PRECISION32™ OPTIMIZATION CONSIDERATIONS FOR CODE SIZE AND SPEED

1. Introduction

The code size and execution speed of a 32-bit MCU project can vary greatly depending on the way the code is written, the toolchain libraries used, and the compiler and linker options. This document addresses how to determine what portions of code are taking extra space or time and ways to optimize for space or speed for different tool chains including GCC redlib and newlib (Precision32™ IDE), Keil, and IAR.

2. Key Points

The key topics of this document are as follows:

- How to determine what portions of the project are taking the most space
- Ways to benchmark code execution speed
- Common strategies to reduce code size or improve execution speed
- Code startup time and ways to reduce it

3. Using CoreMark™ as a Speed Benchmark

CoreMark is a standard code base that can be ported to various processors to provide a speed benchmark. The CoreMark software provides a score that rates how fast the core and code is, providing a relative comparison between various toolchain options and settings. The CoreMark software package cannot be modified except for device-specific information in the **portme** files. For modes that do not support printf (nohosting libraries), the results were calculated using the value of the variable in code. See the CoreMark website for more information on the test and score reporting requirements (www.coremark.org).

4. Non-Toolchain Considerations

The coding style and technique can have a great effect on the overall size of the project.

4.1. Coding Techniques

There are many ways coding technique can affect code size, including library calls, inline code or data, or code optimizations made for global variables or pointers.

For more information on writing C code for ARM architectures, see the following resources:

- **EETimes—Energy efficient C code for ARM devices** by Chris Shore: http://www.eetimes.com/design/embedded/4210470/Efficient-C-Code-for-ARM-Devices
- **Compiler Coding Practices—ARM**: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0472c/CJAFJCFG.html

These guidelines will largely apply regardless of the compiler used for the project.

4.2. Number of Function Parameters

Functions with Keil, GCC, and IAR can have as many parameters as desired. In general, the first four parameters are passed to the function efficiently using registers. Any additional parameters beyond four must be moved on or off the stack, which results in extra code size for each additional parameter and extra time to execute those instructions. If possible, keeping functions to no more than four parameters can help reduce code size and execution time.

4.3. Alignment

In most cases, Cortex-M3 linkers place code in memory efficiently. In some projects, however, the alignment of functions and code can be carefully managed manually to reduce code size or change code execution speed. For example, if two functions in the same file call each other, but one ends up in flash and one ends up in RAM, the compiler may need to place extra code to perform a long jump and take longer to execute that jump. If needed, functions and variables can be explicitly located using scatterfiles and linker flags. More information on linker scripts and scatterfiles can be found on the toolchain websites:

- **Code Red**: http://support.code-red-tech.com/CodeRedWiki/OwnLinkScripts
- **ARM**: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.kui0101a/armlink_babddhbf.htm
- **IAR**: http://supp.iar.com/Support/?note=14438

4.4. RAM Size

The RAM size of a project can be just as important as the code size. In particular, the default configurations for SiM3xxxx projects place the stack at the top of memory growing down and the heap at the end of program data growing up. If too much of the RAM is used by program data, then the stack and heap may collide, leading to difficult debugging issues in run-time. Projects should always leave enough RAM space to accommodate the function-calling depth of the code.

4.5. SiM3xxxx Core and Flash Access Speed

At the maximum device AHB speed, an SiM3xxxx device reading flash every pipeline cycle may violate the maximum flash access speed. To compensate for this, the FLASHCTRL module has controls to reduce the flash access speed (SPMD and RDSEN). Depending on the code density and make-up (i.e., 16-bit or 32-bit instructions), this may lead to stalls in the core before the next instructions can be fetched from flash. Executing at high speeds with strings of 16-bit instructions may yield the fastest core operation.

4.6. SiM3xxxx Core and the Direct Memory Access (DMA) Module

On SiM3xxxx devices, the core and the DMA can access multiple AHB slaves at the same time without any performance degradation. If the core and DMA access the same AHB slave at the same time (i.e., RAM), then the AHB has priority-based arbitration in the following precedence:

1. Core data fetch
2. DMA
3. Core instruction fetch

If multiple DMA channels are active at the same time and accessing the same memory areas as the core, this could lead to a reduction in core execution speed.

5. Precision32 IDE (redlib and newlib)

This section discusses ways to optimize projects using the Precision32 IDE and both redlib and newlib libraries. The Precision32 GCC tools used for the code size and execution speed testing discussed in this document are ARM/embedded-4_6-branch revision 182083 (http://gcc.gnu.org/svn/gcc/branches/ARM/embedded-4_6-branch/) with newlib v1.19 and Redlib v2 (Precision32 IDE v4.2.1 [Build 73]).

5.1. Reading the Map File

The first step in the code size optimization process is to analyze the project map file and determine what portions of code take the most space.

The map file is an output of the linker that shows the size of each function and variable and their positions in memory. This map file is located in the build files for a project.

In addition to the functions, the map file includes information on variables and other symbols, including unused functions that are removed.

For a Precision32 IDE Debug build, the map file is located in the project's **Debug** directory. Figure 1 shows an excerpt of the sim3u1xx_Blinky redlib Debug example map file.

For each function in the project, the map file lists the starting address and the length. For example, the **my_rtc_alarm0_handler** function starts at address 0x0000_04D4 and occupies 0x70 bytes of memory.

```
                0x000002ac                DMACH6_IRQHandler
                0x000002ac                VDDLOW_IRQHandler
*(.text*)
.text.SystemInit
                0x000002b4        0xa8 ./src/sim3u1xx/system_sim3u1xx.o
                0x000002b4                SystemInit
.text.main      0x0000035c       0x120 ./src/main.o
                0x0000035c                main
.text.mySystemInit
                0x0000047c        0x50 ./src/myCpu.o
                0x0000047c                mySystemInit
.text.my_rtc_fail_handler
                0x000004cc         0x8 ./src/myRtc0.o
                0x000004cc                my_rtc_fail_handler
.text.my_rtc_alarm0_handler
                0x000004d4        0x70 ./src/myRtc0.o
                0x000004d4                my_rtc_alarm0_handler
.text.NVIC_SetPriority
                0x00000544        0x5c ./src/generated/gCpu.o
```

Figure 1. sim3u1xx_Blinky Precision32 Debug Map File Example

5.2. Determining a Project's Code Size

Each project's library and function usage is different. Analyzing the project's makeup can help determine the most effective way to reduce code space.

All Precision32 SDK projects automatically output the code and RAM size after a build. To modify this output in the Precision32 IDE:

1. Right-click on the **project_name** in the **Project Explorer** view.
2. Select **Properties**.
3. In the **C/C++ Build→Settings→Build Steps** tab, remove or add the following in the **Post-build steps→Command** box: **arm-none-eabi-size "${BuildArtifactFileName}"**

After building the si32HAL 1.0.1 **sim3u1xx_Blinky** example, the IDE outputs:

```
   text    data     bss     dec     hex

  13312       4     344   13660    355c
```

The areas of memory are as follows:

- **text**: code and read-only memory in decimal
- **data**: read-write data in decimal
- **bss**: zero-initialized data in decimal
- **dec**: total of text, data, and bss in decimal
- **hex**: total of text, data, and bss in hex

More information about the size tool can be found on the Code Red website (http://support.code-red-tech.com/CodeRedWiki/FlashRamSize).

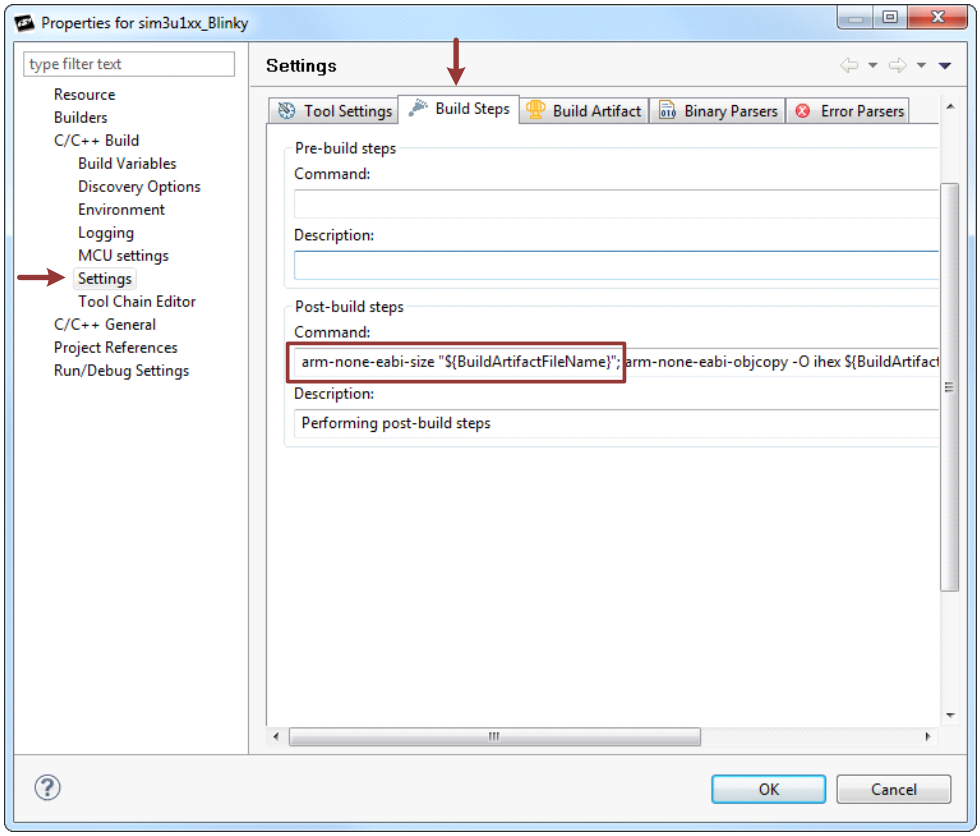


Figure 2. Automatically Reporting Project Size on Project Build in Precision32

5.3. Toolchain Library Usage

Some toolchains have multiple libraries or settings that can change the size or execution speed of code. The Precision32 tools have six options:

- newlib (standard GCC) with no standard I/O
- newlib (standard GCC) with nohosting standard I/O
- newlib (standard GCC) with semihosting standard I/O
- redlib (GCC) with no standard I/O
- redlib (GCC) with nohosting standard I/O
- redlib (GCC) with semihosting standard I/O

The **semihosting** libraries have additional hooks to enable a project to send debugging information to an IDE running on a PC. The **nohosting** libraries have this additional capability removed. The **none** versions of the toolchains have no standard I/O capability (i.e., no printf()).

