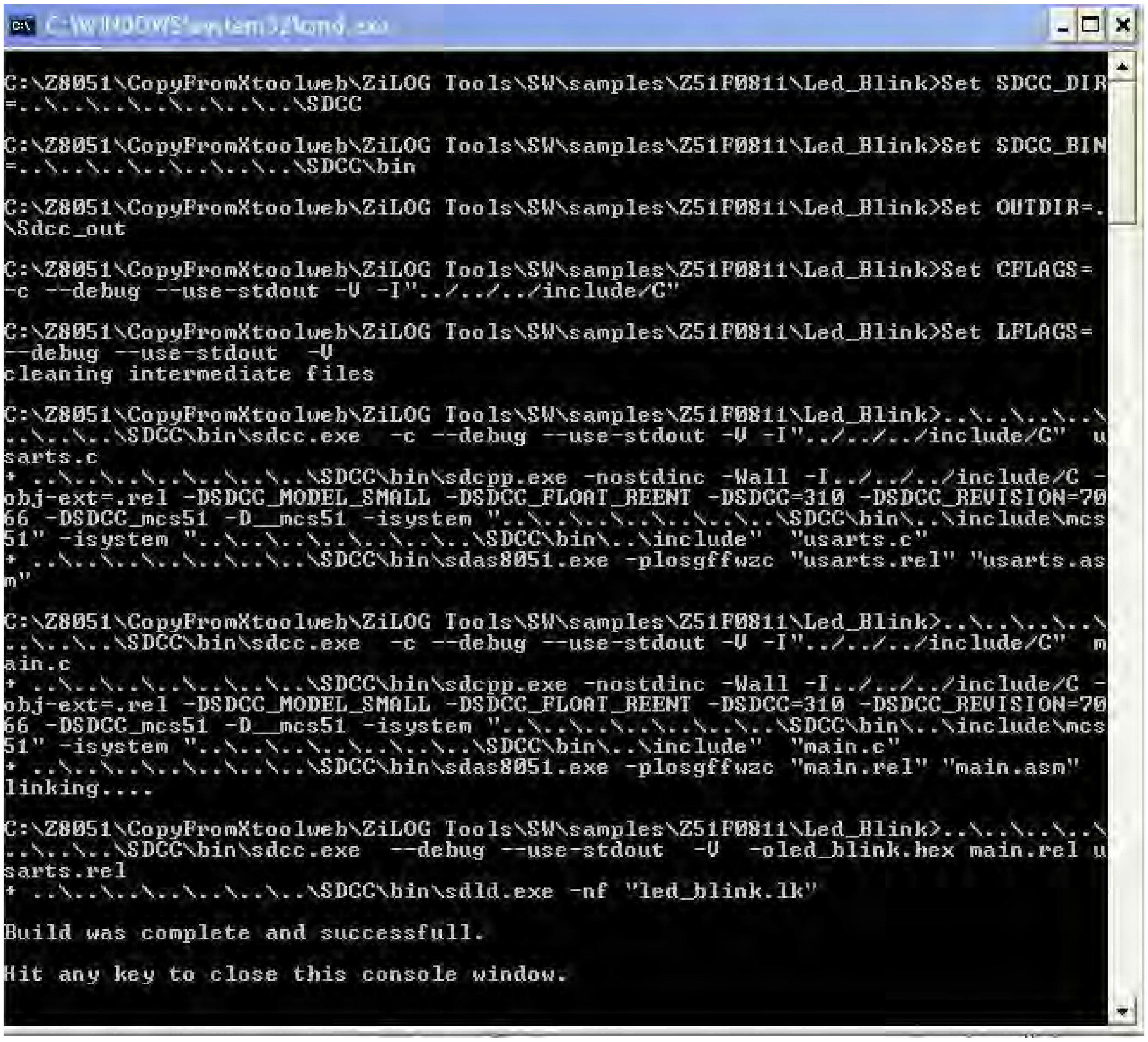
After a batch file has been correctly written, building the application is extremely easy: just navigate to the batch file (an example batch file is highlighted in Figure 2) and double-click it. Figure 3 shows the result of executing this batch file.



C:\Z8051\CopyFromXtoolweb\ZiLOG Tools\SW\samples\Z51F0811\Led_Blink

Name	Size	Type
Keil_out		File Folder
build_sdcc	2 KB	MS-DOS Batch File
clean	1 KB	MS-DOS Batch File
led_blink	1 KB	MS-DOS Batch File
led_blink	14 KB	µVision4 Project
led_blink.uvgui.rball	76 KB	RBALL File
led_blink.uvgui_rball.bak	74 KB	BAK File
led_blink.uvopt	8 KB	UVOPT File
led_blink_uvopt.bak	8 KB	BAK File
main	7 KB	C Source File
startup.a51	5 KB	A51 File
usarts	2 KB	C Source File
usarts	1 KB	C/C++ Header File

Figure 2. Executing a Batch File



```
C:\Z8051\CopyFromXtoolweb\ZiLOG Tools\SW\samples\Z51F0811\Led_Blink>Set SDCC_DIR
=..\..\..\..\..\..\..\..\SDCC

C:\Z8051\CopyFromXtoolweb\ZiLOG Tools\SW\samples\Z51F0811\Led_Blink>Set SDCC_BIN
=..\..\..\..\..\..\..\..\SDCC\bin

C:\Z8051\CopyFromXtoolweb\ZiLOG Tools\SW\samples\Z51F0811\Led_Blink>Set OUTDIR=.
\Sdcc_out

C:\Z8051\CopyFromXtoolweb\ZiLOG Tools\SW\samples\Z51F0811\Led_Blink>Set CFLAGS=
-c --debug --use-stdout -V -I"../../../include/C"

C:\Z8051\CopyFromXtoolweb\ZiLOG Tools\SW\samples\Z51F0811\Led_Blink>Set LFLAGS=
--debug --use-stdout  -V
cleaning intermediate files

C:\Z8051\CopyFromXtoolweb\ZiLOG Tools\SW\samples\Z51F0811\Led_Blink>..\..\..\..\
..\..\..\SDCC\bin\sdcc.exe  -c --debug --use-stdout -V -I"../../../include/C"  u
sarts.c
+ ..\..\..\..\..\..\..\SDCC\bin\sdcpp.exe -nostdinc -Wall -I../../../include/C -
obj-ext=.rel -DSDCC_MODEL_SMALL -DSDCC_FLOAT_REENT -DSDCC=310 -DSDCC_REVISION=70
66 -DSDCC_mcs51 -D__mcs51 -isystem "..\..\..\..\..\..\..\SDCC\bin\..\include\mcs
51" -isystem "..\..\..\..\..\..\..\..\SDCC\bin\..\include"  "usarts.c"
+ ..\..\..\..\..\..\..\SDCC\bin\sdas8051.exe -plosgffwzc "usarts.rel" "usarts.as
m"

C:\Z8051\CopyFromXtoolweb\ZiLOG Tools\SW\samples\Z51F0811\Led_Blink>..\..\..\..\
..\..\..\SDCC\bin\sdcc.exe  -c --debug --use-stdout -V -I"../../../include/C"  m
ain.c
+ ..\..\..\..\..\..\..\SDCC\bin\sdcpp.exe -nostdinc -Wall -I../../../include/C -
obj-ext=.rel -DSDCC_MODEL_SMALL -DSDCC_FLOAT_REENT -DSDCC=310 -DSDCC_REVISION=70
66 -DSDCC_mcs51 -D__mcs51 -isystem "..\..\..\..\..\..\..\SDCC\bin\..\include\mcs
51" -isystem "..\..\..\..\..\..\..\..\SDCC\bin\..\include"  "main.c"
+ ..\..\..\..\..\..\..\SDCC\bin\sdas8051.exe -plosgffwzc "main.rel" "main.asm"
linking....

C:\Z8051\CopyFromXtoolweb\ZiLOG Tools\SW\samples\Z51F0811\Led_Blink>..\..\..\..\
..\..\..\SDCC\bin\sdcc.exe  --debug --use-stdout  -V  -oled_blink.hex main.rel u
sarts.rel
+ ..\..\..\..\..\..\..\SDCC\bin\sdld.exe -nf "led_blink.lk"

Build was complete and successfull.

Hit any key to close this console window.
```

Figure 3. Results of Batch File Execution

The remainder of this section examines the contents of one of these batch files, which typically include the following elements:

- Definitions of some text strings that make writing the commands easier and less repetitious, including paths and build options
- Commands to perform preliminary steps such as cleaning away old files
- Commands to compile modules
- A command to link the compiled modules into an executable

- A command to bring execution to a tidy completion

The specific file to work with is listed in its entirety in Appendix A. SDCC Build Batch File Listing on page 52. This file is similar to those provided in the sample application folders contained in the following path, which includes sample files for building a number of demo applications:

```
<Z8051 Install>\samples
```

Defining Paths

The first section of the file uses the Set command to define some text strings, so that the string name can be used later in the file, and thereby avoid the repeated typing of long text strings. The relevant code fragment from the appendix appears below for convenience:

```
Set SDCC_DIR=..\..\..\SDCC_3.1.0
Set SDCC_BIN=%SDCC_DIR%\bin
Set LIB_DIR=..\..\lib\sdcc
Set OUTDIR=.\Sdcc_out
```

In the above code segment, `SDCC_DIR` represents the path to the location in which SDCC (in this case, SDCC_3.1.0) was installed on your PC. All paths in this batch file are supplied in relative terms, starting from the folder in which the batch file is located. The "`..\`" notation in this path refers to the parent folder of this folder; therefore, the "`..\..\..`" notation means *the folder three levels above this folder*. In other words, this batch file is designed to be run from a folder three levels below the `<Z8051 Install>` directory, and the definition for `SDCC_DIR` points to `<Z8051 Install>\SDCC_3.1.0`. Using similar definitions, you can create a batch file anywhere on your computer and define the path to the SDCC folder with a similar syntax.

The next line defines `SDCC_BIN` as the bin folder beneath `SDCC_DIR`. `LIB_DIR` is defined next as a folder that will be used later; a particular library in this folder will be required in the link step.

Note: The `%` symbols surrounding `SDCC_DIR` in the above code segment mean that `SDCC_DIR` is intended to be interpreted as a previously-defined text string and not as a literal string.

In the line after that, a string called `OUTDIR` is defined. This string could be used to define an output folder to which the build output is to be directed. That folder is located just below the current folder, because the period character (.) refers to the current directory. Due to a quirk of the OCD debugger (a discussion that follows), `OUTDIR` is not actually used for that purpose in this batch file.

Note: Further information regarding command line or batch file syntax can be found in the Microsoft Windows Help and Support utility.

Defining Build Options

The remaining two Set commands define two strings, CFLAGS (compiler flags) and LFLAGS (linker flags), respectively. As a software project, SDCC was developed in a Unix- or Linux-style environment and, although it can be run either in these two types of operating system or in a Windows-style OS, it uses a Unix-like syntax to express the compiler and linker options as command-line *flags*. In this syntax, each option for the Compiler or Linker is given as a text string that begins with a dash or minus sign symbol (–) followed by the text that defines the option. The text which defines that option ends when white space, such as the space character, is encountered. Spaces are not allowed inside a command line option.

The definition of CFLAGS is:

```
Set CFLAGS= -c --debug --use-stdout --model-large -V -I"../../../
include/C" -I"../common"
```

Note: The Set CFLAGS command is a run-on command which continues across the two lines of code above.)

This set of options is useful as a fairly generic group of compiler settings to build a typical application.

The Set CFLAGS command causes the Compiler to include debug information in its output so that the application can be debugged with a symbolic debugger. Error messages must be directed to the standard output, which will cause them to be displayed in the command window in which you eventually will run the batch file. The SDCC *large* memory model will be used to place all variables into the `XData` space, unless otherwise specified. The
`–V` (*verbose)* option causes the Compiler to display commands as it executes them, which makes it easier for you to follow the Compiler's progress as it tries to build your application. The final two commands, with the `–I` option followed by a path inside quotation marks, supply paths that must be searched for include files.

LFLAGS, in our simple example, uses basically the same options:

```
Set LFLAGS= --debug --use-stdout --model-large -V
```

Executing Preliminary Steps

Following the definition of these text strings using the Set command are the commands to execute your actual build, beginning with:

```
@echo cleaning intermediate files
@call clean.bat

@echo compiling....
```

Several of these text strings use a syntax of `@echo <some text>` to instruct the batch process to display, or echo, the data you have specified in the `<some text>` parameter to your command window as the batch file executes. This display mechanism is, again, a way for you to keep track of the Compiler's progress as it builds your application. Another type of command which appears often in this batch file uses the `@rem <some text>` syntax. In this syntax, the `@rem` element can be thought of as a *remark*; such lines are simply comments in the batch file and are not executed.

The first actual command that the batch file executes is to call another batch file, `clean.bat`, which must be included in the same folder as the build batch file. The `clean.bat` file contains only one line which, as it is displayed below, wraps onto a second line.

```
@del *.sym *.lk *.mem *.adb *.lst *.asm *.rel *.rst *.cdb *.hex
*.map *.omf
```

The above code instructs the batch file to delete all files in the current directory (via the `@del` command) that include any of the file name extensions that it lists. This task is necessary because the build process, as defined in this batch file, is going to place all of its output files – both the final executable file as well as all types of intermediate files – in the current directory. We could avoid getting the current folder cluttered with all those files by using `OUTDIR` to send them to a separate folder. As discussed above, however, the properties of the OCD debugger raise issues with that approach. Therefore, we have chosen to use this technique of placing the files into the same folder as the source code and purging the old files with each fresh build.

Compiling Source Code Modules

After `clean.bat` is run, the the Compiler can begin execution, as shown in the following code segment:

```
@echo compiling....
@rem uarts.c
%SDCC_BIN%\sdcc.exe %CFLAGS% -o"."/ ../common/uarts.c
@if errorlevel 1 goto Compiling_Error

@rem main.c
%SDCC_BIN%\sdcc.exe %CFLAGS% main.c
@if errorlevel 1 goto Compiling_Error
```

The previously-defined `SDCC_BIN` string is used to give the path to the `sdcc` executable, which invokes the Compiler. Next, the CFLAGS string is used to specify compiler options, as described earlier; the name of the C source code module is then given. This command is then simply repeated for each of the modules in the application: in this case, the program consists of the source code files `uarts.c` and `main.c`.

Note: `uarts.c` is not located in the same folder as the batch file; therefore, its path must be supplied in the command. The `–o"."/` option for building the `uarts.c` file tells the Compiler to place the output of this compilation (via the `–o` command) in the current directory, remembering that `"."` refers to the current directory using Unix-style commands.

After each source code file is compiled, the batch file checks to determine if errors were reported. If errors exist, execution is diverted to the label specified by the **Goto** command:

```
:Compiling_Error
@Echo.
@Echo !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
@Echo !!!   Compiling Errors occurred. See above  !!
@Echo !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
@goto end

:Linking_Error
@Echo.
@Echo !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
@Echo !!!   Linking Errors occurred. See above    !!
@Echo !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

In this case, the command window, after reporting the specific error messages from the Compiler, would display another message to alert you that the build was terminated by compiler errors; this command window then terminates. A similar approach is used for reporting linker errors. If the build is error-free, the following success message will be displayed:

```
@echo.
@echo Build was complete and successful.
@goto end
```

The compilation process creates several output files for each source file as it is compiled.

Note: The Compiler actually creates assembly code, then automatically invokes the 8051 assembler to assemble the generated assembly code into object code.

For example, in this build – assuming there are no errors – we should create the `uarts.adb`, `uarts.asm`, `uarts.lst` and `uarts.rel` files, as indicated in Table 1. Similarly, when `main.c` is compiled, four new files with these filename extensions will be created, such as `main.adb`, `main.lst`, etc.

Table 1. Intermediate Files Upon C Module Compilation

File	Description
uarts.adb	An intermediate file containing debug information required to create the .cdb file with the OCD tool.
uarts.asm	An assembly source file created by the Compiler.
uarts.lst	An assembler listing file.
uarts.rel	The object file created by the Assembler that becomes input for the Linker.

Linking

After all of the C files are compiled, the final step in the build process is to link them together into a completed executable application by executing the next command in our batch file, as the following code segment shows.

```
@echo linking....
%SDCC_BIN%\sdcc.exe %LFLAGS% -oled_blink.hex main.rel uarts.rel
%LIB_DIR%\crtxinit.rel
@if errorlevel 1 goto Linking_Error
```

Note that SDCC is slightly unusual, in that the same executable program, `sdcc.exe`, performs both the compiling and linking operations. Which operation is to be performed in a given command depends on the type of input file it is tasked to work on. The input file or files are supplied as the final arguments in the command line. In the command above, the input files are primarily the `.rel` files previously created in the compile step; therefore the SDCC knows that it is being tasked to run the Linker on these files. The output of this step (again, the `-o` command is used to specify output) is an executable file called `led_blink.hex`.

This link command also shows how you can link to a separate, previously-built library to create your application simply by including its name among the files to be linked when you invoke the linker phase of the `sdcc.exe` executable. In this case, we have included an initialization library, `crtxinit.rel`, from the folder defined earlier as `LIB_DIR`. You may or may not need to include this library in your own application, depending on which Z8051 MCU you are working with. See the XRAM Initialization section on page 18 for more information about that library. Additionally, note in this case that we have specified the path to this particular library. If you prefer, you can also use the `-L <library-path>` option to indicate where to find a separate library.

The linker flags (LFLAGS) string that is defined in the first part of the batch file is used to define the options for the linker step. In addition to the `led_blink` hex output file, the

Linker will also create the updated assembler listings (i.e., the `.rst` files) discussed previously; see Table 2.

Table 2. Output Files Upon Object File Linkage

File	Description
led_blink.cdb	A file that combines the debug information from all of the source modules.
led_blink.map	A file that contains the application's memory map.
led_blink.mem	A file that summarizes memory usage.
led_blink.lk	A file that contains summary information about the linking process.
led_blink.omf	A file that contains debug information in AOMF or AOMF51 format.
uarts.rst	An updated version of the Assembler listing file, created by the Linker.

Completion

The final line of the batch file simply pauses execution before the batch file exits. This element can be useful toward providing you a chance to read the output from the build process. When you are prompted to press any key to continue, the execution of the batch file finishes by closing the console window:

```
:end
@Echo.
@Echo Hit any key to close this console window. > CON:
@pause > NUL:
```

Now that we have finished discussing the contents of a basic batch file to build an SDCC application, we are ready to take up some more intricate topics.

Vector Table Allocation

The Controlling The Memory Map section on page 4 explains why the Linker should be instructed to avoid placing executable code in the address space used by the interrupt vector table. In the SDCC tools, this instruction can be set by providing at least a dummy ISR for the final interrupt in the vector table; i.e., the interrupt with the lowest priority and highest vector address that is defined for your particular MCU. For the Z51F0811, for example, this final interrupt is external interrupt 7, which has interrupt number 31 and vector address `0xFB`; see the example in Figure 1 on page 6.

The prototype of this ISR, similar to all ISRs used in the SDCC tools, must either be placed into the same module that contains the `main()` function, or be placed into a file that is included in this module. As a result, the Linker places an entry for this ISR into the vector table and only begins placing executable code after that vector. Additionally, an actual definition for the ISR must be provided somewhere in your program, although it can be empty or can consist only of a RETI or NOP instruction.

Application-Specific Hardware Memory Mapping

As discussed in the Controlling The Memory Map section, in some Z8051 MCUs the Linker must be told to avoid using a portion of memory that is dedicated to memory-mapped hardware. In the SDCC tools, you do this by adding commands to the Linker commands, the LFLAGS in our example batch file. In the example of the Z51F3220 MCUs in which LCD segments are mapped to external data addresses `0x00` to `0x1A`, the following LFLAGS definition accommodates this by beginning the XRAM data space at address `0x1B`:

```
@rem XData 0-1A is for LCD. Therefore, XRAM must start at 0x1B
Set LFLAGS= --debug --use-stdout --model-large --xram-loc 0x1b --
float-reent -V
```

Limitations And Oddities Of The SDCC Compiler

A few known, minor limitations of the SDCC Compiler are worth noting. These should not affect typical embedded applications, for the most part. You will receive compiler error messages if you try to use the following somewhat obscure features of the C language:

- SDCC does not support the standard C data type long double
- SDCC does not support the standard C syntax for wide characters, `L'c'` or `L"string"`
- SDCC has difficulty with redefined struct or union names in an inner scope. For example, the following code will fail:

```
struct myStruct
{
int i;
short s;
};

void main()
{
// The following is a new definition
// and will fail
struct myStruct {short s; int i;};
...
}
```

By default, the SDCC Compiler is also generous with its warnings. Some of its warning messages can be more entertaining than informative, reflecting SDCC’s history as an open-source software project. For example, one warning that is often safe to ignore[1] is displayed as:

1. It typically only matters when you should have declared a variable to be volatile, but did not.