For some example projects (like si32HAL 1.0.1 **sim3u1xx_Blinky**), the compile-time library can be modified by opening the **myLinkerOptions_p32.ld** file in the project directory and changing the uncommented line.

```
/******************** C LIBRARY SUPPORT ***************/
/*GROUP(libgcc.a libc.a libm.a libcr_newlib_nohost.a)*/
/*GROUP(libgca.a libc.a libm.a libcr_newlib_semihost.a)*/
GROUP(libcr_semihost.a libcr_c.a libcr_eabihelpers.a)
/*GROUP(libcr_nohost.a libcr_c.a libcr_eabihelpers.a)*/
```

Figure 3. Using the myLinkerOptions_p32.ld File to Select the Project Library

The four lines in the file correspond to a library:

- **GROUP(libgcc.a libc.a libm.a libcr_newlib_nohost.a) (line 4)**: newlib nohosting
- **GROUP(libgca.a libc.a libm.a libcr_newlib_semihost.a) (line 5)**: newlib semihosting
- **GROUP(libcr_semihost.a libcr_c.a libcr_eabihelpers.a) (line 6)**: redlib semihosting
- **GROUP(libcr_nohost.a libcr_c.a libcr_eabihelpers.a) (line 7)**: redlib nohosting

The none libraries do not have corresponding entries in this file. Add these lines to add support for none:

- **GROUP(libgcc.a libc.a libm.a)**: newlib none
- **GROUP(libcr_c.a libcr_eabihelpers.a)**: redlib none

After setting the myLinkerOptions_P32.ld file to the correct setting, set the IDE to the same library using these steps:

1. Left-click on the **project_name** in the **Project Explorer** view.
2. Select **Properties**.
3. Click on **C/C++ Build→Settings→Tool Settings** tab→**MCU Linker→Target** and select the desired library from the **Use C library** drop-down menu. Figure 4 shows this dialog in the Precision32 IDE.
4. Clean and Build the project.

AppBuilder projects and si32HAL examples v1.1.1 or later do not have a **myLinkerOptions_P32.ld** file and can use the **Quickstart** view setting only.

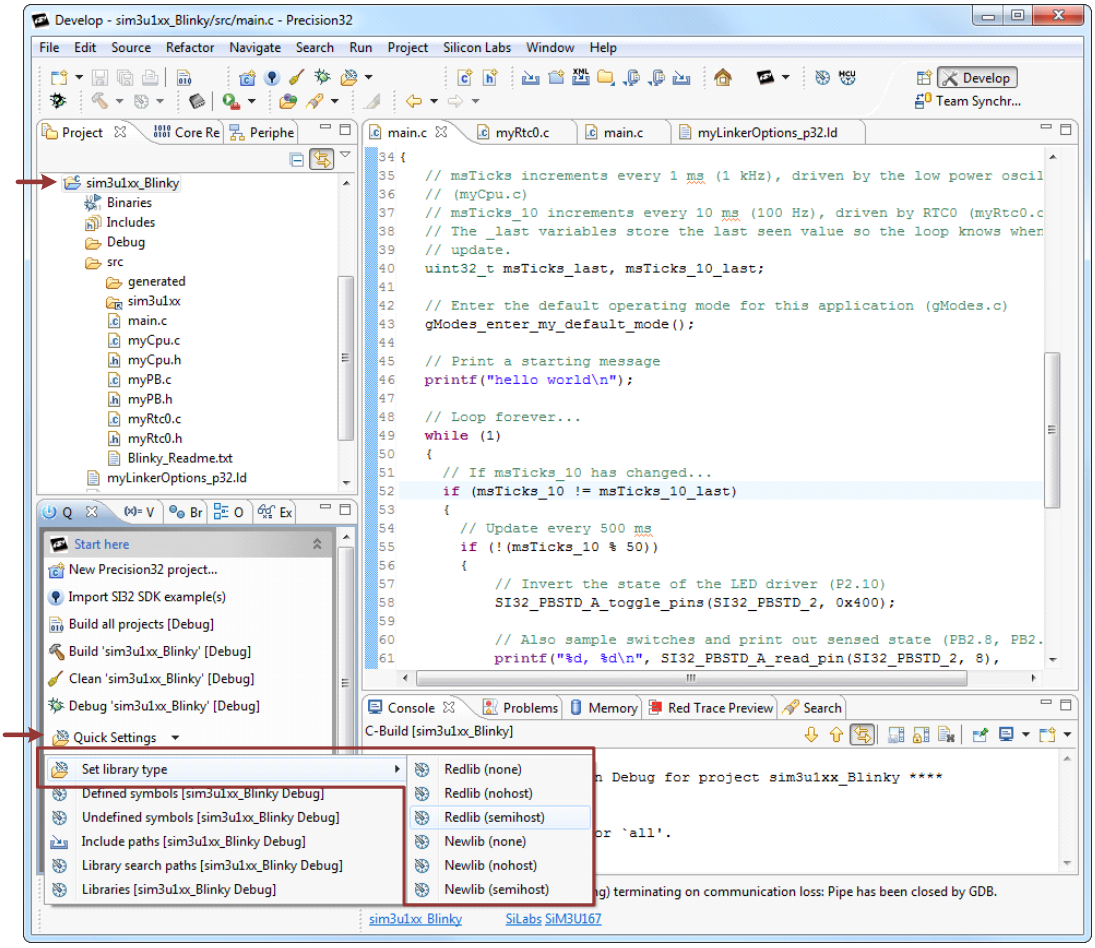


Figure 4. Using the Precision32 IDE to Select the Project Library

Using the **sim3u1xx_Blinky** and **demo_si32UsbAudio** default examples in the si32HAL 1.0.1 software package, Table 1 and Table 2 show the relative Debug build sizes with the different toolchain library options. Table 3 shows the Debug build sizes for CoreMark, and Table 4 shows the relative CoreMark speed scores for each of these library options.

For the newlib and redlib **none** libraries, see “5.4. Function Library Usage”.

Table 1. Precision32 Toolchain Library Usage Comparison—sim3u1xx_Blinky Debug

Library	Code (bytes)	Read Only Data (bytes)	Read-Write Data (bytes)	Zero-Initialized Data (bytes)
newlib semihosting	35564		2248	124
newlib nohosting	34864		2248	68
newlib none	N/A (requires printf() removal)			
redlib semihosting	13080		4	344
redlib nohosting	13136		4	344
redlib none	N/A (requires printf() removal)			

Table 2. Precision32 Toolchain Library Usage Comparison—demo_si32UsbAudio Debug

Library	Code (bytes)	Read Only Data (bytes)	Read-Write Data (bytes)	Zero-Initialized Data (bytes)
newlib semihosting	108844		6944	11904
newlib nohosting	108144		6944	11848
newlib none	N/A (requires printf() removal)			
redlib semihosting	76176		4704	12124
redlib nohosting	76120		4704	12124
redlib none	N/A (requires printf() removal)			

Table 3. Precision32 Toolchain Library Usage Comparison—CoreMark Debug Size

Library	Code (bytes)	Read Only Data (bytes)	Read-Write Data (bytes)	Zero-Initialized Data (bytes)
newlib semihosting	46900		2352	2140
newlib nohosting	46208		2352	2084
newlib none	N/A (requires printf() removal)			
redlib semihosting	24400		112	2360
redlib nohosting	24344		112	2360
redlib none	N/A (requires printf() removal)			

Table 4. Precision32 Toolchain Library Usage Comparison—CoreMark Debug Speed

Library	CoreMark Score
newlib semihosting	CoreMark 1.0: **37.571643** / GCC4.6.2 20110921 (release) [ARM/embedded-4_6-branch revision 182083] Iterations=3000 / STACK
newlib nohosting	CoreMark 1.0: **37.571643** / GCC4.6.2 20110921 (release) [ARM/embedded-4_6-branch revision 182083] Iterations=3000 / STACK
newlib none	N/A (requires printf() removal)
redlib semihosting	CoreMark 1.0: **37.571643** / GCC4.6.2 20110921 (release) [ARM/embedded-4_6-branch revision 182083] Iterations=3000 / STACK
redlib nohosting	CoreMark 1.0: **37.571643** / GCC4.6.2 20110921 (release) [ARM/embedded-4_6-branch revision 182083] Iterations=3000 / STACK
redlib none	N/A (requires printf() removal)

5.4. Function Library Usage

Function libraries such as floating point math and **printf()** can significantly increase the size of a project. If a project is constrained by size, a careful analysis of the usage of these large libraries may be required. For example, floating point can often be approximated well by fixed point math, eliminating the need for the floating point libraries.

The **printf()** library is often needed by projects for debugging or release code. If **printf()** is used for debugging purposes, using a defined symbol in the project to remove **printf()** when compiling a release build can dramatically reduce the size of a project. To define a symbol to differentiate between a Debug project and a Release project, see “5.7. Reset Sequence”. The code can then use **#ifdef**...**#endif** preprocessor statements to remove debugging code or **printf()** calls.

The removal of debugging **printf()** statements can dramatically reduce the code size of a project. A simple way to do this is to redefine the printf function at the top of the file containing the **printf()** calls using the following statement:

```
#define printf(args...)
```

For si32Library examples such as **demo_si32UsbAudio**, define the statement at the top of myBuildOptions.h to remove all calls to printf() with higher optimization settings. Additionally, reduce the code size footprint by disabling logging in myBuildOptions.h:

```
#define si32BuildOption_enable_logging 0
```

This method preserves the **printf()** statements for later use, if needed. The **printf()** define can also be encapsulated with preprocessor **#if** statements to automatically include this define when building with a Release configuration.

When removing **printf()** for use with newlib none or redlib none, all references to **printf()** and **stdio.h** must be commented out of the project. The **none** libraries cannot be used with si32Library projects.

To verify that all instances of **printf()** have been removed, search the map file for the project for the printf library. In the **sim3u1xx_Blinky** example, this means adding the statement to both the main.c and gCpu.c files.

Instead of using standard **printf()**, which can have a high library cost, use integer-only print functions like **iprintf()** for newlib projects. For redlib projects in the Precision32 IDE, create a define **CR_INTEGER_PRINTF** in the project properties to force an integer-only version of **printf()**. For instances of **printf()** with a fixed-string, using **puts()** can dramatically reduce code size.

More information about redlib and **printf()** can be found on the Code Red website: http://support.code-red-tech.com/CodeRedWiki/UsingPrintf.

If a project does not use any standard I/O functions, use the redlib or newlib **none** toolchain option to reduce code size as discussed in “6.3. Toolchain Library Usage”.

Using the **sim3u1xx_Blinky** default example in the si32HAL 1.0.1 software package, Table 5 shows the relative build sizes with the different **printf()** settings. The **demo_si32UsbAudio** comparison is not included since **printf()** removal requires higher optimization settings or code modifications. This section also does not include the CoreMark tests since **printf()** is not part of the CoreMark benchmark.

Table 5. Precision32 printf() Comparison—sim3u1xx_Blinky Debug

Library	Code (bytes)	Read Only Data (bytes)	Read-Write Data (bytes)	Zero-Initialized Data (bytes)
newlib semihosting with printf	35564		2248	124
newlib nohosting with printf	34864		2248	68
newlib nohosting with integer printf (iprintf)	19800		2248	68
newlib nohosting with puts instead of printf	8784		2120	68
newlib nohosting without printf	2064		4	8
newlib none with all calls to stdio and printf removed	2064		4	8
redlib semihosting with printf	12880		4	344
redlib nohosting with printf	12824		4	344
redlib nohosting with integer printf (**CR_INTEGER_PRINTF)**	8111		4	344
redlib nohosting with puts instead of printf	4004		4	344
redlib nohosting without printf	3868		4	344
redlib none with all calls to stdio and printf removed	2068		4	8

5.5. Toolchain Optimization Settings

In addition to the library types, each toolchain has multiple optimization settings that can affect the resulting code size. With the Precision32 toolchain, code optimization can be set by following these steps:

1. Right-click on the **project_name** in the **Project Explorer** view.
2. Select **Properties**.
3. In the **C/C++ Build→Settings→Tool Settings** tab→**MCU C Compiler→Optimization** options, select the desired optimization level.

Figure 5 shows the optimization settings for the Precision32 IDE. Level **-O0** has the least optimization, while **-O3** has the most optimization. An additional flag (**-Os**) allows for specific optimization for code size.

More information on the optimization levels can be found on the Code Red website (http://support.code-red-tech.com/CodeRedWiki/CompilerOptimization) and the GCC website (http://gcc.gnu.org/onlinedocs/gcc-4.0.4/gcc/Optimize-Options.html). Declaring a variable as **volatile** will prevent the compiler from optimizing out the variable.

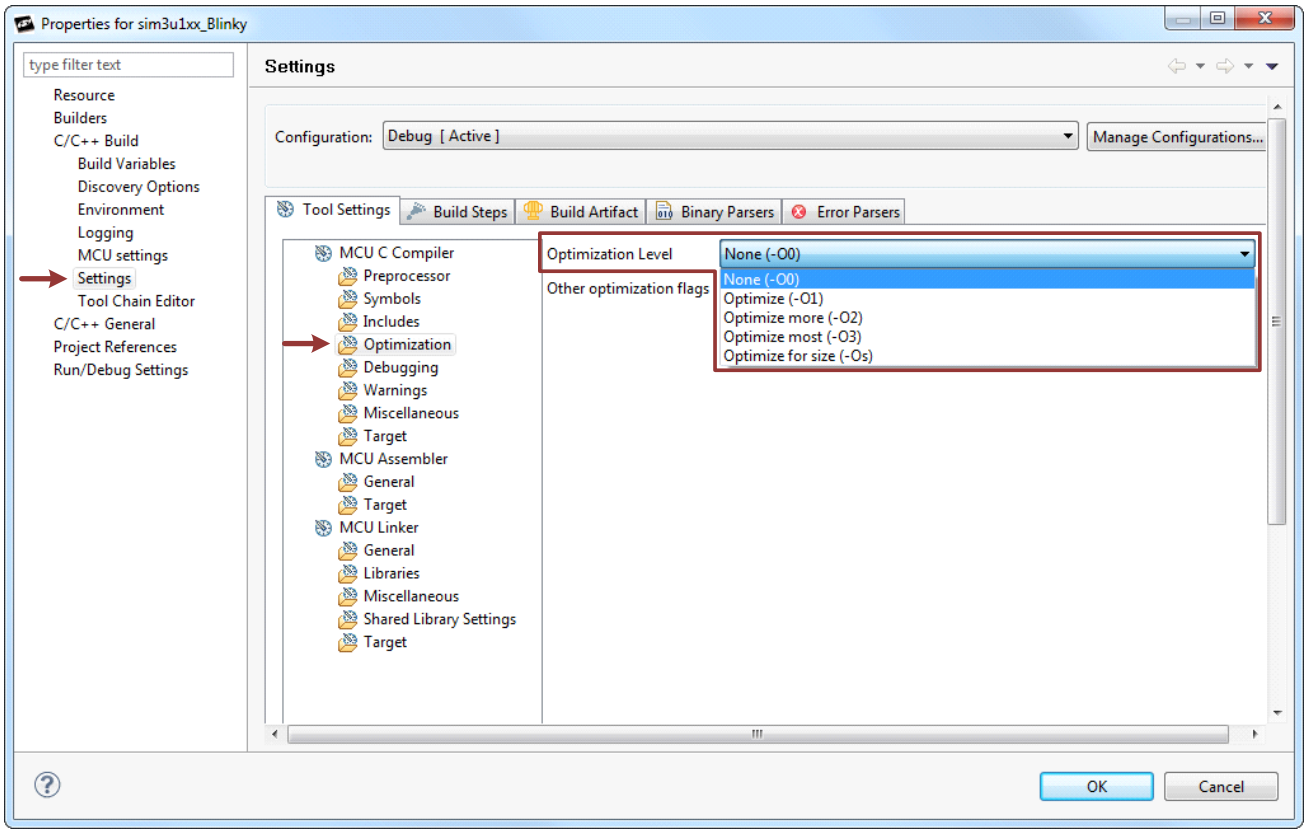


Figure 5. Setting the Project Optimization in the Precision32 IDE

The Precision32 IDE has two build configurations by default: Debug and Release. These build configurations have predefined optimization levels (**None** for Debug, **-O2** for Release). To switch between the two configurations:

1. Right click on the **project_name** in the **Project Explorer** view.
2. Select **Build Configurations→Set Active** and select between **Debug** and **Release**.

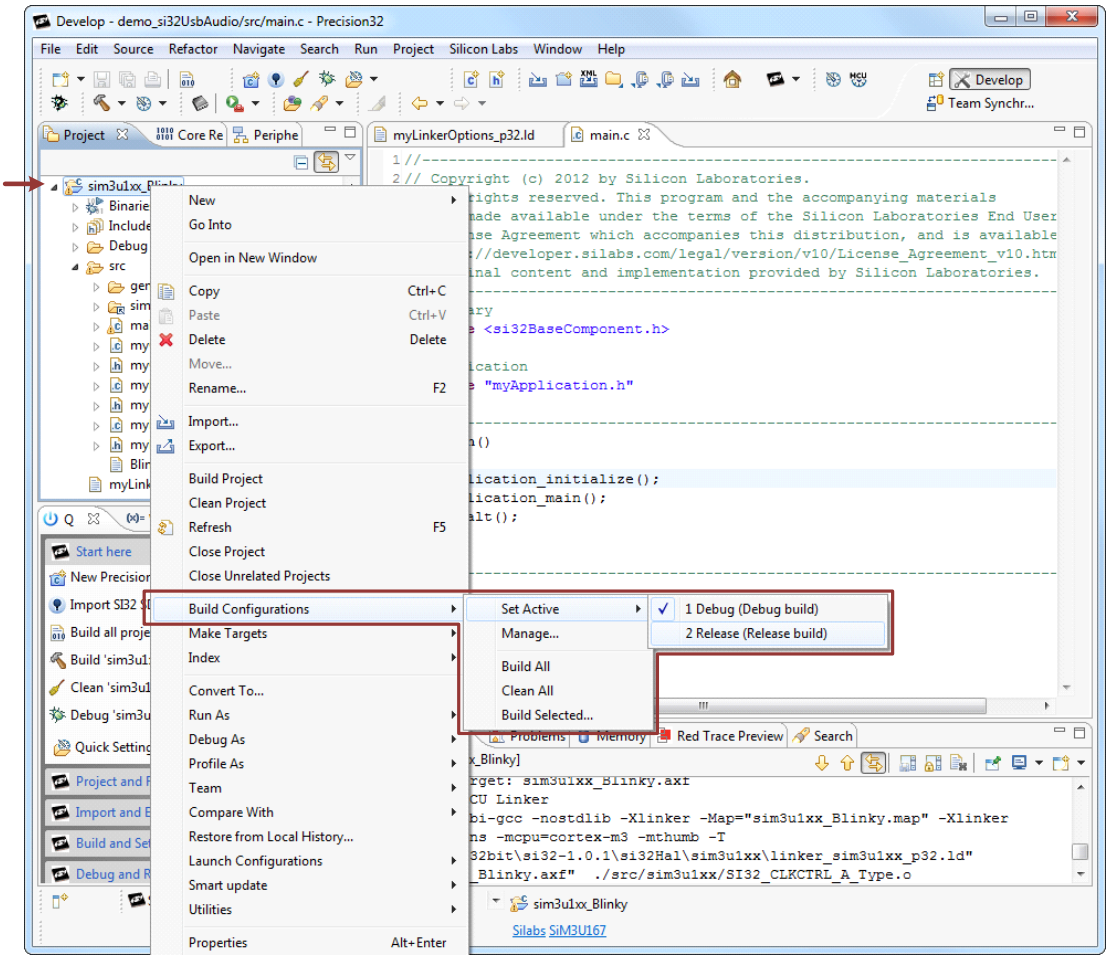


Figure 6. Selecting the Active Build Configuration in the Precision32 IDE

To change the settings of any build configuration:

1. Right click on the **project_name** in the **Project Explorer** view.
2. Select **Properties**.
3. In the **C/C++ Build→Settings→Tool Settings** tab options, select the build configuration at the top and the desired build configuration options.