```
warning 110: conditional flow changed by optimizer: so said EVELYN
the modified DOG
```

The SDCC documentation refers to this as *the (in) famous message*. You can disable this particular message by including the following statement in your code:

```
#pragma disable_warning 110
```

A wider range of warning messages can be disabled by using the `--less-pedantic` command among your compiler options. This command begins with two dashes. However, this command is a bit broad in its scope, and disables some warnings that you might wish to continue seeing. This command is fully described in the SDCC documentation, which you will find in the `sdccman.pdf` file located in the following path:

```
<Z8051 Install>\sdcc_x.y.z\doc\
```

XRAM Initialization

There is a special setup issue in the SDCC tools which affects those Z8051 MCUs that feature on-chip XRAM memory. You can determine whether this issue applies to your MCU by determining whether there is a section titled *XRAM (or XSRAM) Memory* in the chapter about memory in the Product Specification. For parts that do not offer XRAM or XSRAM memory, there is no issue.

However, if your MCU does feature on-chip XRAM or XSRAM memory, there is a potential problem with the default SDCC setup. Specifically, the default method used for initializing variables in that memory space may not work properly. The default initialization used by the SDCC tools uses 8-bit addressing, which will not work if you have over 256 bytes of variables in this space that must be initialized. Instead, you must set up the SDCC tools to use the alternative dual-pointer method for initializing your XRAM data.

Zilog recommends that the dual-pointer method always be used for initializing XDATA, even if you have less than 256 bytes of such data. This approach is safer because, as your program grows over time and you add more initialized data, your application would suddenly stop working with no warning when the 256-byte limit is exceeded.

Zilog has provided a modified version of the initialization library, `ctrxinit.rel`, which performs this dual-pointer initialization for you. To use this modified library, you simply need to include the `ctrxinit.rel` file in the link step of your batch file.

The `ctrxinit.rel` file is located in the following path:

```
<Z8051 Install>\lib\sdcc\
```

The sample batch file discussed above shows an example of how to include this file.

Debugging With The Z8051 OCD Debug Tool

One way in which the SDCC tools fall short of being a full, integrated development platform is that they cover only the build aspect of program development, not the debug side (except that they do create files with debug information that can be used by a debug tool). To download, run, and debug your application, you must use a separate tool. The Z8051 OCD (On-Chip Debugger interface) tool is available for these purposes. Please refer to the Z8051 OCD User Manual (UM0240) for information about using this tool.

Directory Restrictions

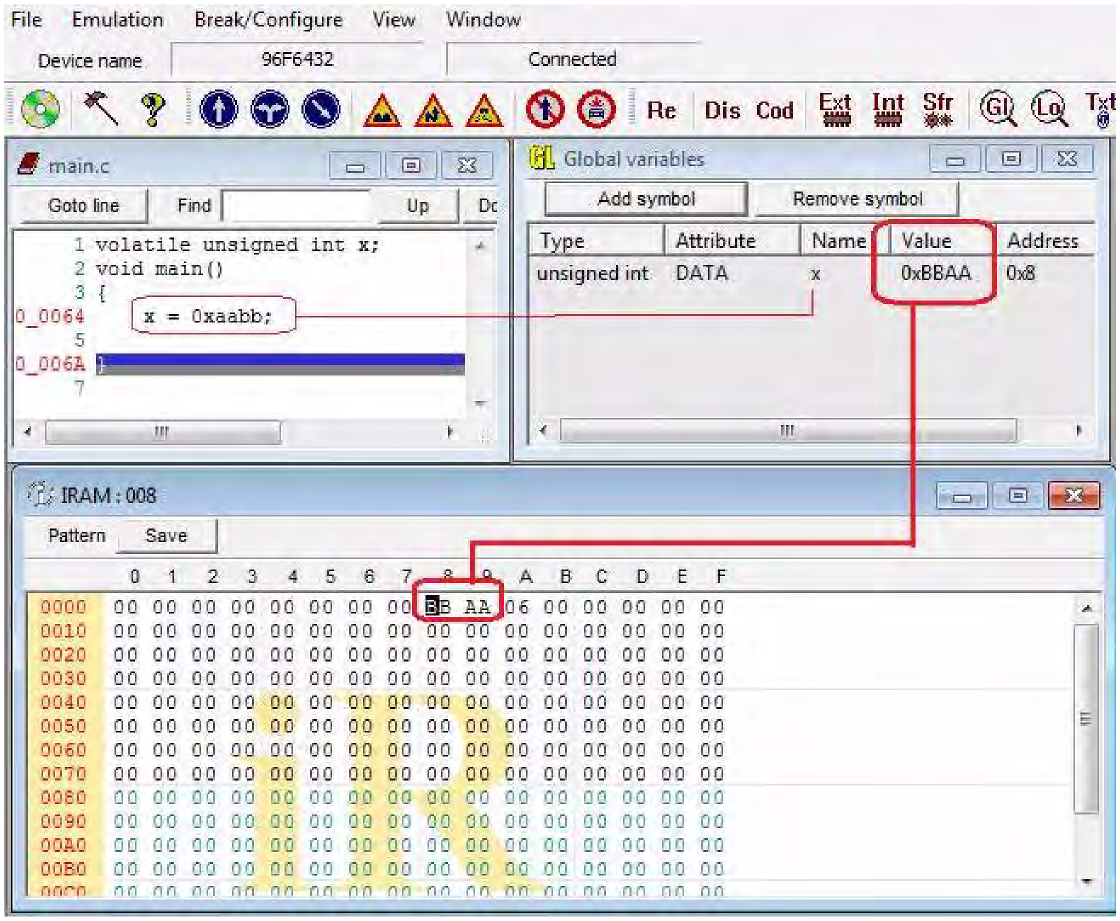
We have referred above to a feature of the OCD Debug Tool which has an impact on how we choose to place particular files involved in our build into a given folder. Specifically, the debugger must find both the source code files and also some files that are created in the build, such as the debug information and the object and hex files, in the same directory. It is for these reasons that, in the batch file discussed above, we chose not to use the `OUTDIR` string to define a separate output folder for the products of the build process, and to use the `clean.bat` batch file in every fresh build to delete old and extraneous files from the folder in which the batch file and source code are located.

Endianness

Another feature of this debug tool that you must be aware of when working with the SDCC tools is the issue of endianness; search the Internet for this term if you are not familiar with the concept. Briefly, the question concerns the byte order in which values are stored in memory. As an example, in the Z8051 both the SDCC and Keil compilers define an *int* to be a 2-byte value. Suppose you have an *int* variable, *x*, for which the value is `0xaabb`; this value is stored in memory at addresses `0x62–0x63`. In a *big-endian* system, the value `0xaa` would be stored at address `0x62` and the value `0xbb` would be stored at address `0x63`. In a *little-endian* system, these would be reversed: `0xbb` would be stored at address `0x62` and `0xaa` would be stored at address `0x63`. Either system is valid as long as it is followed consistently.

The specific issue in the OCD is that SDCC uses little-endian storage, but the OCD follows the convention of the Keil tools which use big-endian storage. This means that the bytes will be reversed whenever you use the OCD global or local watch window to display a value longer than one byte. In the case described above, the OCD will show the value of x as `0xbbaa`, when its true value is `0xaabb`. You must keep this issue in mind and mentally reverse the bytes as you work on debugging your code.

Figure 4, from the Z8051 OCD Debug Tool, shows an example of the OCD watch window in which the bytes in variable *x* are reversed due to the SDCC's use of the little-endian convention.



```
        volatile unsigned int x;
        void main()
        {
0_0064    x = 0xaabb;

0_006A  }
```

Type	Attribute	Name	Value	Address
unsigned int	DATA	x	0xBBAA	0x8

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000  00 00 00 00 00 00 00 00 BB AA 06 00 00 00 00 00
0010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 4. Endianness Reversal In SDCC

Note: In Figure 4, the least-significant byte of x is stored at the lower address.

More information about the SDCC tools is provided in the SDCC material in your Z8051 software installation. You should refer to that documentation for answers to the more in-depth questions that may arise as you explore the tools. You will find this information in the `sdccman.pdf` file, which is located in the following path:

```
<Z8051 Install>\sdcc_x.y.z\doc\
```

Another document to consider is the `asxhtm` file, which describes the assembler used in the SDCC tool chain. This file is located in the following path:

```
<Z8051 Install>\sdcc_x.y.z\doc\sdas\
```

Using The Keil Tools

Keil Software, now a division of ARM Ltd., is a very well-known supplier of high-quality, widely-accepted development tools in the embedded space. The 8051 processor architecture is one of the many architectures they support. Compared to SDCC, the Keil tools for 8051 chips often provide superior performance in metrics such as generated code size, and also offer IDE support which makes use of the tools considerably easier.

Caution: You must not use the Keil tools and software in a manner inconsistent with any license agreement that you may have with Keil. Zilog is not responsible for any such use by you in violation of any license agreement you may have with Keil. ZILOG DISCLAIMS ALL WARRANTIES, INCLUDING WARRANTIES FOR MERCHANTABILITY AND/OR FOR A PARTICULAR PURPOSE.

Keil offers a restricted-usage subset of their 8051 tool chain, which is freely available for download from their website at www.keil.com. This subset is nearly identical to the tools that provide full support for 8051 projects, except for a rather strict limitation on the size of the application that can be built. Therefore, the free download version is appropriate for evaluating and learning the Keil tools, but will probably not be a viable path to developing real applications. This document, however, concentrates primarily on this free version of the Keil tools, because users are very likely to try evaluating the tools before they buy, and it is in this phase of tool evaluation in which guidance can be helpful.

This section discusses the following topics:

- How to build an application with the Keil tools
- How to solve memory mapping issues (vector table allocation and hardware memory mapping) with the Keil tools
- Potential pitfalls when building with the Keil tools
- Debugging a Keil-compiled application with the Z8051 OCD Debug Tool
- Where to find further information about the Keil tools

The first step in getting started with the Keil 8051 evaluation tools is to go to their website and download them. The tools you must download are the C51 Development Tools; as discussed above, *C51* is Keil’s name for their 8051 C Compiler (and, in some contexts, their associated tools for the 8051). You can install these tools wherever you prefer on your PC. If you choose a typical installation, your installation will store a large number of files and folders on your PC. The most important of these files are listed in Table 3.

Table 3. Important C51 Development Tools

File	Description
Uv4.exe	The uVision4 Integrated Development Environment in which you will do your builds.
C51	The 8051 C Compiler.
A51	The 8051 Assembler.
BL51	The 8051 Basic Linker.

Building Your Application With Keil Tools

The Writing Your Code: Notes About Header Files section on page 2 applies equally to the Keil tools as it did to the SDCC tools, and is therefore not repeated in this section.

After you have written some code, the steps to build it are the same as they are for the SDCC (or for any tools): first compile (or assemble, if assembly code) the individual modules, then link them into a complete application. However, the uVision IDE makes these tasks a good deal easier. The steps in building your application are all performed via the GUI. Basically, they are:

- Create a Project and populate it with your source code files
- Set the correct project options
- Click a button to build the project (i.e., compile and link it)

In reality, there is much more complexity involved, most of it hidden in the term *the correct project options*. In this section, we examine the most important points to know about getting successful builds with the free Keil evaluation tools.

This section discusses the following topics:

- How to enable and select your Z8051 MCU in the uVision IDE
- The Project concept in uVision
- Source code in a uVision Project
- Building an existing Project
- Creating a new Project
- Project options and how to avert potential problems

The uVision Project

Now you're ready to start up uVision, either by selecting it from your Start menu (if a shortcut was installed there when you installed the Keil tools) or by navigating to the `<Keil install>\UV4` path and double-clicking the UV4 application. As in most other IDEs, a basic concept in uVision is the Project, which groups together all of the resources required for building one application. It is easy to create a new project from scratch, and we'll get to that procedure soon. However, let's start by examining a project that has already been created for you and included in your Zilog software installation.

Because of the code size restriction of the free Keil tools, you must work with a fairly small application. The `led_blink` application for the Z51F0811 MCU is included in the Zilog installation for just this purpose. This application simply blinks a set of colored LEDs in sequence if you run it on the board that is included in the Z51F0811 MCU Evaluation Kit. If you do not have this kit, you will not be able to actually run the application, but you can still build it using the Keil evaluation tools which will start to give you a feel for using them.

1. In uVision, click **Project** → **Open Project**, then navigate the following path:

   ```
   <Z8051 Install>\samples\Z51F0811\Led_Blink
   ```

2. Locate and select a uVision project file called `led_blink.uvproj`, and click **Open**. In the left-hand pane of the IDE, an item called `led_blink` appears with a plus sign icon (`[+]`) next to it, as shown in Figure 5. This top-level `led_blink` entity is what uVision calls a *target*; i.e., the ultimate entity that will be built. You can think of this *target* as representing the executable you will build.

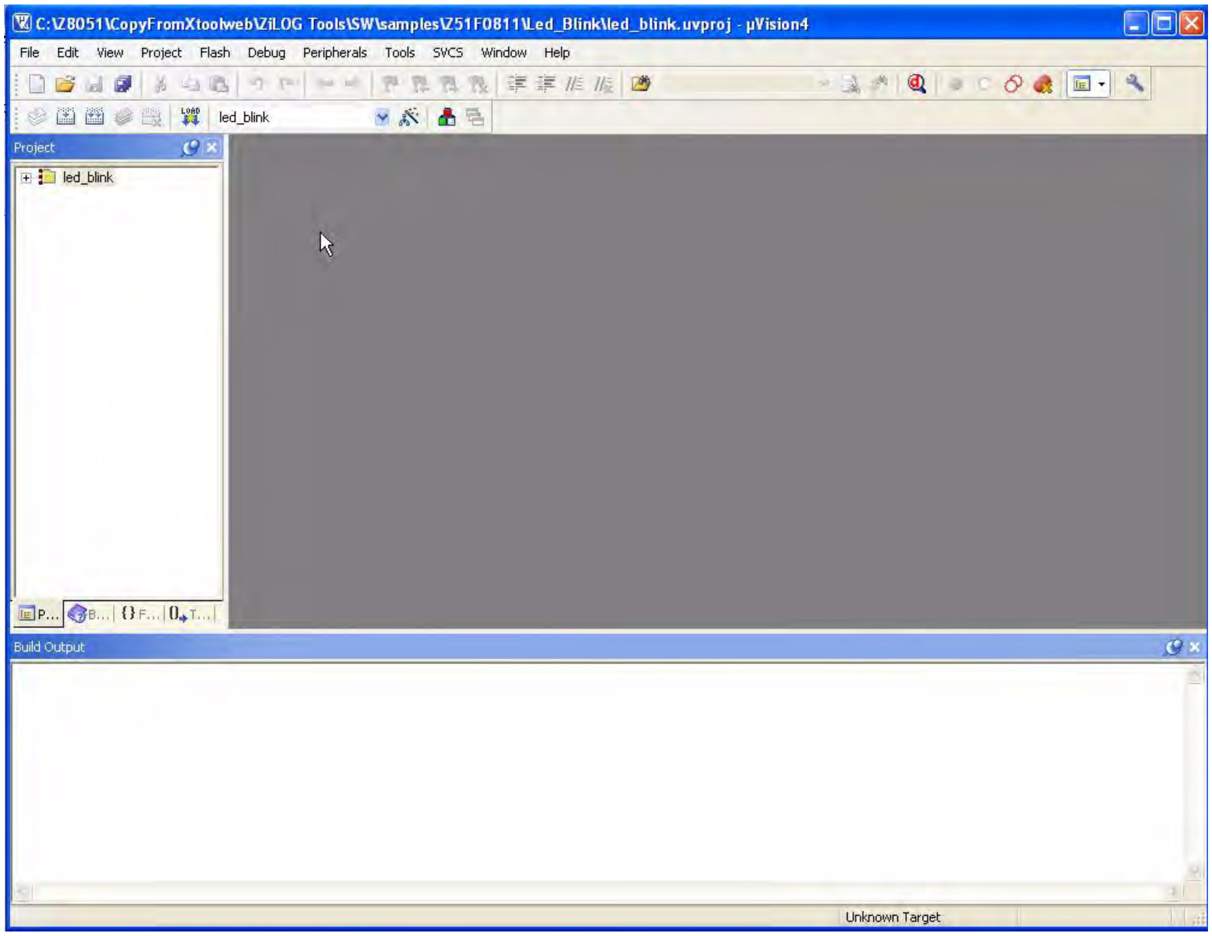

C:\Z8051\CopyFromXtoolweb\ZiLOG Tools\SW\samples\Z51F0811\Led_Blink\led_blink.uvproj - µVision4
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
led_blink
Project
led_blink
Build Output
Unknown Target

Figure 5. Opening The led_blink Project

Selecting Your Z8051 Device

Before building, select your Z8051 MCU as the device that is in use by opening the project options for the `led_blink` target, as follows.

1. Highlight the `led_blink` item in the left-hand pane of the uVision window. Then, from the list of menus at the top of the window, select **Project** → **Options** for the `led_blink` target, as indicated in Figure 6.

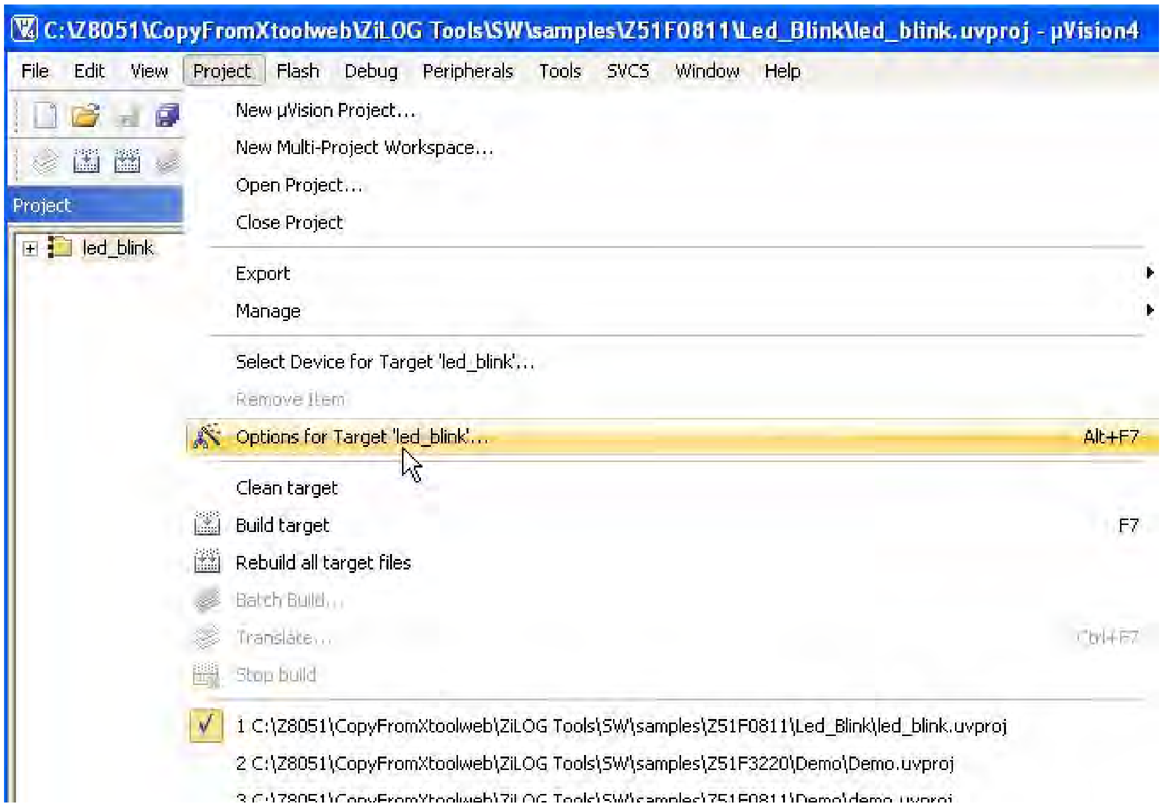
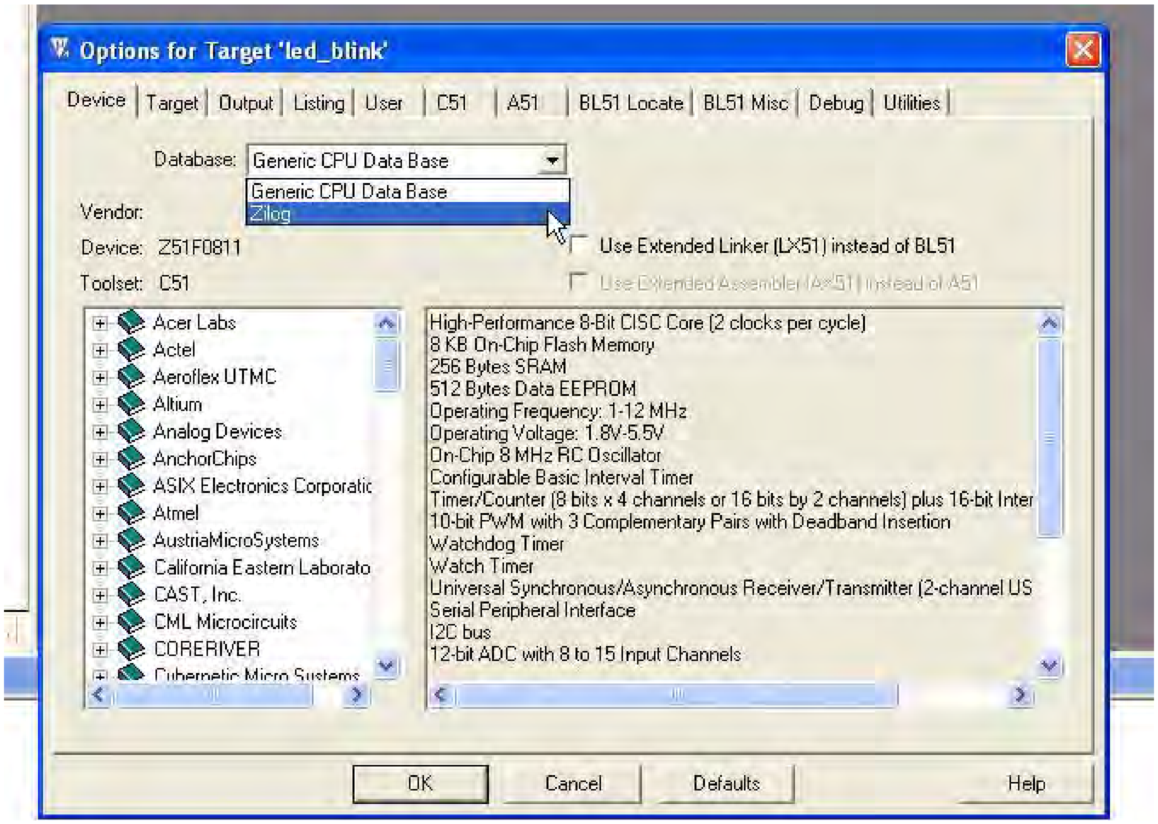

C:\Z8051\CopyFromXtoolweb\ZiLOG Tools\SW\samples\Z51F0811\Led_Blink\led_blink.uvproj - µVision4
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Project
led_blink
New µVision Project…
New Multi-Project Workspace…
Open Project…
Close Project
Export
Manage
Select Device for Target 'led_blink'…
Remove Item
Options for Target 'led_blink'… Alt+F7
Clean target
Build target F7
Rebuild all target files
Batch Build…
Translate… Ctrl+F7
Stop build
1 C:\Z8051\CopyFromXtoolweb\ZiLOG Tools\SW\samples\Z51F0811\Led_Blink\led_blink.uvproj
2 C:\Z8051\CopyFromXtoolweb\ZiLOG Tools\SW\samples\Z51F3220\Demo\Demo.uvproj
3 C:\Z8051\CopyFromXtoolweb\ZiLOG Tools\SW\samples\Z51F0811\Demo\demo.uvproj

Figure 6. Selecting The Project Options

Note: If you had highlighted a lower-level item in the left-hand pane, **Project** → **Options** for the `led_blink` target would not be available from the Project menu, but instead only a more limited selection such as **Project** → **Options** for the `main.c` file.

The Project Options dialog displays many tabs across the top, and a great variety of options are available to control your build. We'll explore a few of these in more detail later.

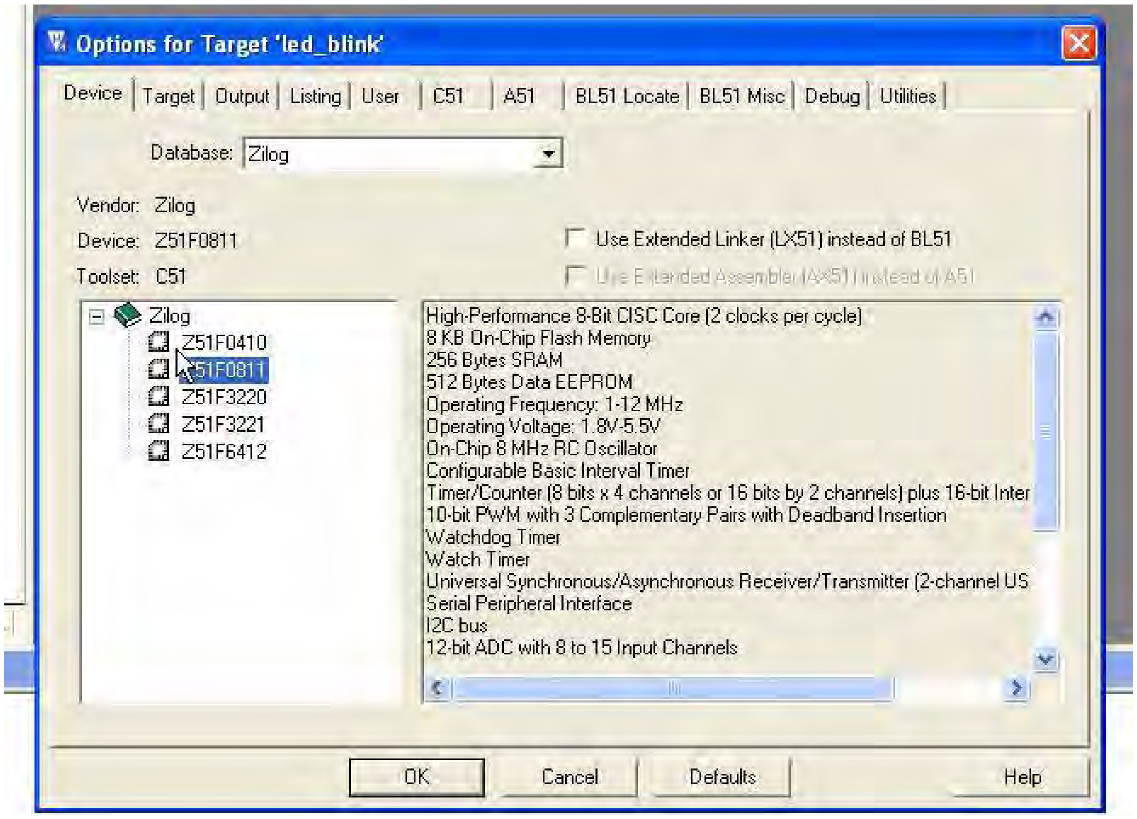
2. Click the **Device** tab. In the **Database** drop-down menu, select **Zilog**, as indicated in Figure 7.



High-Performance 8-Bit CISC Core (2 clocks per cycle)
8 KB On-Chip Flash Memory
256 Bytes SRAM
512 Bytes Data EEPROM
Operating Frequency: 1-12 MHz
Operating Voltage: 1.8V-5.5V
On-Chip 8 MHz RC Oscillator
Configurable Basic Interval Timer
Timer/Counter (8 bits x 4 channels or 16 bits by 2 channels) plus 16-bit Inter
10-bit PWM with 3 Complementary Pairs with Deadband Insertion
Watchdog Timer
Watch Timer
Universal Synchronous/Asynchronous Receiver/Transmitter (2-channel US
Serial Peripheral Interface
I2C bus
12-bit ADC with 8 to 15 Input Channels

Figure 7. Selecting The Zilog Database

3. In the left-hand pane, click to expand the list of Zilog parts, as shown in Figure 8. All of the Z8051 Family processors are listed; simply select the pertinent processor with which you will build your application, and click **OK**.



Database: Zilog

Vendor: Zilog

Device: Z51F0811

Toolset: C51

Use Extended Linker (LX51) instead of BL51

Use Extended Assembler (AX51) instead of A51

Zilog
- Z51F0410
- Z51F0811
- Z51F3220
- Z51F3221
- Z51F6412

High-Performance 8-Bit CISC Core (2 clocks per cycle)
8 KB On-Chip Flash Memory
256 Bytes SRAM
512 Bytes Data EEPROM
Operating Frequency: 1-12 MHz
Operating Voltage: 1.8V-5.5V
On-Chip 8 MHz RC Oscillator
Configurable Basic Interval Timer
Timer/Counter (8 bits x 4 channels or 16 bits by 2 channels) plus 16-bit Inter
10-bit PWM with 3 Complementary Pairs with Deadband Insertion
Watchdog Timer
Watch Timer
Universal Synchronous/Asynchronous Receiver/Transmitter (2-channel US
Serial Peripheral Interface
I2C bus
12-bit ADC with 8 to 15 Input Channels

Figure 8. Selecting The Z8051 Part

Project Source Code

Now we'll return to preparing to build the project; observe the following procedure.

1. Click the `[+]` box next to `led_blink` in the left-hand uVision pane.
2. Open the item labeled **Source**, which represents all of the source code for the project. Click again on the plus sign icon `[+]` to reveal the three source code files that have been placed into this project. These files are `startup.a51`, `main.c`, and `usarts.c`, as shown in Figure 9.

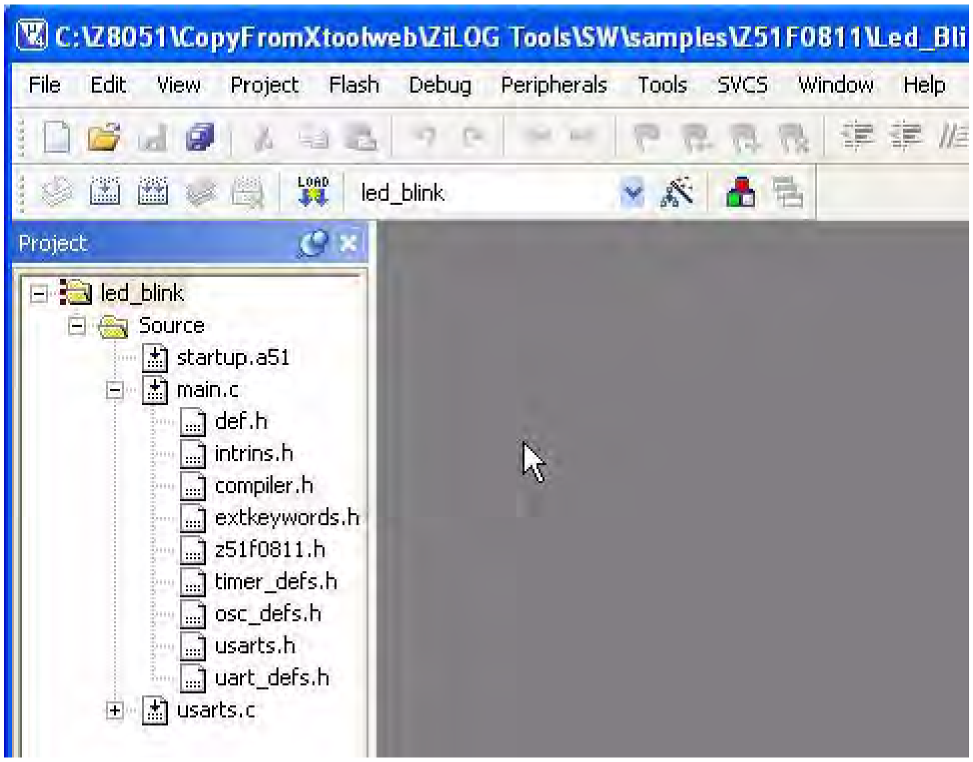


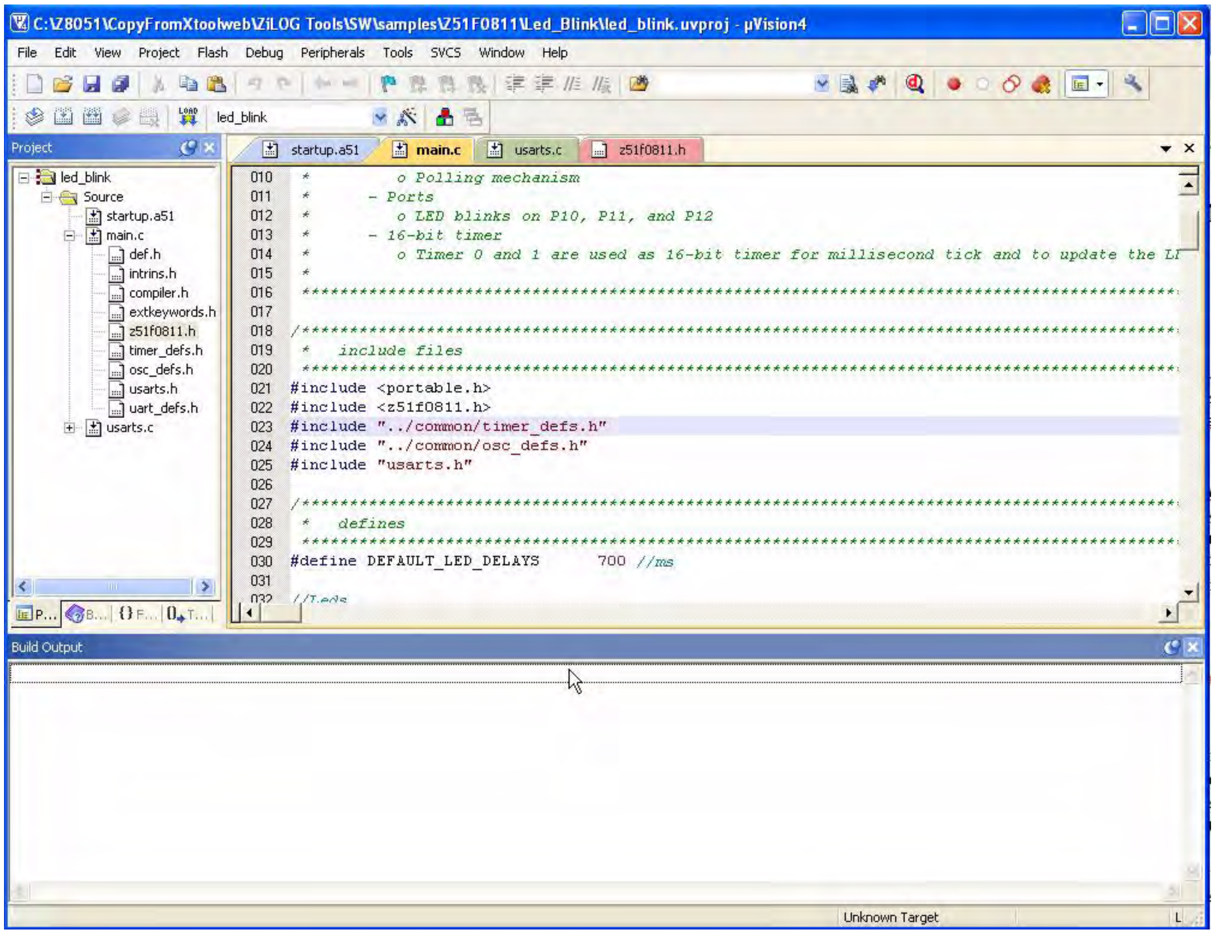
Figure 9. Expanded Source Code

The `startup.a51` file is 8051 assembly code provided by Keil which sets up the C environment. Start-up code such as the `startup.a51` file is always necessary in C applications that run in an embedded environment in which there is no desktop-style operating system, although the start-up code is not always explicitly shown as part of a project by all development tool sets.

The start-up module provides essential support that enables your compiled C code to be linked into a working executable. This code usually is not required to be modified unless you use a particularly customized memory mapping in your application.

The two C source code files, `main.c` and `uarts.c`, define this particular application. Briefly, this program displays a short message to the user, then starts blinking the 3 colored LEDs in a given sequence. If the user presses a key or hits a switch on the board, the direction of LED blinking reverses (i.e yellow, red, green changes to green, red, yellow or vice versa).

Note that you can double-click any of the source code files, such as `main.c`, to display its source code in the large, right-hand pane of the IDE, in which you can also edit the file; see Figure 10. Additionally, after you have built the project the first time, a plus sign icon (`[+]`) is placed next to files such as `main.c` that include other files; you can click the `[+]` icon to view the full list of included header files, and you can double-click any of those in turn to view or edit the header file contents.