Using the **sim3u1xx_Blinky** and **demo_si32UsbAudio** default examples in the si32HAL 1.0.1 software package, Table 6 and Table 7 show the relative Debug build sizes with the different optimization level settings. Table 8 shows the CoreMark Debug build sizes, and Table 9 lists the CoreMark speed scores for these optimization levels.

Table 6. Precision32 Toolchain Optimization Comparison—sim3u1xx_Blinky Debug

Library	Code (bytes)	Read Only Data (bytes)	Read-Write Data (bytes)	Zero-Initialized Data (bytes)
newlib nohosting -O0	34864		2248	68
newlib nohosting -O1	34032		2248	68
newlib nohosting -O2	33960		2248	68
newlib nohosting -O3	33960		2248	68
newlib nohosting -Os	33808		2248	68
redlib nohosting -O0	13080		4	344
redlib nohosting -O1	12056		4	344
redlib nohosting -O2	12096		4	344
redlib nohosting -O3	12096		4	344
redlib nohosting -Os	11768		4	344

Table 7. Precision32 Toolchain Optimization Comparison—demo_si32UsbAudio Debug

Library	Code (bytes)	Read Only Data (bytes)	Read-Write Data (bytes)	Zero-Initialized Data (bytes)
newlib nohosting -O0	108144		6944	11848
newlib nohosting -O1	84400		6944	11852
newlib nohosting -O2	83152		6944	11852
newlib nohosting -O3	85136		6944	11856
newlib nohosting -Os	76528		6928	11848
redlib nohosting -O0	76120		4704	12124
redlib nohosting -O1	52048		4700	12124
redlib nohosting -O2	50752		4700	12124
redlib nohosting -O3	52736		4700	12128
redlib nohosting -Os	44128		4688	12120

Table 8. Precision32 Toolchain Optimization Comparison—CoreMark Debug Size

Library	Code (bytes)	Read Only Data (bytes)	Read-Write Data (bytes)	Zero-Initialized Data (bytes)
newlib semihosting -O0	46900		2352	2140
newlib semihosting -O1	41812		2256	2140
newlib semihosting -O2	42828		2256	2140
newlib semihosting -O3	45948		2256	2140
newlib semihosting -Os	40284		2256	2140
redlib nohosting -O0	24344		112	2360
redlib nohosting -O1	19160		12	2360
redlib nohosting -O2	20176		12	2360
redlib nohosting -O3	23296		12	2360
redlib nohosting -Os	17624		12	2360

Table 9. Precision32 Toolchain Optimization Comparison—CoreMark Debug Speed

Library	CoreMark Score
newlib semihosting -O0	CoreMark 1.0: **36.478654** / GCC4.6.2 20110921 (release) [ARM/embedded-4_6-branch revision 182083] Iterations=3000 / STACK
newlib semihosting -O1	CoreMark 1.0: **79.807436** / GCC4.6.2 20110921 (release) [ARM/embedded-4_6-branch revision 182083] Iterations=3000 / STACK
newlib semihosting -O2	CoreMark 1.0: **107.984518** / GCC4.6.2 20110921 (release) [ARM/embedded-4_6-branch revision 182083] Iterations=3000 / STACK
newlib semihosting -O3	CoreMark 1.0: **103.509985** / GCC4.6.2 20110921 (release) [ARM/embedded-4_6-branch revision 182083] Iterations=3000 / STACK
newlib semihosting -Os	CoreMark 1.0: **87.64509** / GCC4.6.2 20110921 (release) [ARM/embedded-4_6-branch revision 182083] Iterations=3000 / STACK
redlib nohosting -O0	CoreMark 1.0: **37.571643** / GCC4.6.2 20110921 (release) [ARM/embedded-4_6-branch revision 182083] Iterations=3000 / STACK
redlib nohosting -O1	CoreMark 1.0: **79.998784** / GCC4.6.2 20110921 (release) [ARM/embedded-4_6-branch revision 182083] Iterations=3000 / STACK
redlib nohosting -O2	CoreMark 1.0: **107.984518** / GCC4.6.2 20110921 (release) [ARM/embedded-4_6-branch revision 182083] Iterations=3000 / STACK
redlib nohosting -O3	CoreMark 1.0: **103.509985** / GCC4.6.2 20110921 (release) [ARM/embedded-4_6-branch revision 182083] Iterations=3000 / STACK
redlib nohosting -Os	CoreMark 1.0: **87.64509** / GCC4.6.2 20110921 (release) [ARM/embedded-4_6-branch revision 182083] Iterations=3000 / STACK

5.6. Unused Code Removal

Each file in a project becomes an object that is included. In other words, if any functions in a file are used, then the entire file is included by default. This can become an issue for a project using the si32HAL and only a few functions from each module.

Removed (unused) functions can be viewed in the map files for the projects.

For Precision32, the **-ffunction-sections** and **-fdata-sections** optimization flags place each function and data item into separate sections in the file before linking them into the project. This means the compiler can optimize out any unused functions. These flags are present in Example and AppBuilder projects by default and should be configured on a file-by-file basis. To add or remove these options to a file:

1. Right-click on the **file_name** in the **Project Explorer** view.
2. Select **Properties**.
3. In the **C/C++ Build→Settings→Tool Settings** tab→**MCU C Compiler→Miscellaneous** options, add or remove the **-ffunction-sections** and **-fdata-sections** flags after the **-fno-builtin** flag to the **Other flags** text box.

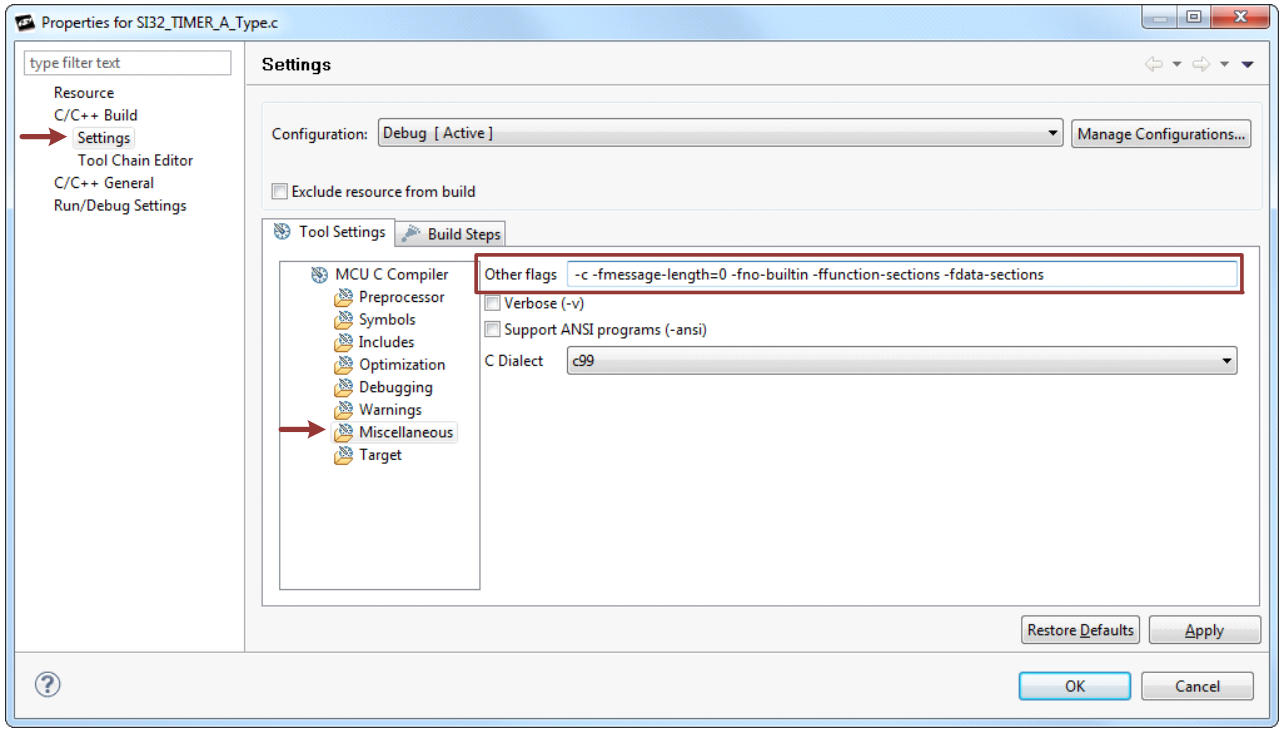


Figure 7. Modifying the Remove Unused Code Compiler Flags in the Precision32 IDE

These flags must be compiled with the **--gc-sections** linker command, which is enabled by default in the Precision32 IDE. It is recommended that this linker command always remain enabled. These flags only have a benefit in some cases, and may cause larger code size and slower execution in some cases.

Using the **sim3u1xx_Blinky** and **demo_si32UsbAudio** default examples in the si32HAL 1.0.1 software package, Table 10 and Table 11 show the relative Debug build sizes with different unused code removal settings. For no unused code removal, the projects were compiled without **-ffunction-sections** and**-fdata-sections** and with **--gc-sections**. For the examples with unused code removal, the projects were compiled with **-ffunction-sections**, **-fdata-sections**, and **--gc-sections**. Table 12 shows the CoreMark build sizes, and Table 13 shows the CoreMark scores for the different unused code removal settings.