```
 *          o Polling mechanism
 *       - Ports
 *          o LED blinks on P10, P11, and P12
 *       - 16-bit timer
 *          o Timer 0 and 1 are used as 16-bit timer for millisecond tick and to update the LI
 *
 ******************************************************************************************

/******************************************************************************************
 *   include files
 ******************************************************************************************
#include <portable.h>
#include <z51f0811.h>
#include "../common/timer_defs.h"
#include "../common/osc_defs.h"
#include "usarts.h"

/******************************************************************************************
 *   defines
 ******************************************************************************************
#define DEFAULT_LED_DELAYS      700 //ms

//Leds
```

Figure 10. Opened Source Code

The Build Step

To build the application in the uVision IDE, all you have to do is to click **Project** → **Rebuild All Target Files**, as indicated in Figure 11. The results of the build are shown in Figure 12.

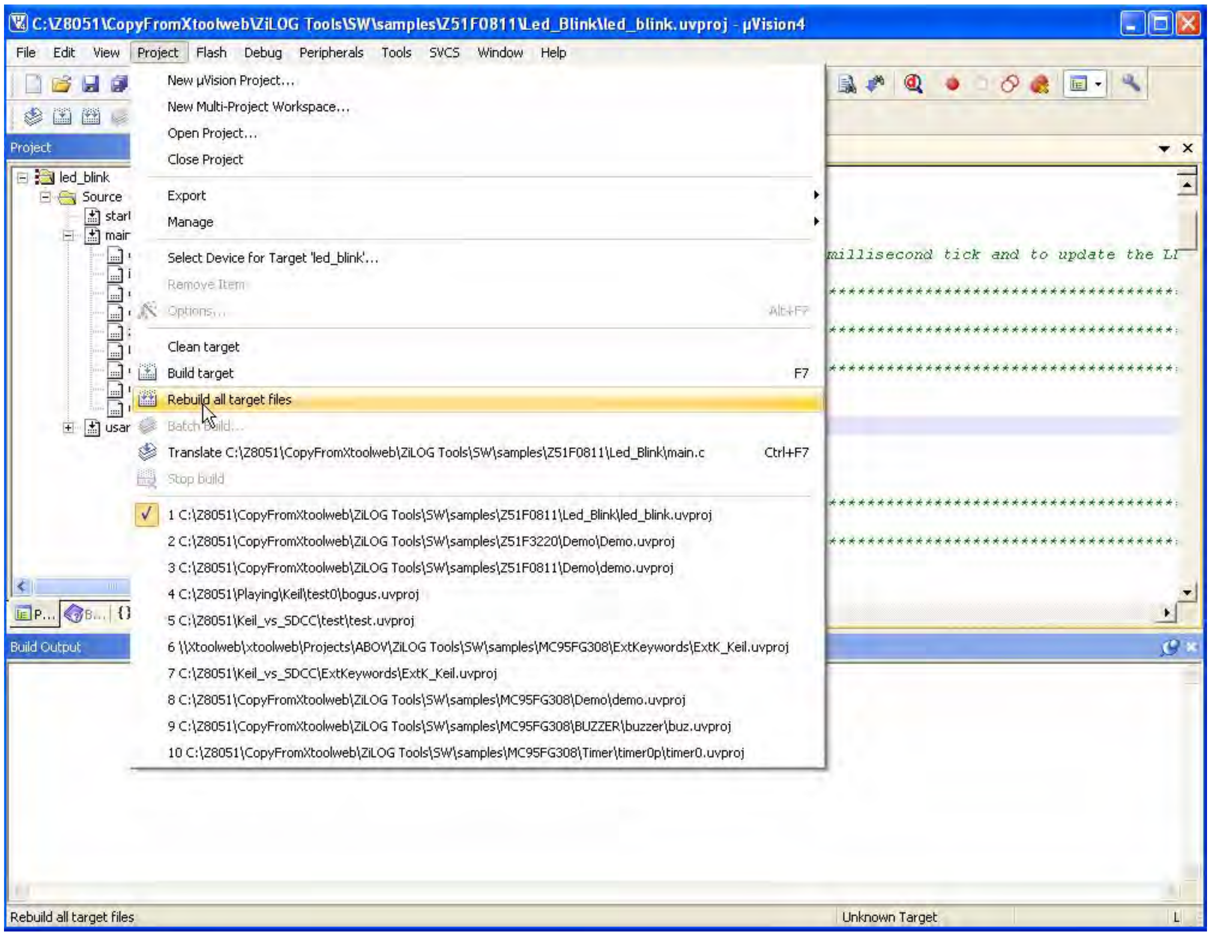
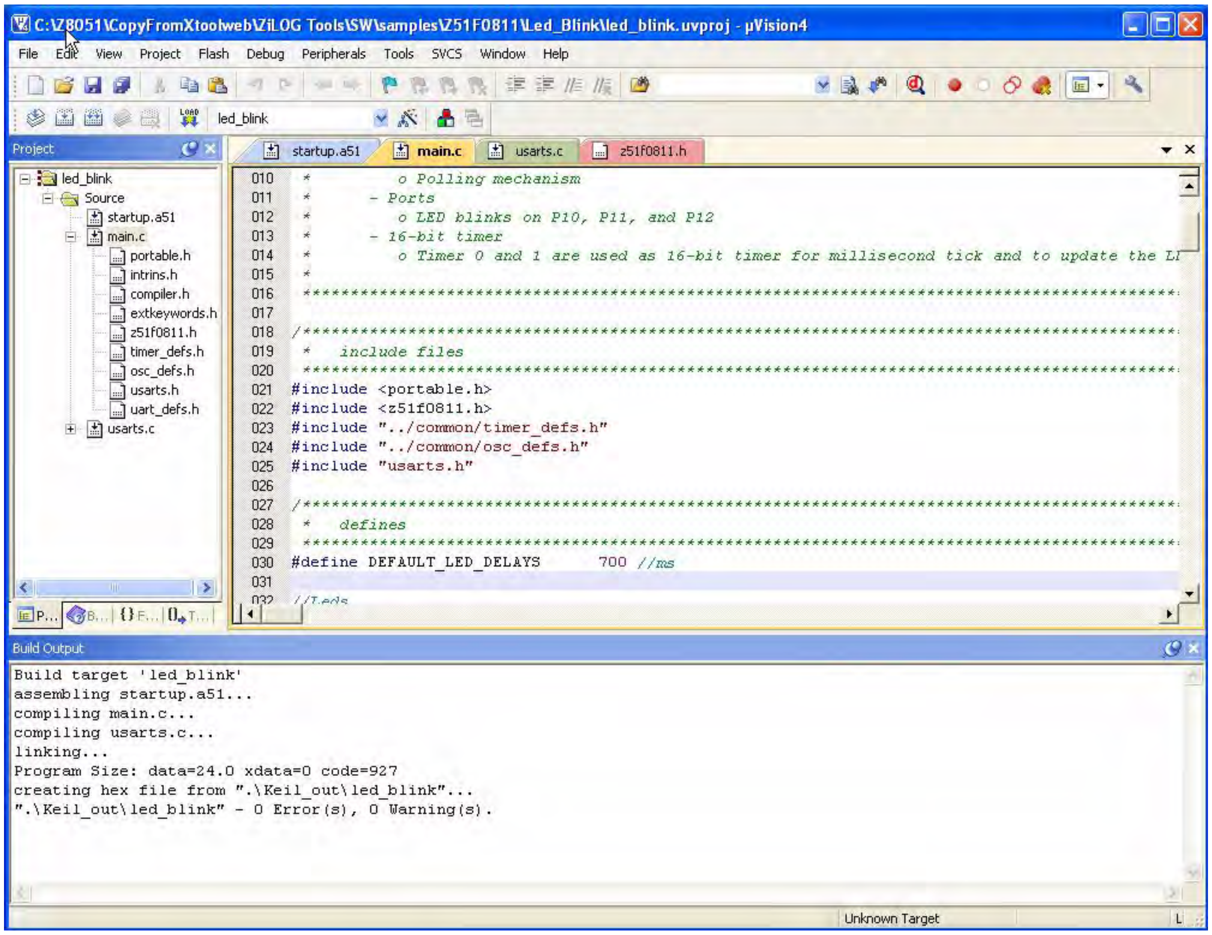

C:\Z8051\CopyFromXtoolweb\ZiLOG Tools\SW\samples\Z51F0811\Led_Blink\led_blink.uvproj - µVision4
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
New µVision Project...
New Multi-Project Workspace...
Open Project...
Close Project
Export
Manage
Select Device for Target 'led_blink'...
Remove Item
Options... Alt+F7
Clean target
Build target F7
Rebuild all target files
Batch Build...
Translate C:\Z8051\CopyFromXtoolweb\ZiLOG Tools\SW\samples\Z51F0811\Led_Blink\main.c Ctrl+F7
Stop build
1 C:\Z8051\CopyFromXtoolweb\ZiLOG Tools\SW\samples\Z51F0811\Led_Blink\led_blink.uvproj
2 C:\Z8051\CopyFromXtoolweb\ZiLOG Tools\SW\samples\Z51F3220\Demo\Demo.uvproj
3 C:\Z8051\CopyFromXtoolweb\ZiLOG Tools\SW\samples\Z51F0811\Demo\demo.uvproj
4 C:\Z8051\Playing\Keil\test0\bogus.uvproj
5 C:\Z8051\Keil_vs_SDCC\test\test.uvproj
6 \\Xtoolweb\xtoolweb\Projects\ABOV\ZiLOG Tools\SW\samples\MC95FG308\ExtKeywords\ExtK_Keil.uvproj
7 C:\Z8051\Keil_vs_SDCC\ExtKeywords\ExtK_Keil.uvproj
8 C:\Z8051\CopyFromXtoolweb\ZiLOG Tools\SW\samples\MC95FG308\Demo\demo.uvproj
9 C:\Z8051\CopyFromXtoolweb\ZiLOG Tools\SW\samples\MC95FG308\BUZZER\buzzer\buz.uvproj
10 C:\Z8051\CopyFromXtoolweb\ZiLOG Tools\SW\samples\MC95FG308\Timer\timer0p\timer0.uvproj
Project
led_blink
Source
Build Output
Rebuild all target files
Unknown Target

Figure 11. Initiating The Build



```
 *         o Polling mechanism
 *       - Ports
 *         o LED blinks on P10, P11, and P12
 *       - 16-bit timer
 *         o Timer 0 and 1 are used as 16-bit timer for millisecond tick and to update the LI
 *
 *******************************************************************************************

/*******************************************************************************************
 *   include files
 *******************************************************************************************
#include <portable.h>
#include <z51f0811.h>
#include "../common/timer_defs.h"
#include "../common/osc_defs.h"
#include "usarts.h"

/*******************************************************************************************
 *   defines
 *******************************************************************************************
#define DEFAULT_LED_DELAYS      700 //ms