Table 10. Precision32 Unused Code Removal Comparison—sim3u1xx_Blinky Debug

Library	Code (bytes)	Read Only Data (bytes)	Read-Write Data (bytes)	Zero-Initialized Data (bytes)
newlib nohosting with no unused code removal	35504		2248	68
newlib nohosting with unused code removal	35112		2248	68
redlib nohosting with no unused code removal	13472		4	344
redlib nohosting with unused code removal	13080		4	344

Table 11. Precision32 Unused Code Removal Comparison—demo_si32UsbAudio Debug

Library	Code (bytes)	Read Only Data (bytes)	Read-Write Data (bytes)	Zero-Initialized Data (bytes)
newlib nohosting with no unused code removal	122424		7240	12116
newlib nohosting with unused code removal	108144		6944	11848
redlib nohosting with no unused code removal	90288		5000	12392
redlib nohosting with unused code removal	76120		4704	12124

Table 12. Precision32 Unused Code Removal Comparison—CoreMark Debug Size

Library	Code (bytes)	Read Only Data (bytes)	Read-Write Data (bytes)	Zero-Initialized Data (bytes)
newlib semihosting with no unused code removal	47188		2368	2140
newlib semihosting with unused code removal	46900		2352	2140
redlib nohosting with no unused code removal	24656		124	2360
redlib nohosting with unused code removal	24344		112	2360

Table 13. Precision32 Unused Code Removal Comparison—CoreMark Debug Speed

Library	CoreMark Score
newlib semihosting with no unused code removal	CoreMark 1.0: **37.452232** / GCC4.6.2 20110921 (release) [ARM/embedded-4_6-branch revision 182083] Iterations=3000 / STACK
newlib semihosting with unused code removal	CoreMark 1.0: **37.571643** / GCC4.6.2 20110921 (release) [ARM/embedded-4_6-branch revision 182083] Iterations=3000 / STACK
redlib nohosting with no unused code removal	CoreMark 1.0: **37.875848** / GCC4.6.2 20110921 (release) [ARM/embedded-4_6-branch revision 182083] Iterations=3000 / STACK
redlib nohosting with unused code removal	CoreMark 1.0: **37.571643** / GCC4.6.2 20110921 (release) [ARM/embedded-4_6-branch revision 182083] Iterations=3000 / STACK

5.7. Reset Sequence

The speed of the reset sequence of a device can be an important factor, especially for devices like the SiM3U1xx/SiM3C1xx that require a reset to exit the lowest power mode.

After the hardware jumps to the reset vector and loads the stack pointer address, the core must initialize the memory of the device. This involves copying data from flash to RAM and zero-filling any zero-initialized segments. Then, the reset code typically calls a system initialization function and jumps to main.

This reset sequence may take different times based on the library used with the project. The startup code should always be compiled with the fastest speed optimization to ensure it takes as little time as possible.

The si32HAL examples have a ~500 ms delay added to a pin reset event to prevent code from switching to a non-existent clock source and disable the device. This delay can be removed by defining the **si32HalOption_disable_pin_reset_delay** symbol in the project.

To define a symbol in the Precision32 IDE:

1. Right-click on the **project_name** in the **Project Explorer** view.
2. Select **Properties**.
3. In the **C/C++ Build→Settings→Tool Settings** tab→**MCU C Compiler→Settings** options, add or remove the symbol to the **Defined symbols (-D)** area.

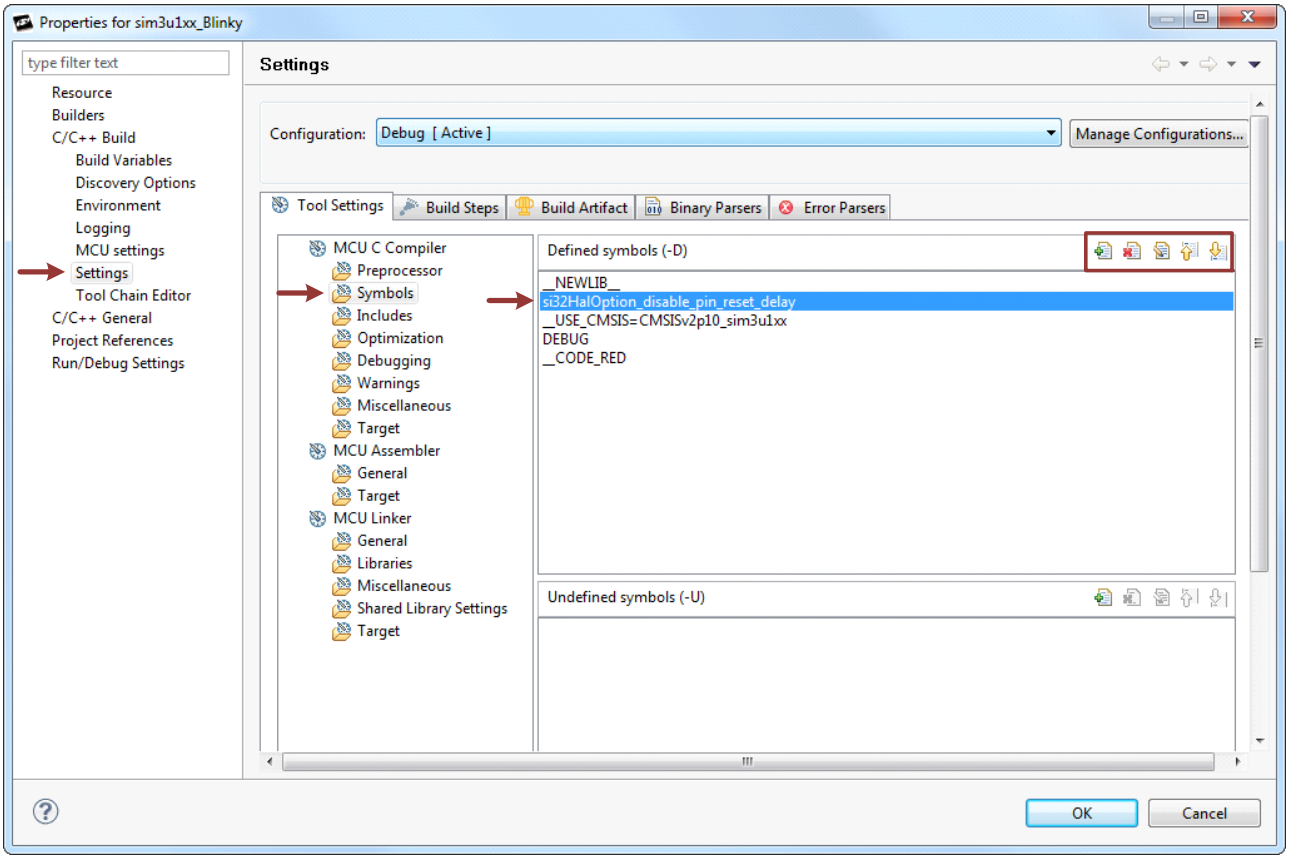


Figure 8. Adding a Project Define Symbol in the Precision32 IDE

Table 14 shows the reset time comparison for the toolchain libraries using the fastest speed optimization on the start up code. This time was measured using the **sim3u1xx_Blinky** example in Debug mode from the fall of a port pin at the beginning of the Reset IRQ handler to the fall of a port pin at the beginning of main() on an oscilloscope. This test requires modification of the si32HAL startup sequence file **startup_<device>_p32.c**.

Table 14. Precision32 Toolchain Library Usage Comparison—sim3u1xx_Blinky Debug Reset Sequence

Library	Reset Time (µs)
newlib semihosting with printf()	242
newlib nohosting with printf()	236
newlib none with printf() removed	9.4
redlib semihosting with printf()	90
redlib nohosting with printf()	90
redlib none with printf() removed	9.4

6. ARM/Keil µVision

This section discusses ways to optimize projects using the Keil or ARM toolchain in the µVision IDE. The Keil µVision tools used for the code size and execution speed testing discussed in this document are version v4.1.0.894.

6.1. Reading the Map File

The map file is an output of the linker that shows the size of each function and variable and their positions in memory. This map file is located in the build files for a project. In addition to the functions, the map file includes information on variables and other symbols, including unused functions that are removed.

Figure 9 shows an excerpt from the sim3u1xx_Blinky map file from the Keil toolchain. The functions are listed with a base address and size. In this case, the **my_rtc_alarm0_handler** is 50 bytes located at address 0x0000_03A5.

```
VBUSINVALID_IRQHandler                  0x000002a7   Thumb Code     2  startup_sim3u1xx_arm.o(.text)
VREG0LOW_IRQHandler                     0x000002a9   Thumb Code     2  startup_sim3u1xx_arm.o(.text)
main                                    0x000002b5   Thumb Code   136  main.o(.text)
mySystemInit                            0x00000369   Thumb Code    38  mycpu.o(.text)
my_rtc_fail_handler                     0x000003a1   Thumb Code     4  myrtc0.o(.text)
my_rtc_alarm0_handler                   0x000003a5   Thumb Code    50  myrtc0.o(.text)
SysTick_Handler                         0x00000401   Thumb Code    12  gcpu.o(.text)
gCpu_enter_default_config               0x0000040d   Thumb Code    86  gcpu.o(.text)
gModes_enter_my_default_mode            0x00000499   Thumb Code    16  gmodes.o(.text)
gModes_enter_my_off_mode                0x000004a9   Thumb Code    12  gmodes.o(.text)
gPB_enter_off_config                    0x000004b5   Thumb Code    98  gpb.o(.text)
gPB_enter_default_config                0x00000517   Thumb Code   122  gpb.o(.text)
RTC0FAIL_IRQHandler                     0x000005c9   Thumb Code     8  grtc0.o(.text)
RTC0ALRM_IRQHandler                     0x000005d1   Thumb Code    26  grtc0.o(.text)
gRtc0_enter_off_config                  0x000005eb   Thumb Code    52  grtc0.o(.text)
```

Figure 9. sim3u1xx_Blinky µVision Map File Example

6.2. Determining a Project's Code Size

The Keil µVision IDE automatically displays the code size information at the end of a successful build. After building the si32HAL 1.0.1 **sim3u1xx_Blinky** example, the IDE outputs:

```
Program Size: Code=1968 RO-data=296 RW-data=24 ZI-data=1536

".\build\BlinkyApp.axf" - 0 Error(s), 0 Warning(s).
```

The areas of memory are:

- **Code**: all program code in decimal
- **RO-data**: read-only data located in flash in decimal
- **RW-data**: read-write uninitialized data located in RAM in decimal
- **ZI-data**: zero-initialized data located in RAM in decimal

6.3. Toolchain Library Usage

Some toolchains have multiple libraries or settings that can change the size or execution speed of code.

The Keil µVision tools have two options: standard and MicroLIB. To switch between the two:

1. Right-click on the **project_name** in the **Project** window and select **Options for Target 'project_name'** or go to **Project→Options for Target 'project_name'**.
2. Select the **Target** tab.
3. Use the **Use MicroLIB** checkbox to select the library.

Figure 10 shows this dialog in the µVision IDE.