//Leds
```

```
Build target 'led_blink'
assembling startup.a51...
compiling main.c...
compiling usarts.c...
linking...
Program Size: data=24.0 xdata=0 code=927
creating hex file from ".\Keil_out\led_blink"...
".\Keil_out\led_blink" - 0 Error(s), 0 Warning(s).
```

Figure 12. Results Of The Build

The Build Output pane at the bottom of the IDE shown in Figure 12 displays a series of messages about each of the source code files being assembled or compiled, along with any error or warning messages; in this case, you should receive no such messages. The Build Output pane will next indicate that the application is being linked. If there are no errors or warnings in the linking process (as would be appropriate in this sample project), it will then display some data about the size of the compiled program and report that it is creating the hex file, which is the ultimate output of the build. As is the usual case with IDEs, the build process is extremely simple, as long as you have set up the project correctly and no errors occur.

Creating A New Project

If you have not already had a defined project, or want to start a new one, the procedure is quite simple and is probably familiar from your experience with other IDEs.

1. Begin by selecting **Project → New uVision Project**, as indicated in Figure 13. The **Create New Project** dialog box will appear.

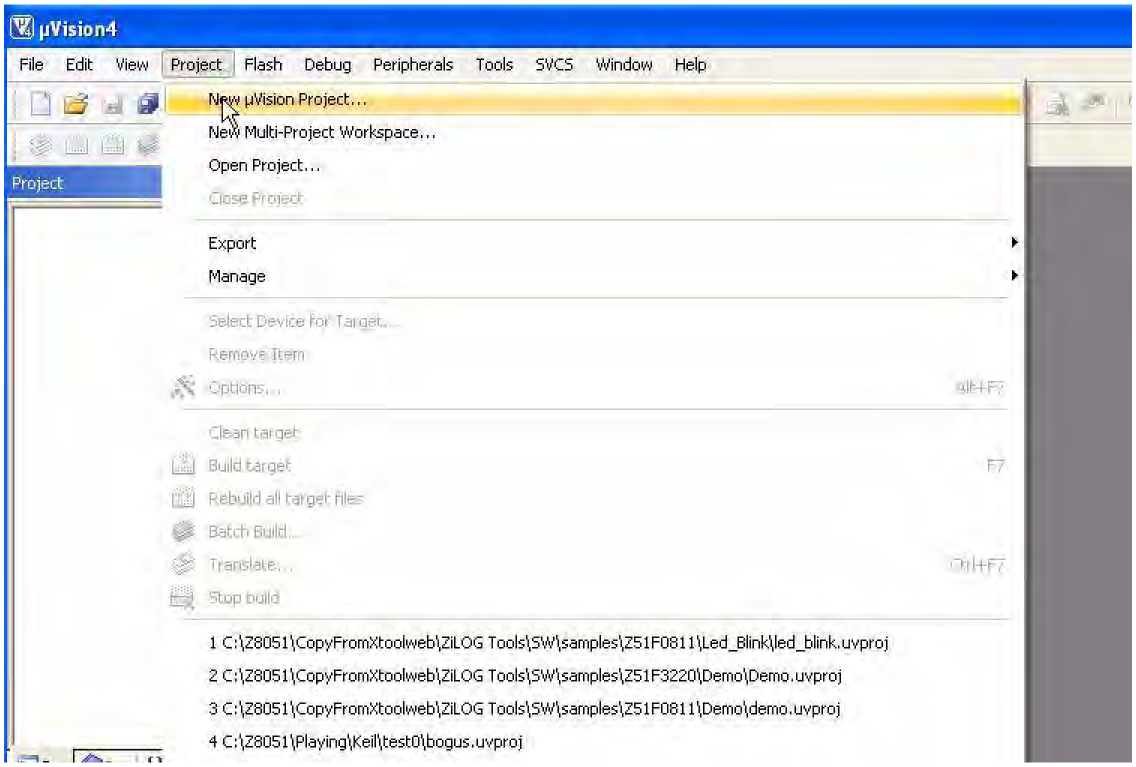

µVision4
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Project
New µVision Project…
New Multi-Project Workspace…
Open Project…
Close Project
Export
Manage
Select Device for Target…
Remove Item
Options… Alt+F7
Clean target
Build target F7
Rebuild all target files
Batch Build…
Translate… Ctrl+F7
Stop build
1 C:\Z8051\CopyFromXtoolweb\ZiLOG Tools\SW\samples\Z51F0811\Led_Blink\led_blink.uvproj
2 C:\Z8051\CopyFromXtoolweb\ZiLOG Tools\SW\samples\Z51F3220\Demo\Demo.uvproj
3 C:\Z8051\CopyFromXtoolweb\ZiLOG Tools\SW\samples\Z51F0811\Demo\demo.uvproj
4 C:\Z8051\Playing\Keil\test0\bogus.uvproj

Figure 13. Creating A New Project

2. In this **Create New Project** dialog, select or create a directory for your project, then give your project a name.
3. When prompted whether you want to copy the standard 8051 start-up code into your project; select **Yes** to give your project a target (i.e., the executable to be built) called *Target 1*, wherein the source code consists of the start-up assembly code module.
4. To add a source code file that already exists to this project, select **Project** → **Manage** → **Components, Environment, Books**, as indicated in Figure 14.

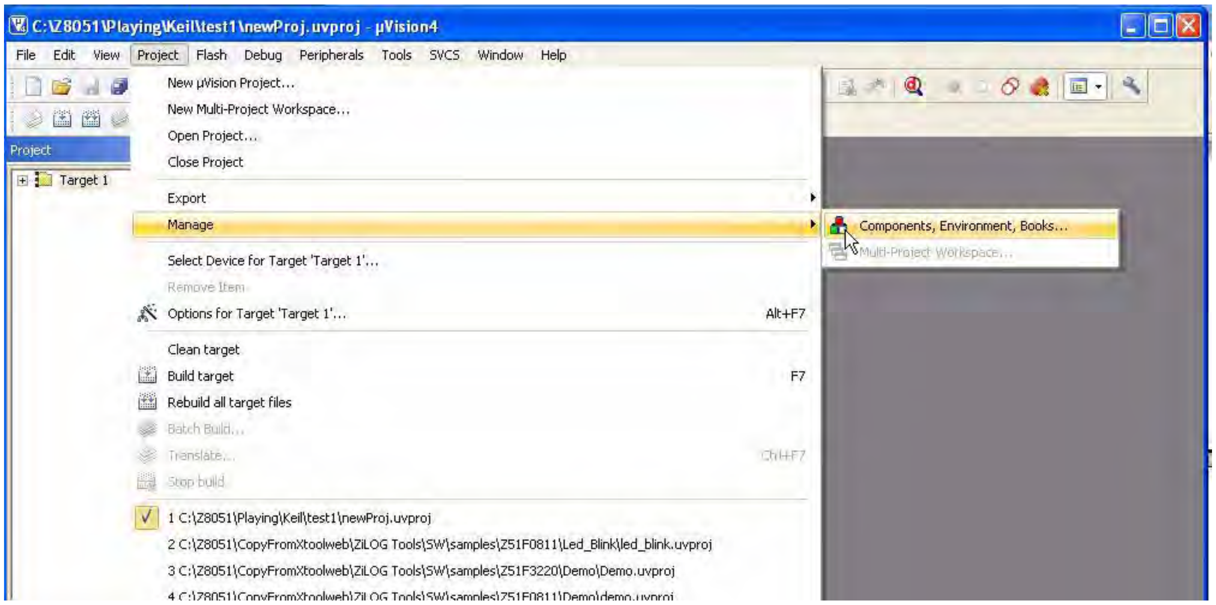

C:\Z8051\Playing\Keil\test1\newProj.uvproj - µVision4
File
Edit
View
Project
Flash
Debug
Peripherals
Tools
SVCS
Window
Help
Project
Target 1
New µVision Project...
New Multi-Project Workspace...
Open Project...
Close Project
Export
Manage
Select Device for Target 'Target 1'...
Remove Item
Options for Target 'Target 1'... Alt+F7
Clean target
Build target F7
Rebuild all target files
Batch Build...
Translate... Ctrl+F7
Stop build
1 C:\Z8051\Playing\Keil\test1\newProj.uvproj
2 C:\Z8051\CopyFromXtoolweb\ZiLOG Tools\SW\samples\Z51F0811\Led_Blink\led_blink.uvproj
3 C:\Z8051\CopyFromXtoolweb\ZiLOG Tools\SW\samples\Z51F3220\Demo\Demo.uvproj
Components, Environment, Books...
Multi-Project Workspace...

Figure 14. Adding An Existing File To The Project

5. In the **Components, Environment, Books** dialog that appears, click the **Add Files** button, shown in Figure 15. Navigate your way to your source code, select the file(s), and click **OK**.

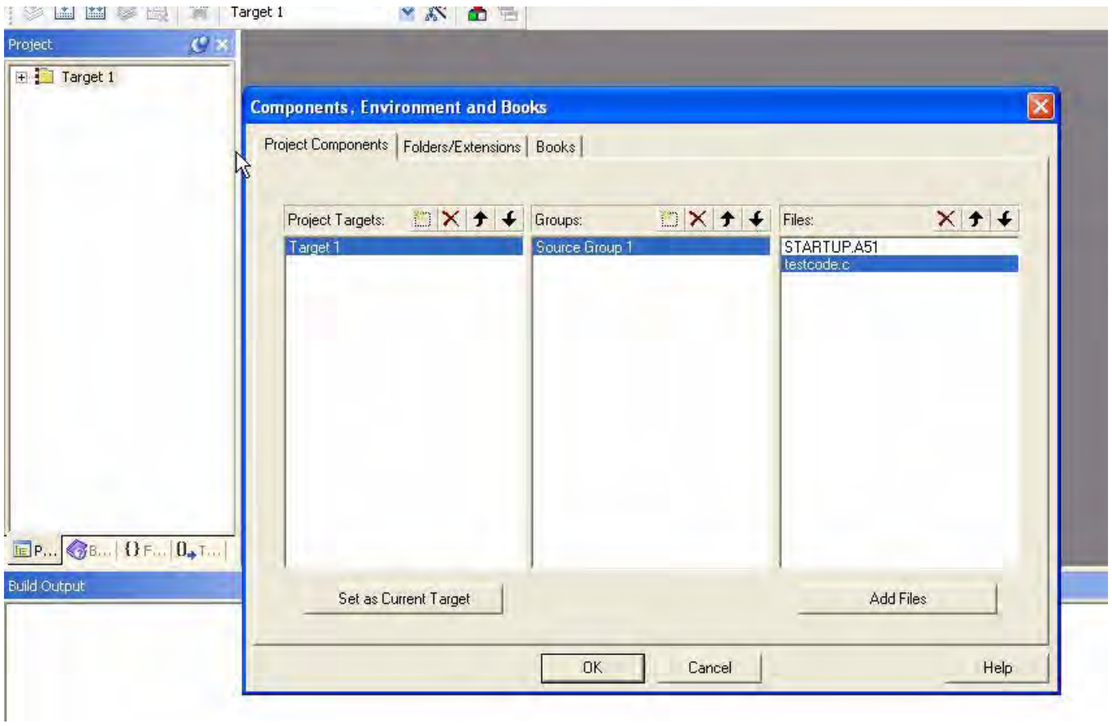


Figure 15. Adding The testcode.c File To The Project

As an alternative or addition to Steps 4 and 5, you may wish to create a brand-new source code file by selecting **File** → **New…**. A new file will be opened with the filename *Text1*. You can change this filename by navigating to **File** → **Save As**, then applying a name with the proper file extension, such as *myCode.c*. Next, you can again use **Project** → **Manage** to add this new file to your project. You could also change the name of the target by using **Project** → **Manage** → **Components, Environment, Books**, click the first (new target) icon in the Project Targets pane, create a new target with a different name, then click **Set as Current Target**.

Project Options

There is much more to explore in the Keil tools than we can hope to cover in this brief guide. The best way to start digging into these tools is to look at the large number of project options, which control all aspects of the build process. To view all options for the `led_blink` project, highlight the top-level target, `led_blink` in this case, in the left-hand pane. Next, select **Project** → **Options** for the `led_blink` target to display the full project options dialog box.

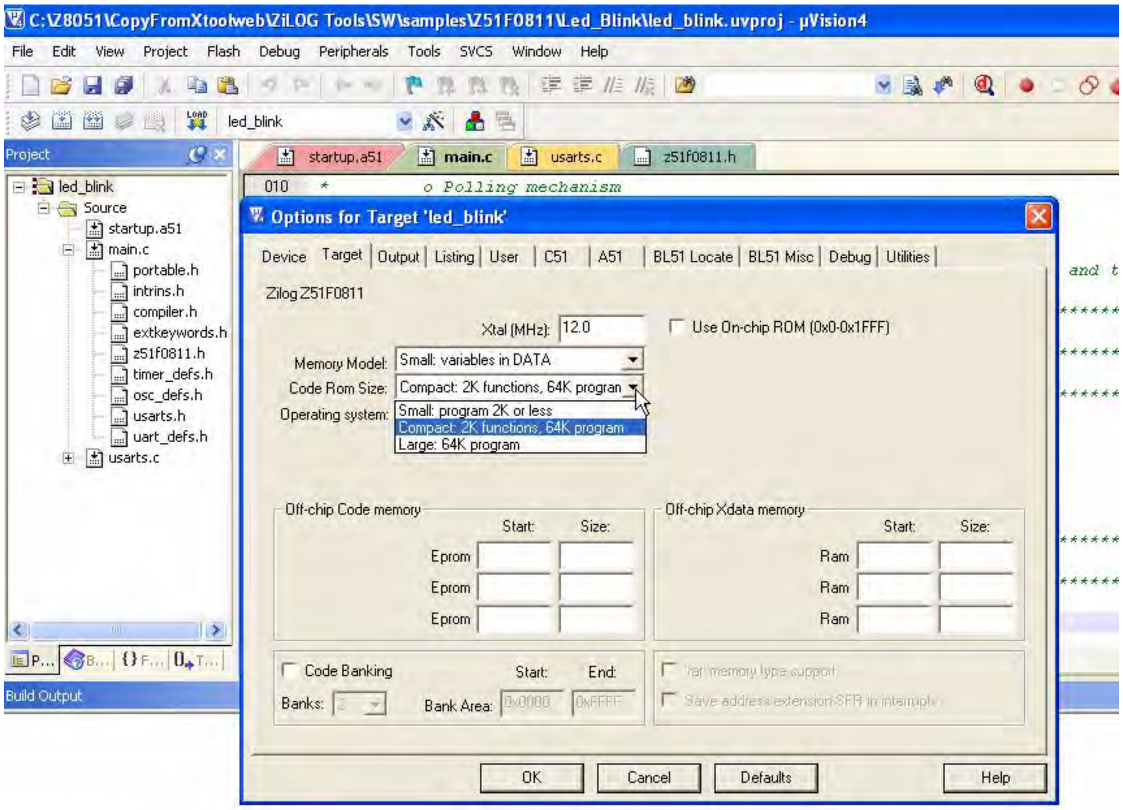
It is well worth the time to explore each tab and the options that are offered in this dialog. Many of these can be familiar to you from other IDEs. We will comment briefly on a couple of settings that can cause problems that you may not have anticipated as you try to build some simple projects.

Potential Trouble Areas

One very important setting when using the free tool set in particular can be found on the **Target** tab, under **Code ROM Size**. For virtually all projects, you must modify this setting to either **Compact** or **Large**. Using the **Small** setting (which is the default setting if you start uVision from scratch) almost always results in the following Linker error message:

```
L121: IMPROPER FIXUP
```

This error indicates that the Linker is unable to create a linked application within the **Small** setting's very small limit for the entire code space. Figure 16 shows an example of setting this option.



C:\Z8051\CopyFromXtoolweb\ZiLOG Tools\SW\samples\Z51F0811\Led_Blink\led_blink.uvproj - µVision4

010 * o Polling mechanism

Options for Target 'led_blink'

Zilog Z51F0811

Xtal (MHz): 12.0

Use On-chip ROM (0x0-0x1FFF)

Memory Model: Small: variables in DATA

Code Rom Size: Compact: 2K functions, 64K program

Operating system:

Small: program 2K or less
Compact: 2K functions, 64K program
Large: 64K program

Code Banking

Banks: 2

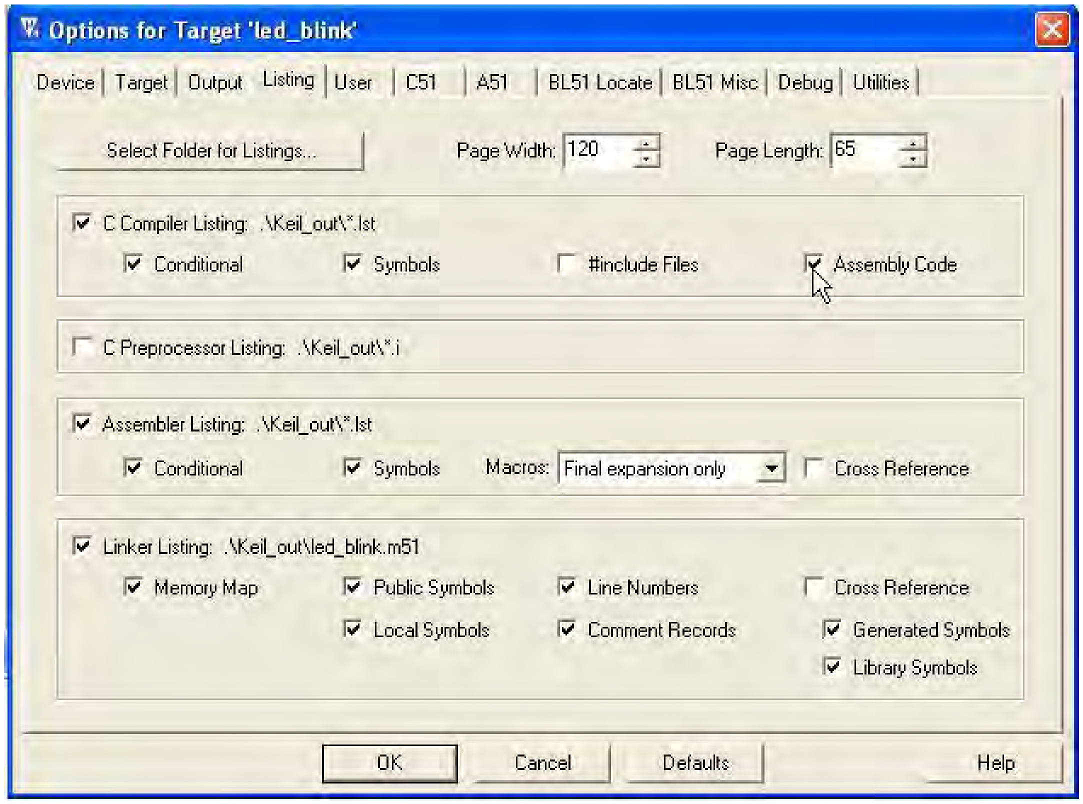
Bank Area: 0x0000 0xFFFF

'far' memory type support

Save address extension SFR in interrupts

Figure 16. Setting The Code ROM Size To Avoid An Improper Fix-Up Linker Error

Another minor quirk of the Keil tools is that it is a bit difficult to obtain a listing of generated assembly code when building a C project. The solution is to make sure that the **Assembly Code** checkbox is selected in the **C Compiler Listing** panel, as indicated in Figure 17.


Options for Target 'led_blink'

Device | Target | Output | Listing | User | C51 | A51 | BL51 Locate | BL51 Misc | Debug | Utilities

Select Folder for Listings... Page Width: 120 Page Length: 65

- [✓] C Compiler Listing: .\Keil_out*.lst
  - [✓] Conditional
  - [✓] Symbols
  - [ ] #include Files
  - [✓] Assembly Code
- [ ] C Preprocessor Listing: .\Keil_out*.i
- [✓] Assembler Listing: .\Keil_out*.lst
  - [✓] Conditional
  - [✓] Symbols
  - Macros: Final expansion only
  - [ ] Cross Reference
- [✓] Linker Listing: .\Keil_out\led_blink.m51
  - [✓] Memory Map
  - [✓] Public Symbols
  - [✓] Line Numbers
  - [ ] Cross Reference
  - [✓] Local Symbols
  - [✓] Comment Records
  - [✓] Generated Symbols
  - [✓] Library Symbols

Figure 17. Ensuring A Generated Assembly Code Listing

Selecting the **Assembly Code** checkbox ensures that a listing of the generated assembly code will be displayed at the end of each `<filename>.lst` file that lists the C code. That information can be combined with the Linker listing file (often called a map file in other tool environments), `<target_name>.m51`, to determine the actual addresses of modules and instructions in your linked code, if appropriate. Be sure that you check the box for **Listing** → **Linker Listing** if you want to view that file.

The limitation of the Keil evaluation tools will be seen if you try to build an application that exceeds the limit of 2K (`0x800` bytes) of application code. In this case, the Compiler and Linker still run and report any error or warning messages that might result from code errors or improper project options. However, at the end of the linker output, it will display the following message:

```
FATAL ERROR L250: CODE SIZE LIMIT IN RESTRICTED VERSION EXCEEDED
```

As a result of this error, no output hex file will be created.

Note: The issue discussed in the XRAM Initialization section on page 18 does not apply when you are building your application with the Keil tools. By default, the Keil tools use the safer, dual-pointer method for XRAM initialization.

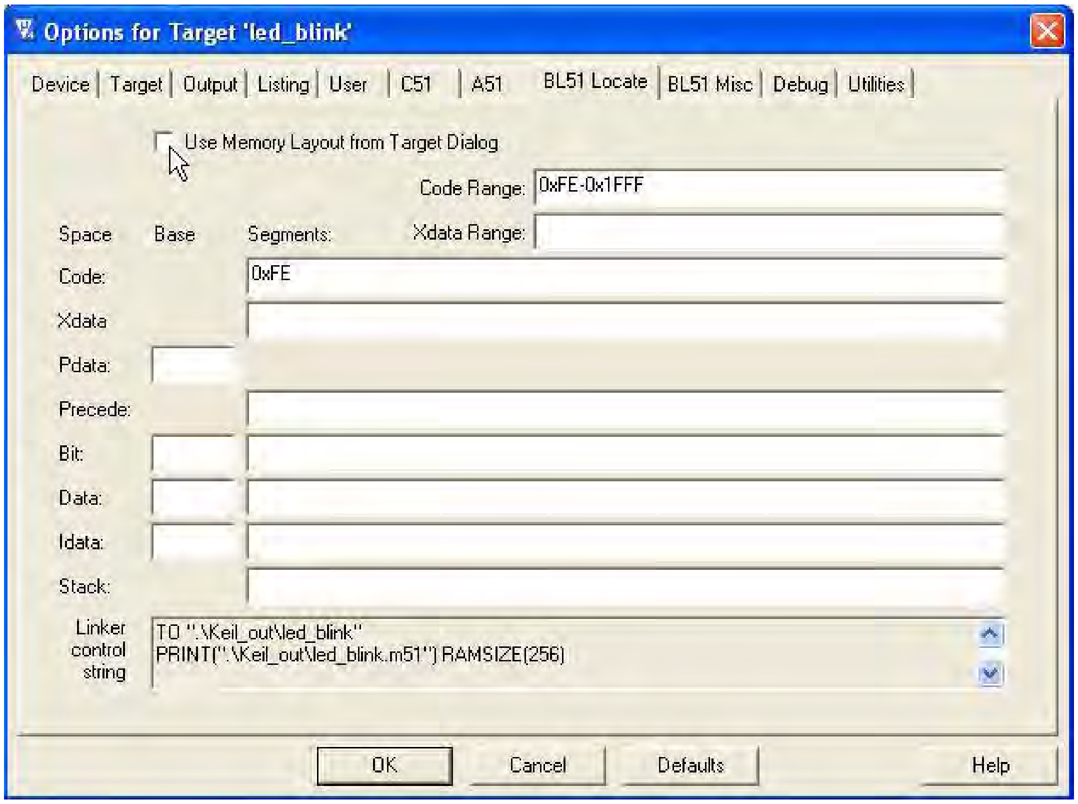
Vector Table Allocation

The Controlling The Memory Map section on page 4 explains why the Linker should be instructed to avoid placing executable code in the address space used by the interrupt vector table. In the Keil tools, this instruction is set using the linker project options, as follows:

1. Open the Project Options dialog box and click the **BL51 Locate** tab.
2. Uncheck the **Use Memory Layout from Target Dialog** box to turn off the default memory mapping.
3. Set a range for code memory which the Linker can use for normal executable code, thereby avoiding the vector table.

In the example we examined in the Controlling The Memory Map section, the vector table must occupy addresses `0x00–0xFD`; therefore, `0xFE` should be the start address for the code range. The end address depends on your MCU specifications; check the Z8051 data sheet for your particular device. For the Z51F0811 MCU, this end address is `0x1FFF`.

This address range can be inserted in either the **Code Range** field or the **Code** field of the **BL51 Locate** tab, as shown in Figures 18 and 19.



☐ Use Memory Layout from Target Dialog

Code Range: 0xFE-0x1FFF

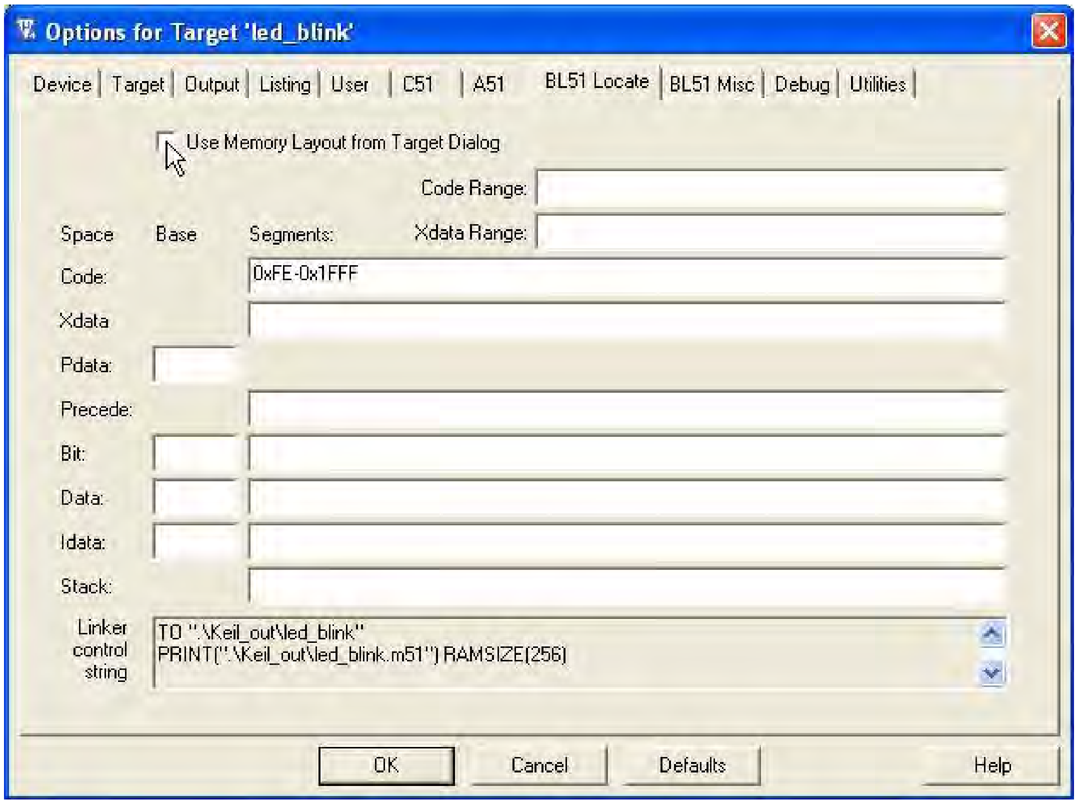
Xdata Range:

Space	Base	Segments:
Code:		0xFE
Xdata		
Pdata:		
Precede:		
Bit:		
Data:		
Idata:		
Stack:		

Linker control string:

```
TO ".\Keil_out\led_blink"
PRINT(".\Keil_out\led_blink.m51") RAMSIZE(256)
```

Figure 18. Vector Table Allocation, Method 1


Options for Target 'led_blink'
Device | Target | Output | Listing | User | C51 | A51 | BL51 Locate | BL51 Misc | Debug | Utilities
OK
Cancel
Defaults
Help

Use Memory Layout from Target Dialog

Code Range:

Xdata Range:

Space	Base	Segments:
Code:		0xFE-0x1FFF
Xdata		
Pdata:		
Precede:		
Bit:		
Data:		
Idata:		
Stack:		

Linker control string

```
TO ".\Keil_out\led_blink"
PRINT(".\Keil_out\led_blink.m51") RAMSIZE(256)
```

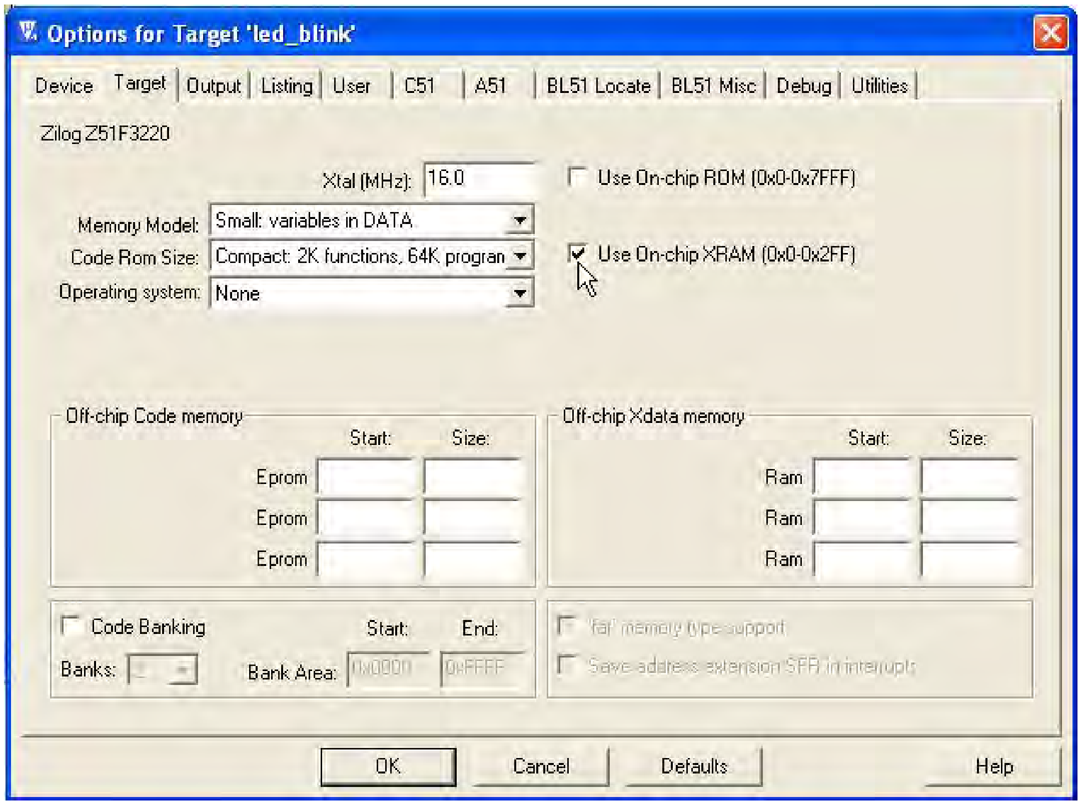
Figure 19. Vector Table Allocation, Method 2

Application-Specific Hardware Memory Mapping

As previously discussed, some Z8051 MCUs map special hardware devices into a portion of the memory map, and you must force the Linker not to use that memory mapped area for normal data storage. For example, in the Z51F3220 MCUs, LCD segments are mapped to external data addresses `0x00` to `0x1A`.

To manage this issue in the Keil toolset, observe the following procedure.

1. Select the **Target** → **Use on-chip XRAM** checkbox, as indicated in Figure 20.



Zilog Z51F3220

Xtal (MHz): 16.0

Memory Model: Small: variables in DATA

Code Rom Size: Compact: 2K functions, 64K progran

Operating system: None

☐ Use On-chip ROM (0x0-0x7FFF)

☑ Use On-chip XRAM (0x0-0x2FF)

Off-chip Code memory

	Start:	Size:
Eprom		
Eprom		
Eprom		

Off-chip Xdata memory

	Start:	Size:
Ram		
Ram		
Ram		

☐ Code Banking   Start: End:

Banks: 2   Bank Area: 0x0000   0xFFFF

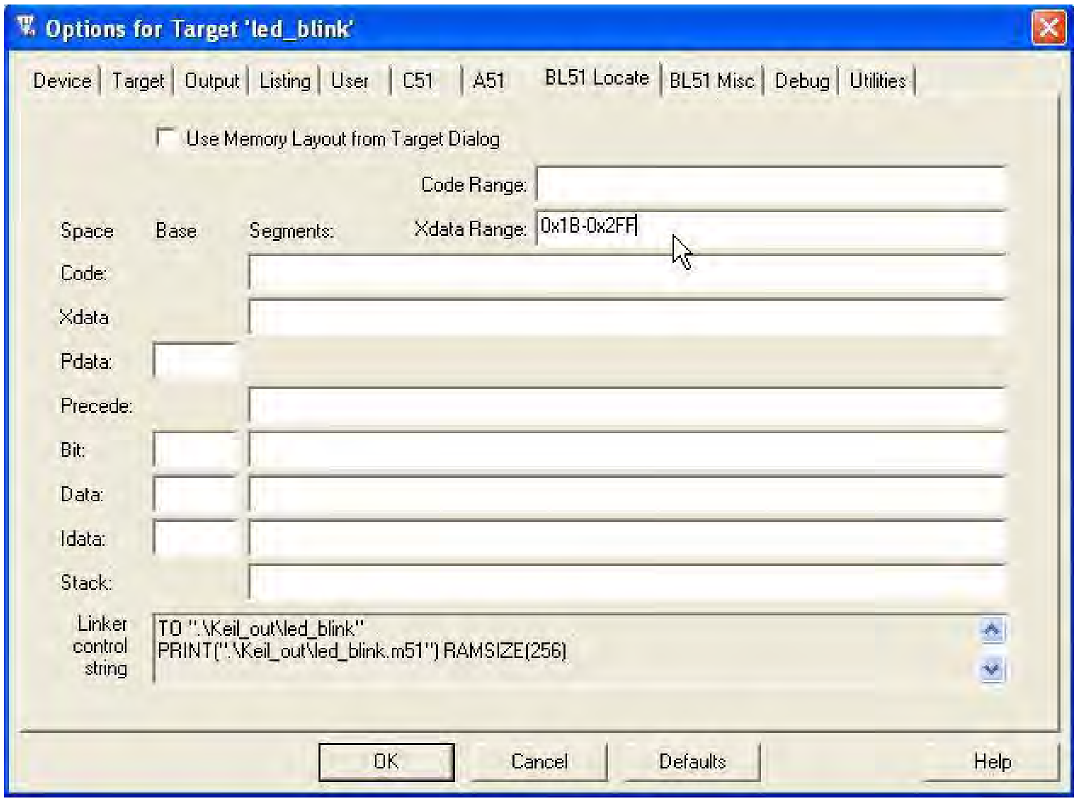
☐ 'far' memory type support

☐ Save address extension SFR in interrupts

Figure 20. On-Chip XRAM

Note: The **Use On-Chip XRAM** option is provided for the Z51F3220 MCU, but is not provided on some other Z8051 MCUs.

2. Click the **BL51 Locate** tab, then deselect the **Use Memory Layout from Target Dialog** checkbox.
3. Enter a range for acceptable XData addresses, in either the XData range or the XData fields (see Figure 21), in a manner similar to setting the range for code memory to exclude the vector table. In this case, the range would start at `0x1B` (therefore excluding the memory-mapped LCD segments) and end at `0x2FF` (the extent of XData in the Z51F3220 memory map).



Use Memory Layout from Target Dialog

Code Range:

Xdata Range: 0x1B-0x2FF

Space Base Segments:

Code:

Xdata

Pdata:

Precede:

Bit:

Data:

Idata:

Stack:

Linker control string

```
TO ".\Keil_out\led_blink"
PRINT(".\Keil_out\led_blink.m51") RAMSIZE(256)
```

Figure 21. XData Range

Debugging with the Keil µVision IDE and Zilog OCD

Zilog's On-Chip Debugger hardware now fully supports the Keil µVision IDE. Our target driver is seamlessly integrated with the Keil debugger, allowing Keil C51 users to work within the µVision4 environment without switching between the Keil compiler and Zilog's external OCD software.

Note: The projects discussed in this document have been tested with the Keil µVision IDE V4.53.0.6 (PK51 Professional Developers Kit) and later versions. To verify the version of the Keil IDE you are using, choose **About µVision...** from the Keil **Help** menu.

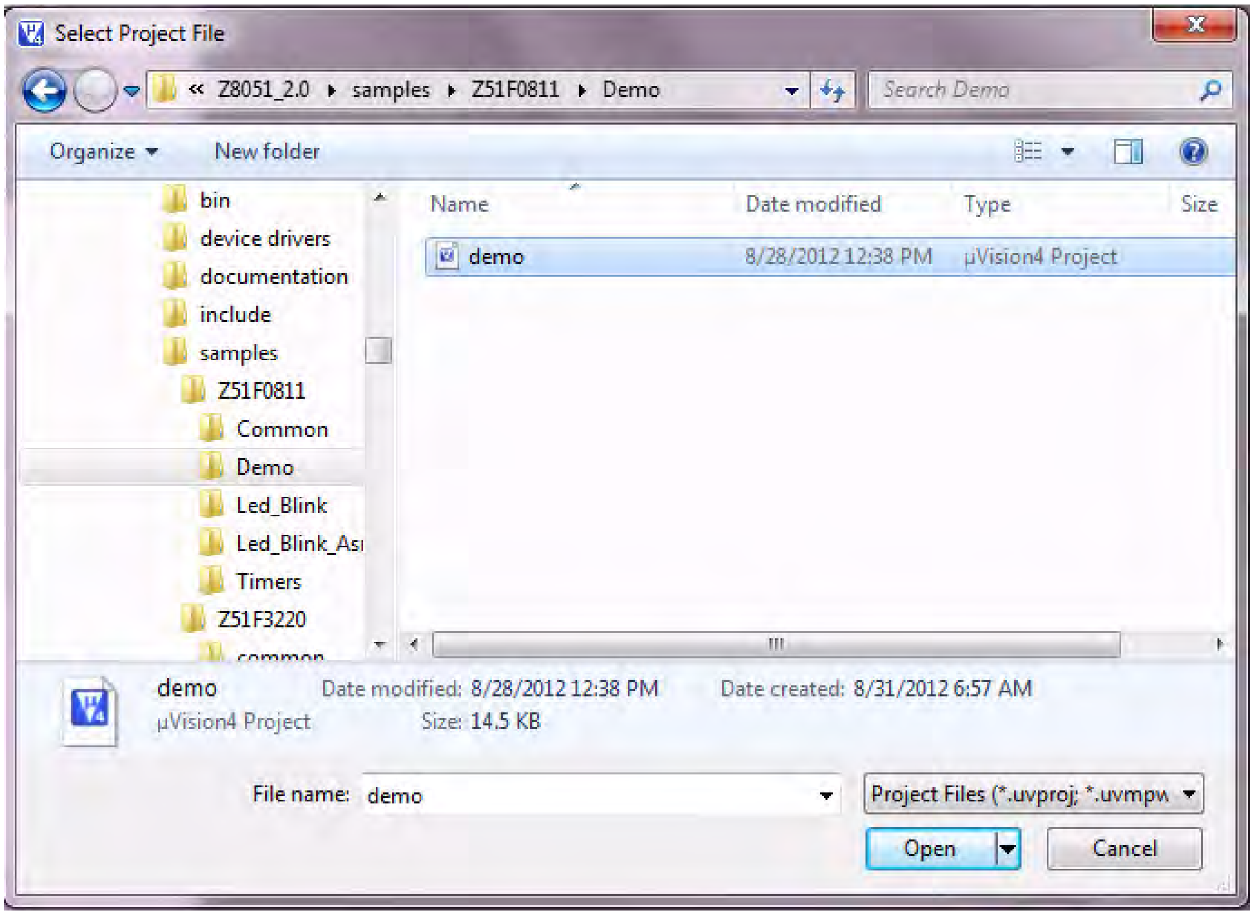
Debugger Configuration

In the demo project example that follows, the Z51F0811 MCU-related project is referenced as `Demo`. Observe the following procedure to run your application with the Zilog OCD target driver.

1. Start the Keil µVsion4 IDE.
2. From the **Project** menu, select **Open Project** and navigate to the following filepath:

   <Installation directory>\Z8051_<version>\samples\Z51F0811\Demo

3. Select the `Demo.uvproj` file and click **Open**; see Figure 22.



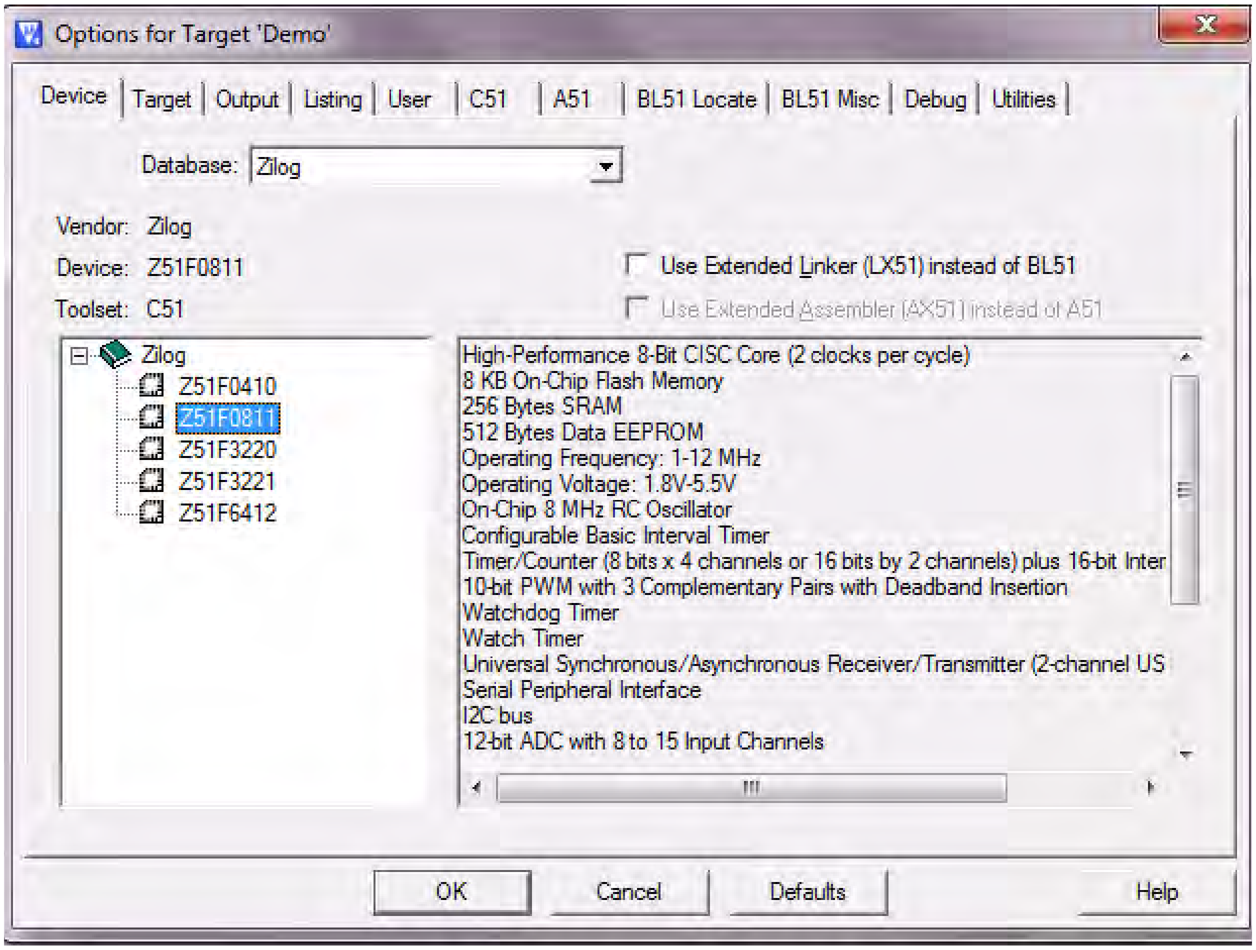
Name	Date modified	Type	Size
demo	8/28/2012 12:38 PM	µVision4 Project	

demo Date modified: 8/28/2012 12:38 PM Date created: 8/31/2012 6:57 AM
µVision4 Project Size: 14.5 KB

File name: demo
Project Files (*.uvproj; *.uvmpw

Figure 22. Selecting the Demo Project File

4. Return to the **Project** menu and select **Options for Target 'Demo'**.
5. In the Options for Target 'Demo' dialog that appears, click the **Device** tab and ensure that your target is properly selected for your project, as illustrated in Figure 23.


Options for Target 'Demo'

Device | Target | Output | Listing | User | C51 | A51 | BL51 Locate | BL51 Misc | Debug | Utilities

Database: Zilog

Vendor: Zilog
Device: Z51F0811
Toolset: C51

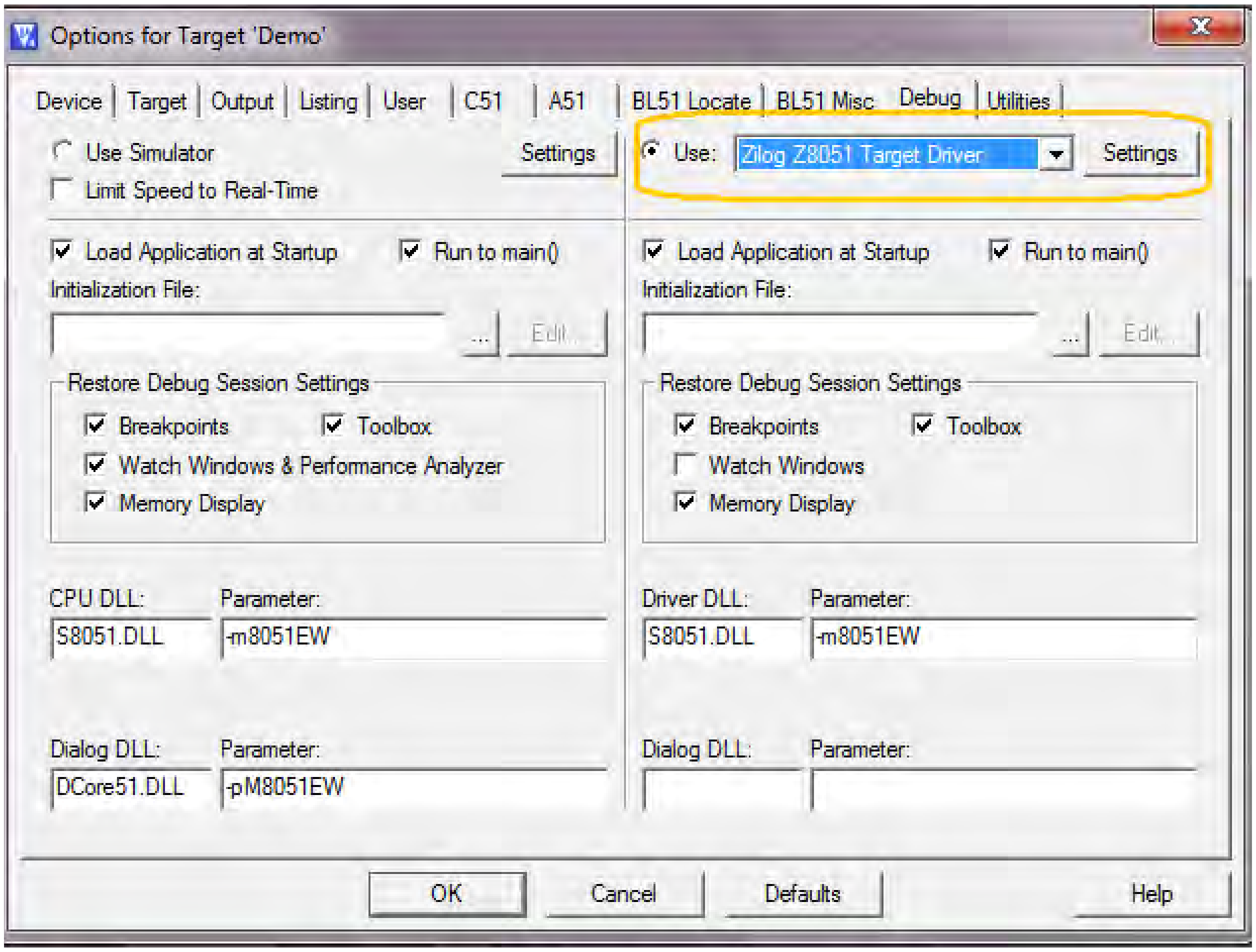
Use Extended Linker (LX51) instead of BL51
Use Extended Assembler (AX51) instead of A51

Zilog
- Z51F0410
- Z51F0811
- Z51F3220
- Z51F3221
- Z51F6412