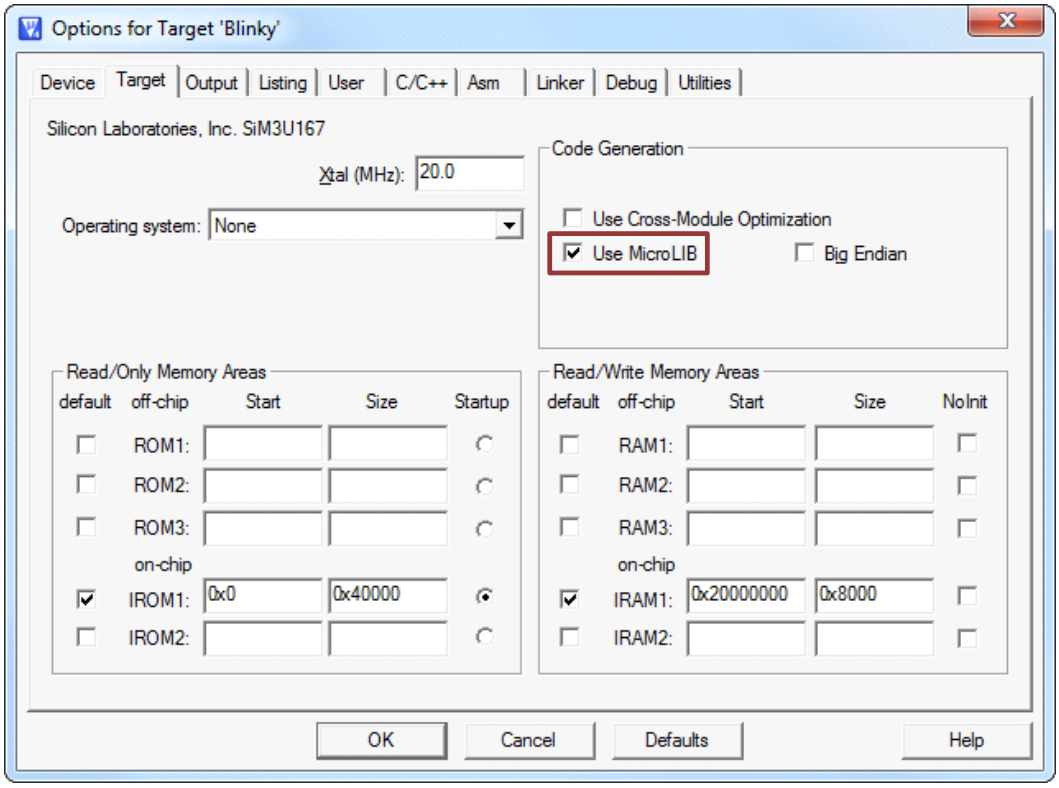


Figure 10. Using the µVision IDE to Select the Project Library

Using the **sim3u1xx_Blinky** and **demo_si32UsbAudio** default examples in the si32HAL 1.0.1 software package, Table 15 and Table 16 show the relative Debug build sizes with the different toolchain library options. Table 17 shows the Debug build sizes for CoreMark, and Table 18 shows the relative CoreMark speed scores for each of these library options.

Table 15. Keil Toolchain Library Usage Comparison—sim3u1xx_Blinky Debug

Library	Code (bytes)	Read Only Data (bytes)	Read-Write Data (bytes)	Zero-Initialized Data (bytes)
µVision standard	2296	312	24	1632
µVision MicroLIB	2068	296	24	1536

Table 16. Keil Toolchain Library Usage Comparison—demo_si32UsbAudio Debug

Library	Code (bytes)	Read Only Data (bytes)	Read-Write Data (bytes)	Zero-Initialized Data (bytes)
µVision standard	51176	4388	5196	18068
µVision MicroLIB	47264	3832	5208	17972

Table 17. Keil Toolchain Library Usage Comparison—CoreMark Debug Size

Library	Code (bytes)	Read Only Data (bytes)	Read-Write Data (bytes)	Zero-Initialized Data (bytes)
µVision standard	13860	868	156	3632
µVision MicroLIB	11276	636	156	3536

Table 18. Keil Toolchain Library Usage Comparison—CoreMark Debug Speed

Library	CoreMark Score
µVision standard	CoreMark 1.0: **65.602324**/ARM4.2 (EDG gcc mode) Iterations=3000/STACK
µVision MicroLIB	CoreMark 1.0:CoreMark 1.0: **69.402323**/ARM4.2 (EDG gcc mode) Iterations=3000/STACK

6.4. Function Library Usage

The removal of debugging **printf()** statements can dramatically reduce the code size of a project. A simple way to do this is to redefine the printf function at the top of the file containing the **printf()** calls using the following statement:

```
#define printf(args...)
```

For si32Library examples such as **demo_si32UsbAudio**, define the statement at the top of myBuildOptions.h to remove all calls to **printf()**. Additionally, reduce the footprint by disabling logging in myBuildOptions.h:

```
#define si32BuildOption_enable_logging 0
```

This method preserves the **printf()** statements for later use, if needed. The **printf()** define can also be encapsulated with preprocessor **#if** statements to automatically include this define when building with a Release configuration.

To verify that all instances of **printf()** have been removed, search the map file for the project for the printf library. In the **sim3u1xx_Blinky** example, this means adding the statement to both the main.c and gCpu.c files.

Using the **sim3u1xx_Blinky** and **demo_si32UsbAudio** default examples in the si32HAL 1.0.1 software package, Table 19 and Table 20 show the relative build sizes with the different printf() settings. This section does not include the CoreMark tests since printf is not part of the CoreMark benchmark.

Table 19. Keil printf() Comparison—sim3u1xx_Blinky Debug

Library	Code (bytes)	Read Only Data (bytes)	Read-Write Data (bytes)	Zero-Initialized Data (bytes)
µVision MicroLIB with printf	2068	296	24	1536
µVision MicroLIB without printf	1392	296	12	1536

Table 20. Keil printf() Comparison—demo_si32UsbAudio Debug

Library	Code (bytes)	Read Only Data (bytes)	Read-Write Data (bytes)	Zero-Initialized Data (bytes)
µVision MicroLIB with printf	47264	3832	5208	17972
µVision MicroLIB without printf	39760	4312	5196	17972

6.5. Toolchain Optimization Settings

In addition to the library types, each toolchain has multiple optimization settings that can affect the resulting code size. In Keil µVision, the optimization settings are set using the following steps:

1. Right-click on the **project_name** in the **Project** window and select **Options for Target 'project_name'** or go to **Project→Options for Target 'project_name'**.
2. Select the **C/C++** tab.
3. Use the **Optimization** drop-down menu to set the project optimization setting.

Figure 11 shows the optimization settings in the IDE.

The available options are:

- **Level 0**: minimum optimization
- **Level 1**: restricted optimization, removing inline functions and unused static functions
- **Level 2**: high optimization
- **Level 3**: maximum optimization with aims to produce faster code or smaller code size than Level 2, depending on the options used

In addition to the levels, µVision also has an **Optimize for Time** selection available below the **Optimization** drop-down menu. Declaring a variable as **volatile** will prevent the compiler from optimizing out the variable.

More information on these optimization levels can be found on the Keil website (http://www.keil.com/support/man/docs/uv4/uv4_dg_adscc.htm).

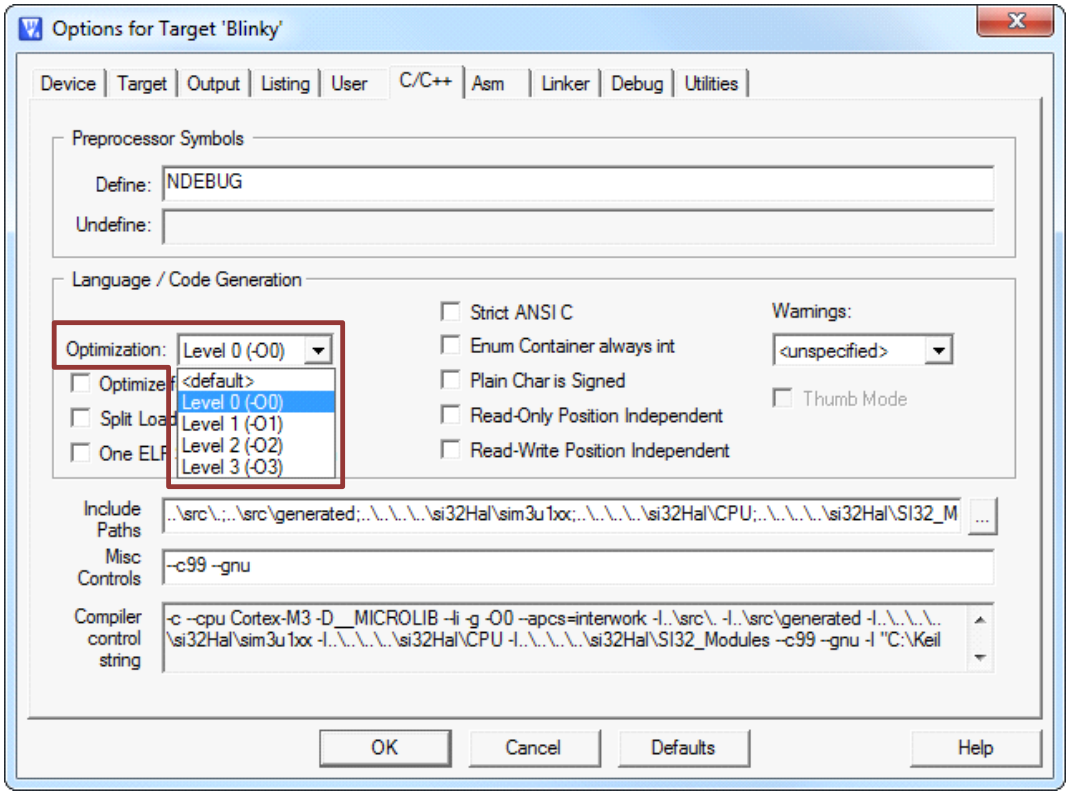


Figure 11. Setting the Project Optimization in the µVision IDE

Using the **sim3u1xx_Blinky** and **demo_si32UsbAudio** default examples in the si32HAL 1.0.1 software package, Table 21 and Table 22 show the relative Debug build sizes with the different optimization level settings. Table 23 shows the CoreMark Debug build sizes, and Table 24 lists the CoreMark speed scores for these optimization levels.