```
High-Performance 8-Bit CISC Core (2 clocks per cycle)
8 KB On-Chip Flash Memory
256 Bytes SRAM
512 Bytes Data EEPROM
Operating Frequency: 1-12 MHz
Operating Voltage: 1.8V-5.5V
On-Chip 8 MHz RC Oscillator
Configurable Basic Interval Timer
Timer/Counter (8 bits x 4 channels or 16 bits by 2 channels) plus 16-bit Inter
10-bit PWM with 3 Complementary Pairs with Deadband Insertion
Watchdog Timer
Watch Timer
Universal Synchronous/Asynchronous Receiver/Transmitter (2-channel US
Serial Peripheral Interface
I2C bus
12-bit ADC with 8 to 15 Input Channels
```

Figure 23. Selecting the Target

6. After selecting the target, click the **Debug** tab and select the **Zilog Z8051 Target Driver** from the **Use:** drop-down menu, as highlighted in Figure 24.



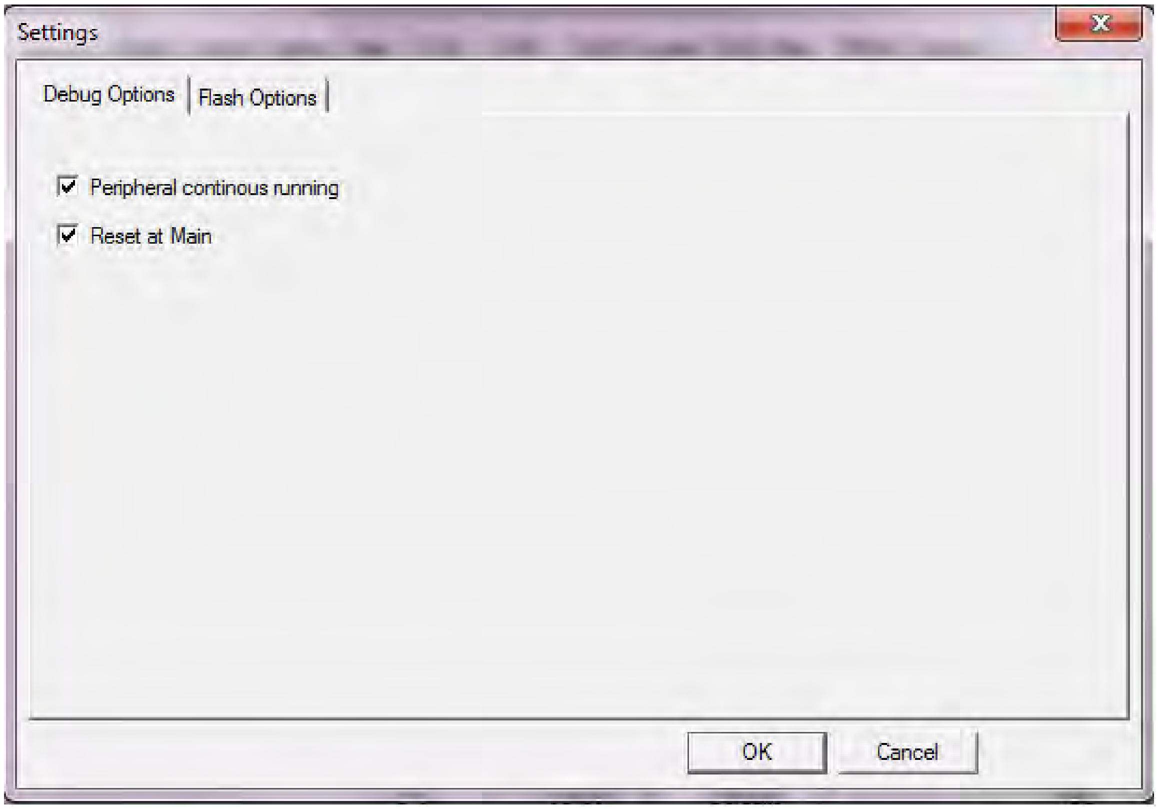
Options for Target 'Demo'

Device | Target | Output | Listing | User | C51 | A51 | BL51 Locate | BL51 Misc | Debug | Utilities

◯ Use Simulator	● Use: Zilog Z8051 Target Driver
☐ Limit Speed to Real-Time	
☑ Load Application at Startup ☑ Run to main()	☑ Load Application at Startup ☑ Run to main()
Initialization File:	Initialization File:
Restore Debug Session Settings	Restore Debug Session Settings
☑ Breakpoints ☑ Toolbox	☑ Breakpoints ☑ Toolbox
☑ Watch Windows & Performance Analyzer	☐ Watch Windows
☑ Memory Display	☑ Memory Display
CPU DLL: S8051.DLL Parameter: -m8051EW	Driver DLL: S8051.DLL Parameter: -m8051EW
Dialog DLL: DCore51.DLL Parameter: -pM8051EW	Dialog DLL: Parameter:

Figure 24. Selecting the Target Driver

7. Click the **Settings** button, located to the right of this drop-down menu, to configure your Debug and Flash options. The Settings dialog is displayed with the Debug Options tab appearing by default, as shown in Figure 25.



☑ Peripheral continous running

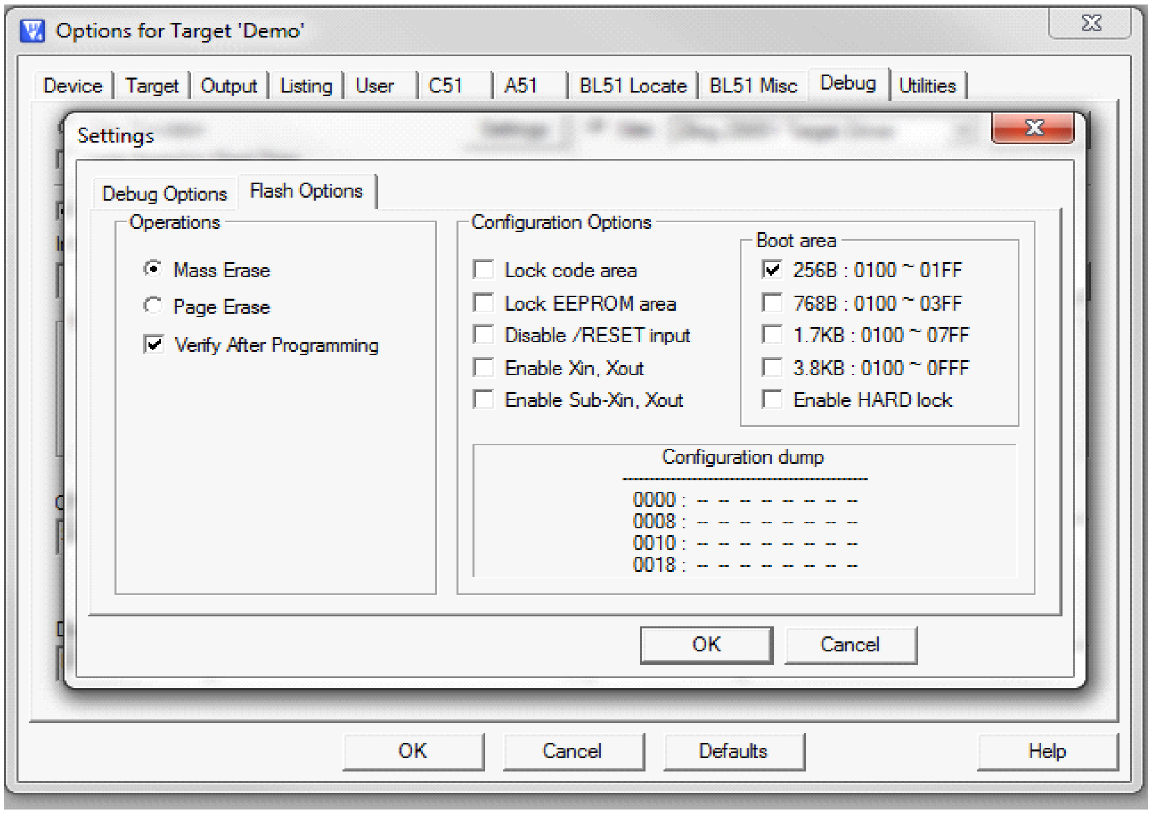
☑ Reset at Main

Figure 25. Configuring the Debug Options

Notes: When configuring the appropriate Debug option, be aware of the following conditions:

- Checking the **Peripheral continues running** option means that the timers used in your project will run while the processor is stopped by the debugger.
- The **Reset at Main** option will only work if you have a `main` file in your project; otherwise you should deselect this option.

8. Click the **Flash Options** tab. The Flash Options Settings dialog will appear, as shown in Figure 26. To select the proper options for Flash programming, refer to the Z51F0811 Product Specification (PS0296).


Options for Target 'Demo'

Device | Target | Output | Listing | User | C51 | A51 | BL51 Locate | BL51 Misc | Debug | Utilities

Settings

Debug Options | Flash Options

Operations
- ● Mass Erase
- ○ Page Erase
- ☑ Verify After Programming

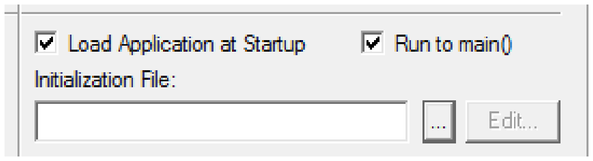
Configuration Options
- ☐ Lock code area
- ☐ Lock EEPROM area
- ☐ Disable /RESET input
- ☐ Enable Xin, Xout
- ☐ Enable Sub-Xin, Xout

Boot area
- ☑ 256B : 0100 ~ 01FF
- ☐ 768B : 0100 ~ 03FF
- ☐ 1.7KB : 0100 ~ 07FF
- ☐ 3.8KB : 0100 ~ 0FFF
- ☐ Enable HARD lock

```
Configuration dump
-----------------------------------
0000 : -- -- -- -- -- -- -- --
0008 : -- -- -- -- -- -- -- --
0010 : -- -- -- -- -- -- -- --
0018 : -- -- -- -- -- -- -- --
```

Figure 26. Configuring the Flash Options

9. Click **OK** to exit the Settings dialog.
10. From the **Options for Target 'Demo'** dialog, select the **Load Application at Startup** checkbox, as shown in Figure 27, the so that the IDE will download the code upon connection. There is no need to select or enter an initialization file.


- ☑ Load Application at Startup
- ☑ Run to main()

Initialization File:

Figure 27. Load Application at Startup Settings

11. Click **OK** to exit the **Options for Target 'Demo'** dialog.

Run the Debugger

The following procedure uses the Z51F0811 MCU as a debugger configuration example of how to connect your Z8051 development board to your PC.

1. Using the USB cable supplied in the kit, connect the Zilog OCD Module to the USB port of the PC.
2. Connect the 10-pin cable to the OCD Module and to the Z51F0811 Evaluation Board, and ensure that the connection appears as shown in Figure 28.

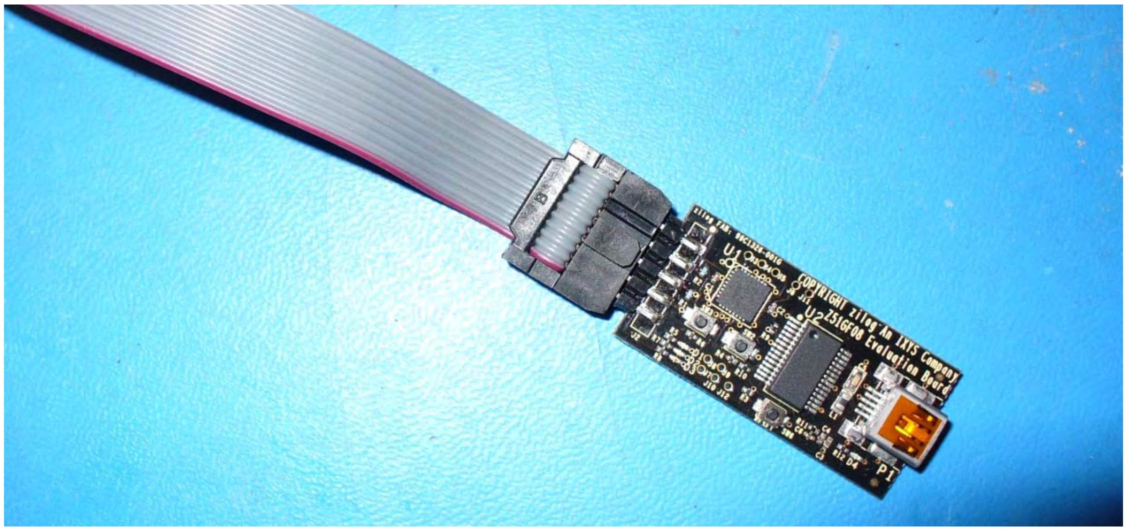


Figure 28. 10-pin Cable Connected to the Z51F0811 Evaluation Board

3. Connect the USB cable to the Z51F0811 Evaluation Board and to the PC. If installing for the first time, the USB driver will be automatically installed on your PC. For more details about this driver installation, please refer to the *FTDI USB-to-UART Driver Installation* section of the Z51F0811 Evaluation Kit User Manual (UM0242).
4. Start your debugging session by clicking the **Start/Stop Debug Session** icon, as indicated in Figure 29. A default Windows configuration of the debug session is shown in Figure 30.

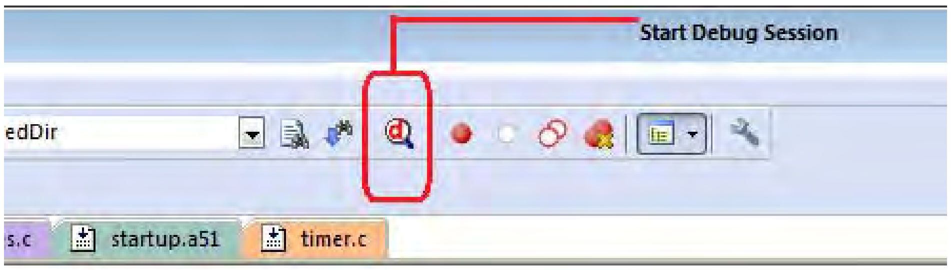
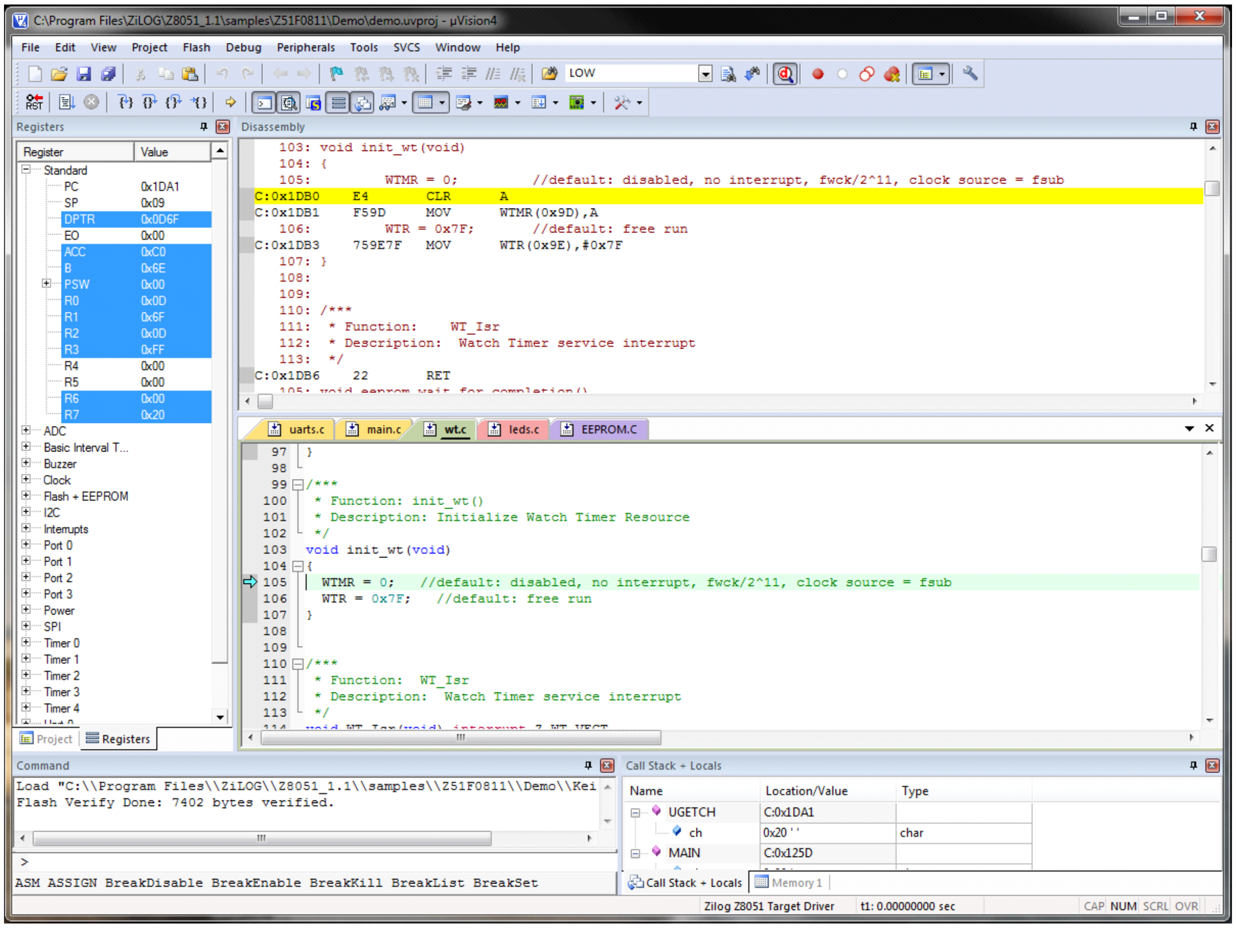


Figure 29. Beginning a Debug Session


C:\Program Files\ZiLOG\Z8051_1.1\samples\Z51F0811\Demo\demo.uvproj - µVision4
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
LOW
Registers
Disassembly
uarts.c
main.c
wt.c
leds.c
EEPROM.C
Project
Registers
Command
Call Stack + Locals
Memory 1
Zilog Z8051 Target Driver
t1: 0.00000000 sec
CAP NUM SCRL OVR

Register	Value
Standard	
PC	0x1DA1
SP	0x09
DPTR	0x0D6F
EO	0x00
ACC	0xC0
B	0x6E
PSW	0x00
R0	0x0D
R1	0x6F
R2	0x0D
R3	0xFF
R4	0x00
R5	0x00
R6	0x00
R7	0x20
ADC	
Basic Interval T...	
Buzzer	
Clock	
Flash + EEPROM	
I2C	
Interrupts	
Port 0	
Port 1	
Port 2	
Port 3	
Power	
SPI	
Timer 0	
Timer 1	
Timer 2	
Timer 3	
Timer 4	

```
   103: void init_wt(void)
   104: {
   105:          WTMR = 0;          //default: disabled, no interrupt, fwck/2^11, clock source = fsub
C:0x1DB0    E4       CLR      A
C:0x1DB1    F59D     MOV      WTMR(0x9D),A
   106:          WTR = 0x7F;        //default: free run
C:0x1DB3    759E7F   MOV      WTR(0x9E),#0x7F
   107: }
   108:
   109:
   110: /***
   111:  * Function:    WT_Isr
   112:  * Description:  Watch Timer service interrupt
   113:  */
C:0x1DB6    22       RET
```

```
}

/***
 * Function: init_wt()
 * Description: Initialize Watch Timer Resource
 */
void init_wt(void)
{
  WTMR = 0;   //default: disabled, no interrupt, fwck/2^11, clock source = fsub
  WTR = 0x7F;   //default: free run
}


/***
 * Function:  WT_Isr
 * Description:  Watch Timer service interrupt
 */
```

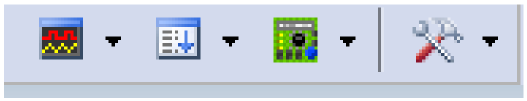
```
Load "C:\\Program Files\\ZiLOG\\Z8051_1.1\\samples\\Z51F0811\\Demo\\Kei
Flash Verify Done: 7402 bytes verified.

>
ASM ASSIGN BreakDisable BreakEnable BreakKill BreakList BreakSet
```

Name	Location/Value	Type
UGETCH	C:0x1DA1	
ch	0x20 ' '	char
MAIN	C:0x125D	

Figure 30. A Default Debug Session

Note: The following buttons in the Keil µVision IDE are not supported by the Zilog OCD driver:



To learn more about the full functionality of the Keil µVision IDE, please refer to the Keil Keil µVision4 IDE documentation.

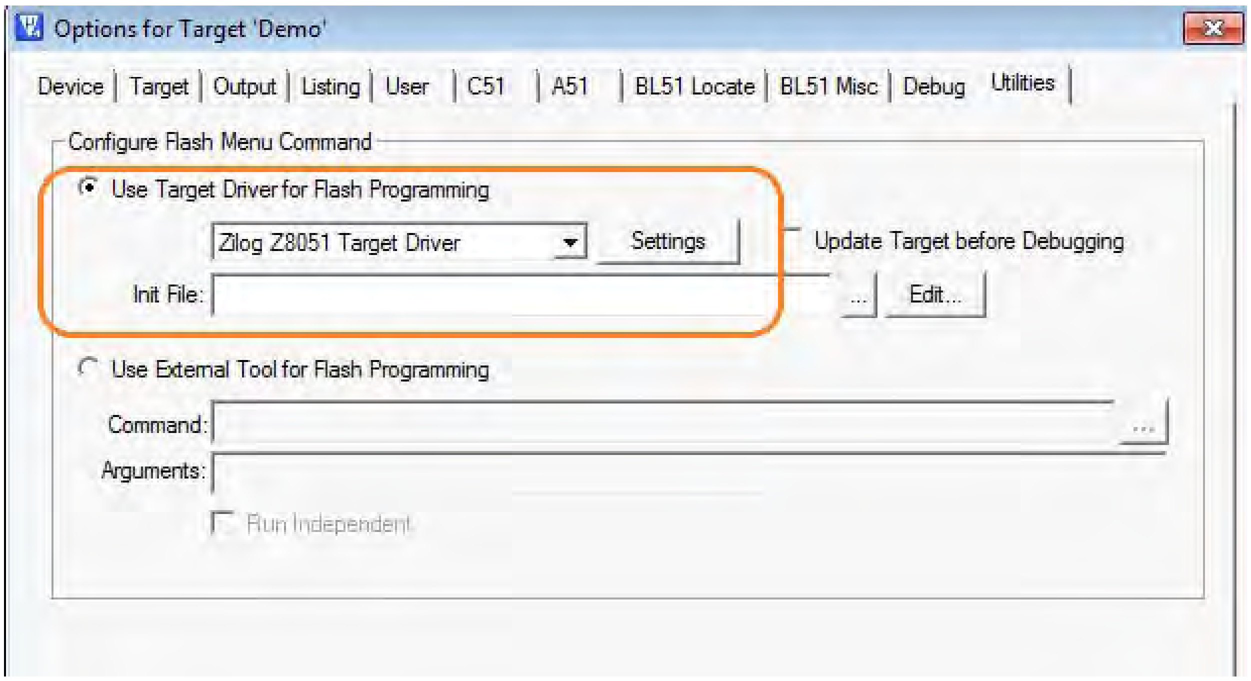
5. From the **Debug** menu, select **Run**, or simply press the **F5** key on your Windows keyboard to run the demo project. As a result, LEDs D1, D2 and D3 on the Z51F0811 Evaluation Board will blink in sequence.
6. To stop code execution, select **Stop** from the **Debug** menu.

7. To stop your debugging session, click the **Start/Stop Debug Session** icon.

Stand-Alone Flash Programming Using the Keil µVision IDE

Observe the following procedure to program Flash memory without debugging.

1. From the **Project** menu, open the **Options for Target 'your project'**, and click the **Utilities** tab. In the **Use Target Driver for Flash Programming** drop-down menu within the Configure Flash Menu Command pane, ensure that **Zilog Z8051 Target Driver** is selected, as indicated in Figure 31.

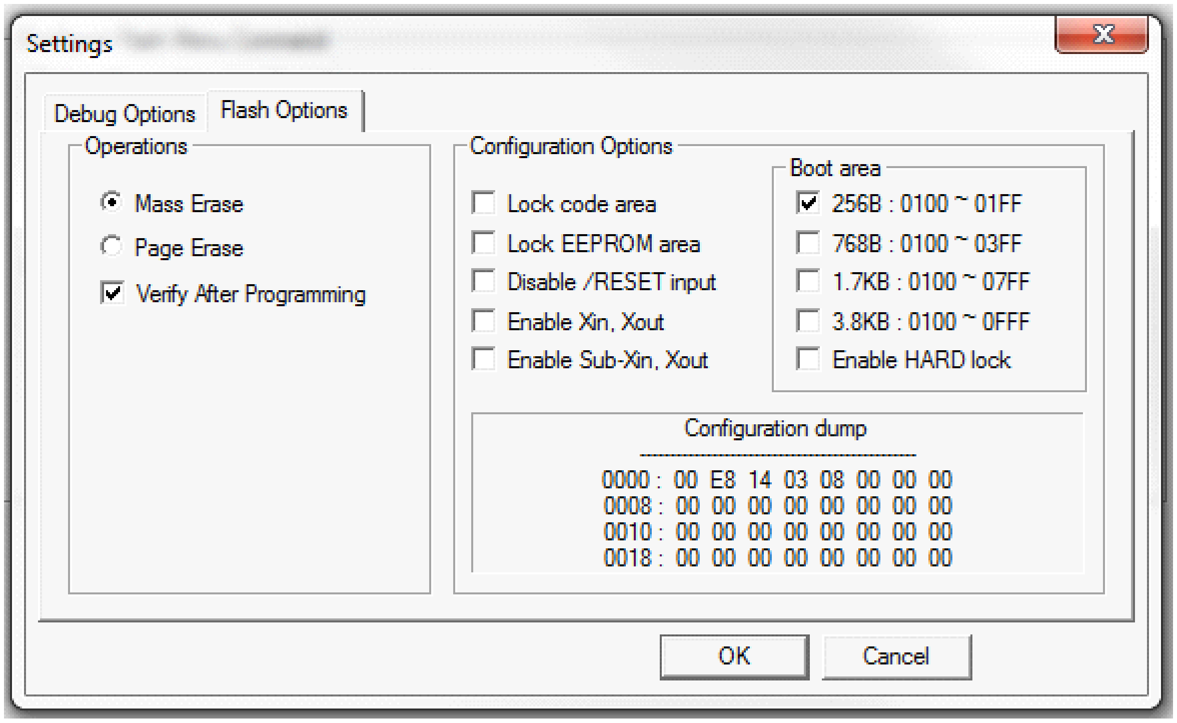

Options for Target 'Demo'
Device
Target
Output
Listing
User
C51
A51
BL51 Locate
BL51 Misc
Debug
Utilities
Configure Flash Menu Command
Settings
Update Target before Debugging
...
Edit...
Use External Tool for Flash Programming
Command:
Arguments:
Run Independent

(●) Use Target Driver for Flash Programming: Zilog Z8051 Target Driver

Init File:

Figure 31. Selecting A Target Driver For Flash Programming

2. Click the **Settings** button to change any additional Flash options. The Settings dialog will appear, as shown in Figure 32.


Settings

Debug Options | Flash Options

Operations
- Mass Erase [✓]
- Page Erase
- [✓] Verify After Programming

Configuration Options
- [ ] Lock code area
- [ ] Lock EEPROM area
- [ ] Disable /RESET input
- [ ] Enable Xin, Xout
- [ ] Enable Sub-Xin, Xout

Boot area
- [✓] 256B : 0100 ~ 01FF
- [ ] 768B : 0100 ~ 03FF
- [ ] 1.7KB : 0100 ~ 07FF
- [ ] 3.8KB : 0100 ~ 0FFF
- [ ] Enable HARD lock

```
Configuration dump
------------------------------------
0000 :  00  E8  14  03  08  00  00  00
0008 :  00  00  00  00  00  00  00  00
0010 :  00  00  00  00  00  00  00  00
0018 :  00  00  00  00  00  00  00  00
```

Figure 32. Configuring Additional Flash Options

3. After you have selected your Flash options, click **OK** to exit the **Flash Options** dialog.
4. Click **OK** to exit the **Options For Target 'Demo'** dialog.
5. From the **Flash** menu of the Keil IDE (see Figure 33), select either of the following options:
   - Select **Download** to program Flash memory with the current project
   - Select **Erase** to perform a mass erase of internal Flash memory

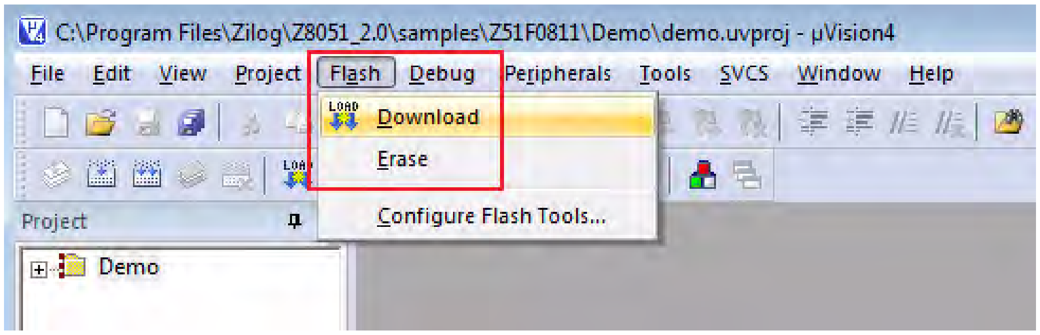


Figure 33. The Keil IDE Flash Menu

For Further Information

Working with the unrestricted, paid version of the Keil tools is very much the same, except that you have the freedom to build significant real-world applications. For information, see the Keil website.

Obviously, there are more details to consider when working with the Keil tools. A large and well-organized body of information about these tools can be accessed through the uVision Help menu. You can also access this information without running uVision by examining the HTML Help files contained in the following path:

```
<Keil install>\C51\Hlp
```

Especially helpful are the `c51` file about the compiler, the `a51` file about the assembler and the `bl51` file that discusses the basic linker.

Appendix A. SDCC Build Batch File Listing

This appendix offers a complete listing of the SDCC build batch file that is discussed in the Using The SDCC Tools section on page 7.

```
Set SDCC_DIR=..\..\..\SDCC_3.1.0
Set SDCC_BIN=%SDCC_DIR%\bin
Set LIB_DIR=..\..\lib\sdcc
Set OUTDIR=.\Sdcc_out
Set CFLAGS= -c --debug --use-stdout --model-large -V -I"../../../
include/C" -I"../common"
Set LFLAGS= --debug --use-stdout --model-large -V


@echo cleaning intermediate files
@call clean.bat

@echo compiling....
@rem uarts.c
%SDCC_BIN%\sdcc.exe %CFLAGS% -o"."/ ../common/uarts.c
@if errorlevel 1 goto Compiling_Error

@rem main.c
%SDCC_BIN%\sdcc.exe %CFLAGS% main.c
@if errorlevel 1 goto Compiling_Error

@echo linking....
%SDCC_BIN%\sdcc.exe %LFLAGS% -oled_blink.hex main.rel uarts.rel
%LIB_DIR%\crtxinit.rel
@if errorlevel 1 goto Linking_Error

@echo.
@echo Build was complete and successful.
@goto end

:Compiling_Error
@Echo.
@Echo !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
@Echo !!!   Compiling Errors occurred. See above  !!
@Echo !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
@goto end

:Linking_Error
@Echo.
@Echo !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
@Echo !!!   Linking Errors occurred. See above    !!
@Echo !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

:end
@Echo.
```

```
@Echo Hit any key to close this console window. > CON:
@pause > NUL:
```

Warning: DO NOT USE THIS PRODUCT IN LIFE SUPPORT SYSTEMS.

LIFE SUPPORT POLICY

As used herein

Document Disclaimer

Customer Support

To share comments, get your technical questions answered, or report issues you may be experiencing with our products, please visit Zilog's Technical Support page at http://support.zilog.com.

To learn more about this product, find additional documentation, or to discover other facets about Zilog product offerings, please visit the Zilog Knowledge Base or consider participating in the Zilog Forum.

This publication is subject to replacement by a later edition. To determine whether a later edition exists, please visit the Zilog website at http://www.zilog.com.