Table 21. Keil Toolchain Optimization Comparison—sim3u1xx_Blinky Debug

Library	Code (bytes)	Read Only Data (bytes)	Read-Write Data (bytes)	Zero-Initialized Data (bytes)
µVision MicroLIB -O0	2068	296	24	1536
µVision MicroLIB -O0 (with **Optimize for Time**)	2068	296	24	1536
µVision MicroLIB -O1	1704	296	20	1536
µVision MicroLIB -O1 (with **Optimize for Time**)	1648	296	20	1536
µVision MicroLIB -O2	1616	296	20	1536
µVision MicroLIB -O2 (with **Optimize for Time**)	1600	296	20	1536
µVision MicroLIB -O3	1604	296	20	1536
µVision MicroLIB -O3 (with **Optimize for Time**)	1596	296	20	1536

Table 22. Keil Toolchain Optimization Comparison—demo_si32UsbAudio Debug

Library	Code (bytes)	Read Only Data (bytes)	Read-Write Data (bytes)	Zero-Initialized Data (bytes)
µVision MicroLIB -O0	47264	3832	5208	17972
µVision MicroLIB -O0 (with **Optimize for Time**)	47264	3832	5208	17972
µVision MicroLIB -O1	38816	3832	5132	17952
µVision MicroLIB -O1 (with **Optimize for Time**)	39924	3832	5132	17952
µVision MicroLIB -O2	36540	3832	5132	17952
µVision MicroLIB -O2 (with **Optimize for Time**)	39840	3832	5132	17952
µVision MicroLIB -O3	36468	3832	5132	17952
µVision MicroLIB -O3 (with **Optimize for Time**)	41532	3832	5132	17952

Table 23. Keil Toolchain Optimization Comparison—CoreMark Debug Size

Library	Code (bytes)	Read Only Data (bytes)	Read-Write Data (bytes)	Zero-Initialized Data (bytes)
µVision MicroLIB -O0	11276	636	156	3536
µVision MicroLIB -O0 (with **Optimize for Time**)	11276	636	156	3536
µVision MicroLIB -O1	9788	616	140	3536
µVision MicroLIB -O1 (with **Optimize for Time**)	10136	616	140	3536
µVision MicroLIB -O2	9640	616	140	3536
µVision MicroLIB -O2 (with **Optimize for Time**)	10684	616	140	3536
µVision MicroLIB -O3	9680	616	140	3536
µVision MicroLIB -O3 (with **Optimize for Time**)	11500	616	140	3536

Table 24. Keil Toolchain Optimization Comparison—CoreMark Debug Speed

Library	CoreMark Score
µVision MicroLIB -O0	CoreMark 1.0: **69.402323** / ARM4.2 (EDG gcc mode) Iterations=3000 / STACK
µVision MicroLIB -O0 (with **Optimize for Time**)	CoreMark 1.0: **69.402323** / ARM4.2 (EDG gcc mode) Iterations=3000 / STACK
µVision MicroLIB -O1	CoreMark 1.0: **75.279256** / ARM4.2 (EDG gcc mode) Iterations=3000 / STACK
µVision MicroLIB -O1 (with **Optimize for Time**)	CoreMark 1.0: **75.206352** / ARM4.2 (EDG gcc mode) Iterations=3000 / STACK
µVision MicroLIB -O2	CoreMark 1.0: **74.247855** / ARM4.2 (EDG gcc mode) Iterations=3000 / STACK
µVision MicroLIB -O2 (with **Optimize for Time**)	CoreMark 1.0: **87.277701** / ARM4.2 (EDG gcc mode) Iterations=3000 / STACK
µVision MicroLIB -O3	CoreMark 1.0: **79.520321** / ARM4.2 (EDG gcc mode) Iterations=3000 / STACK
µVision MicroLIB -O3 (with **Optimize for Time**)	CoreMark 1.0: **102.697150** / ARM4.2 (EDG gcc mode) Iterations=3000 / STACK

6.6. Unused Code Removal

Each file in a project becomes an object that is included. In other words, if any functions in a file are used, then the entire file is included by default. This can become an issue for a project using the si32HAL and only a few functions from each module.

Removed (unused) functions can be viewed in the map files for the projects.

The unused code removal feature is not automatically enabled in the Keil µVision IDE. To enable this feature:

1. Right-click on the **project_name** in the **Project** window and select **Options for Target 'project_name'** or go to **Project→Options for Target 'project_name'**.
2. Select the **C/C++** tab.
3. Use the **One ELF Section per Function** checkbox to enable or disable unused code removal.

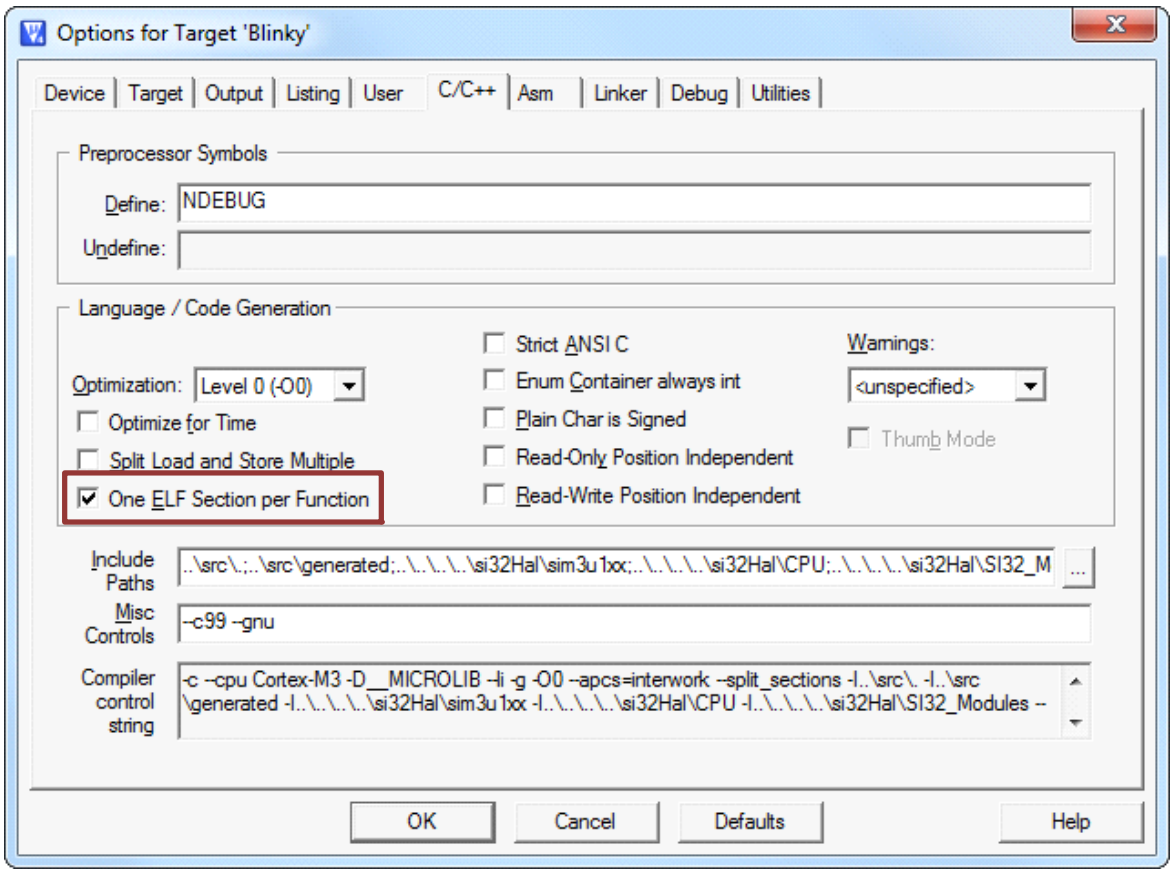


Figure 12. Setting the Remove Unused Code Option in the µVision IDE

Using the **sim3u1xx_Blinky** and **demo_si32UsbAudio** default examples in the si32HAL 1.0.1 software package, Table 25 and Table 26 show the relative Debug build sizes with different unused code removal settings. Table 27 shows the CoreMark build sizes, and Table 28 shows the CoreMark scores for the different unused code removal settings.

Table 25. Keil Unused Code Removal Comparison—sim3u1xx_Blinky Debug

Library	Code (bytes)	Read Only Data (bytes)	Read-Write Data (bytes)	Zero-Initialized Data (bytes)
µVision MicroLIB with no unused code removal	1392	296	12	1536
µVision MicroLIB with unused code removal	1184	296	12	1536

Table 26. Keil Unused Code Removal Comparison—demo_si32UsbAudio Debug

Library	Code (bytes)	Read Only Data (bytes)	Read-Write Data (bytes)	Zero-Initialized Data (bytes)
µVision MicroLIB with no unused code removal	47264	3832	5208	17972
µVision MicroLIB with unused code removal	43464	3772	5060	17780

Table 27. Keil Unused Code Removal Comparison—CoreMark Debug Size

Library	Code (bytes)	Read Only Data (bytes)	Read-Write Data (bytes)	Zero-Initialized Data (bytes)
µVision MicroLIB with no unused code removal	11276	636	156	3536
µVision MicroLIB with unused code removal	11012	636	156	3536

Table 28. Keil Unused Code Removal Comparison—CoreMark Debug Speed

Library	CoreMark Score
µVision MicroLIB with no unused code removal	CoreMark 1.0: **69.402324** / ARM4.2 (EDG gcc mode) Iterations=3000 / STACK
µVision MicroLIB with unused code removal	CoreMark 1.0: **67.374626** / ARM4.2 (EDG gcc mode) Iterations=3000 / STACK

6.7. Reset Sequence

The speed of the reset sequence of a device can be an important factor, especially for devices like the SiM3U1xx/SiM3C1xx that require a reset to exit the lowest power mode.

After the hardware jumps to the reset vector and loads the stack pointer address, the core must initialize the memory of the device. This involves copying data from flash to RAM and zero-filling any zero-initialized segments. Then, the reset code typically calls a system initialization function and jumps to main.

This reset sequence may take different times based on the library used with the project. The startup code should always be compiled with the fastest speed optimization to ensure it takes as little time as possible.

The si32HAL examples have a ~500 ms delay added to a pin reset event to prevent code from switching to a non-existent clock source and disable the device. This delay can be removed by defining the **si32HalOption_disable_pin_reset_delay** symbol in the project.

To define a symbol in Keil µVision:

1. Right-click on the **project_name** in the **Project** window and select **Options for Target 'project_name'** or go to **Project→Options for Target 'project_name'**.
2. Select the **C/C++** tab.
3. Use the **Define** text box to add or remove project symbols.

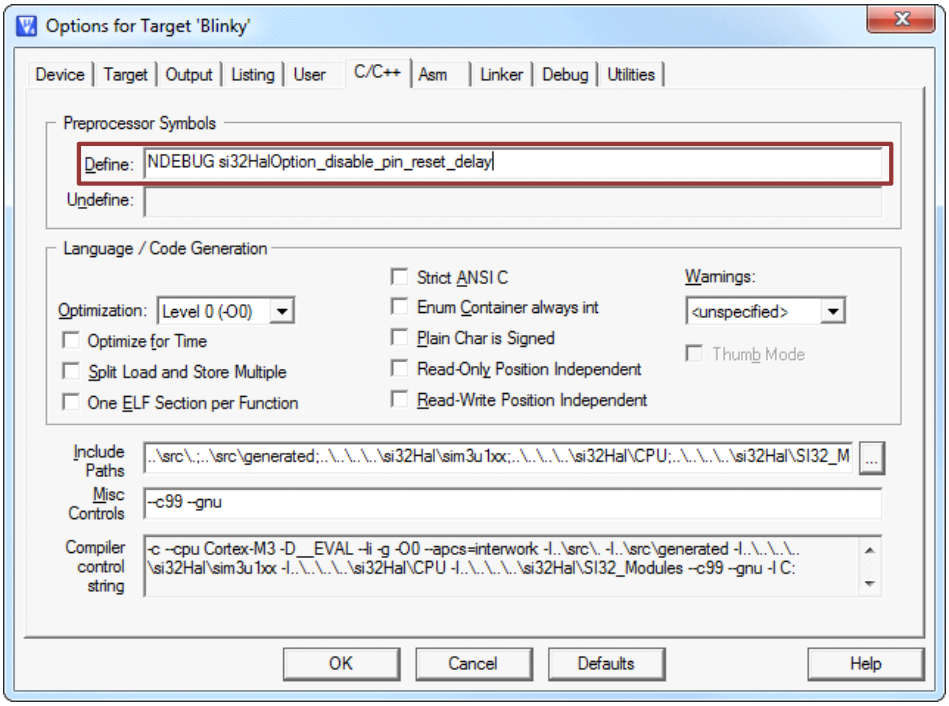


Figure 13. Adding a Project Define Symbol in the µVision IDE

Table 29 shows the reset time comparison for the toolchain libraries using the fastest speed optimization on the start up code. This time was measured using the **sim3u1xx_Blinky** example in Debug mode from the rise of RESETb to the fall of a port pin at the beginning of main() on an oscilloscope.

Table 29. Keil Toolchain Library Usage Comparison—sim3u1xx_Blinky Debug Reset Sequence

Library	Reset Time (µs)
µVision standard	52
µVision MicroLIB	48

7. IAR Embedded Workbench IDE

This section discusses ways to optimize projects using the IAR Embedded Workbench ID. The IAR tools used for the code size and execution speed testing discussed in this document are IAR Embedded Workbench for ARM version 6.50.1.4388 and IAR Embedded Workbench common components version 6.5.0.2458.

7.1. Reading the Map File

The first step in the code size optimization process is to analyze the project map file and determine what portions of code take the most space.

The map file is an output of the linker that shows the size of each function and variable and their positions in memory. This map file is located in the build files for a project.

In addition to the functions, the map file includes information on variables and other symbols, including unused functions that are removed.

For an IAR Debug build, the map file is located in the project's **Debug\List** directory. Figure 14 shows an excerpt of the sim3u1xx_Blinky Debug example map file.

For each function in the project, the map file lists the starting address and the length. For example, in this build of the sim3u1xx_Blinky example, the **my_rtc_alarm0_handler** function starts at address 0x0000170B and occupies 48 bytes of memory.

```
                          0x000018f9   0xbc  Code  Gb  gRB.o [1]
gRtc0_enter_default_config
                          0x00001649   0x96  Code  Gb  gRtc0.o [1]
main                      0x00001755   0xc0  Code  Gb  main.o [1]
memchr                    0x00000d25         Code  Gb  memchr.o [5]
msTicks                   0x20000004    0x4  Data  Gb  gCpu.o [1]
msTicks_10                0x20000008    0x4  Data  Gb  myRtc0.o [1]
mySystemInit              0x000019f9   0x30  Code  Gb  myCpu.o [1]
my_rtc_alarm0_handler     0x0000170b   0x30  Code  Gb  myRtc0.o [1]
my_rtc_fail_handler       0x00001709    0x2  Code  Gb  myRtc0.o [1]
pad                       0x000006a3   0x3a  Code  Lc  xprintflarge.o [3]
printf                    0x00001825   0x24  Code  Gb  printf.o [3]
putchar                   0x00001a75   0x28  Code  Gb  putchar.o [3]
scale                     0x000007ef   0x46  Code  Lc  xprintflarge.o [3]
strchr                    0x00000cd5         Code  Gb  strchr.o [5]
strlen                    0x00000ced         Code  Gb  strlen.o [5]
```

Figure 14. sim3u1xx_Blinky IAR Debug Map File Example

7.2. Determining a Project's Code Size

Each project's library and function usage is different. Analyzing the project's makeup can help determine the most effective way to reduce code space.

The default IAR IDE settings does not automatically output the code and RAM size after a build. To view the build size of a project:

1. Right-click on the **Messages** window at the bottom of the IDE.
2. Select **Options...**.
3. Under **Messages**, click the **Show build messages** drop-down menu and select **All**.

After building the si32HAL 1.0.1 **sim3u1xx_Blinky** example with the default project settings, the IDE outputs:

```
      7 298 bytes of readonly code memory

         90 bytes of readonly data memory

      1 036 bytes of readwrite data memory
```

The values reported by the IDE are given in decimal.

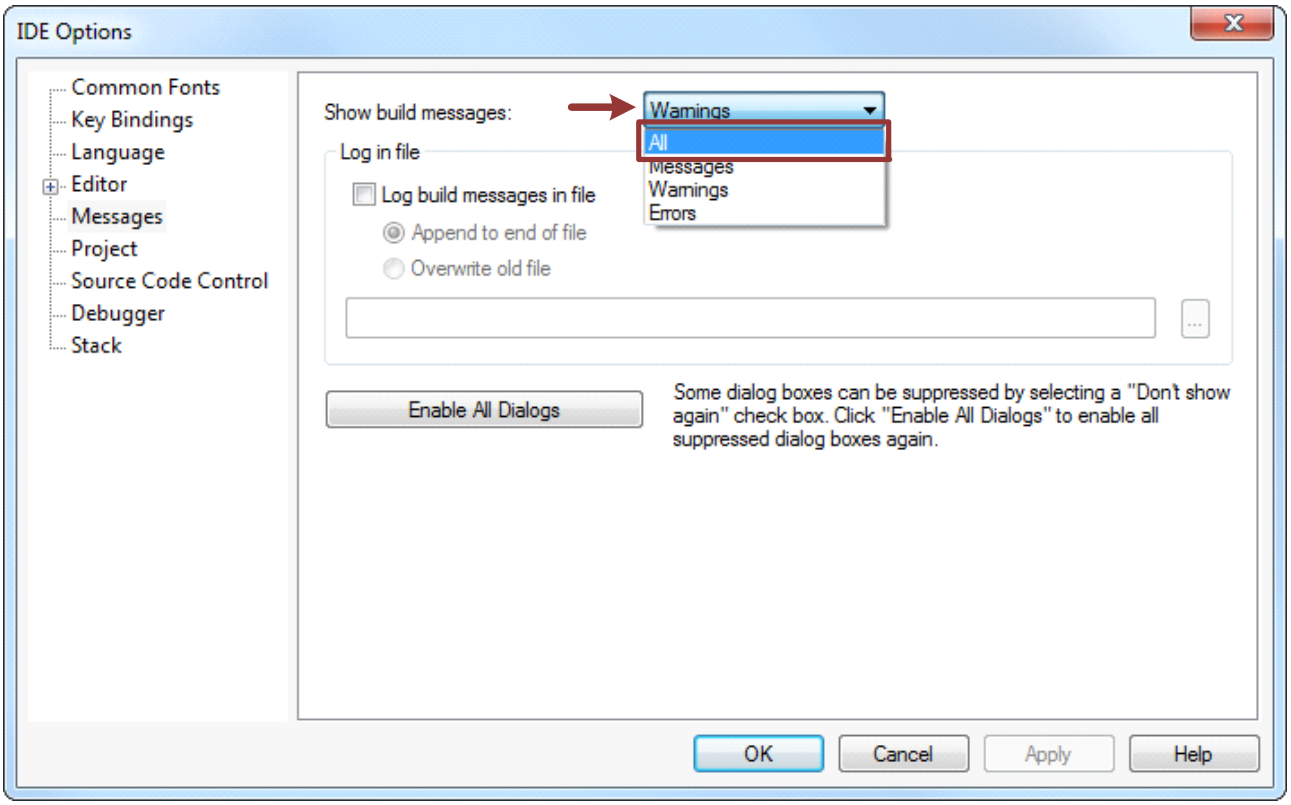


Figure 15. Automatically Reporting Project Size on Project Build in IAR Embedded WorkBench

7.3. Toolchain Library Usage

Some toolchains have multiple libraries or settings that can change the size or execution speed of code.

The IAR Embedded Workbench tools have three options: Normal, Full, and Custom. In addition, there are **Semihosted** and **None** library low-level interface implementations. To switch between these:

1. With the project open and not in debug mode, select the project name in the Workspace window.
2. Go to **Project→Options...**.
3. Under **Category** on the left, select **General Options**.
4. Click the **Library Configuration** tab.

Figure 16 shows this dialog in the IAR Embedded Workbench IDE.

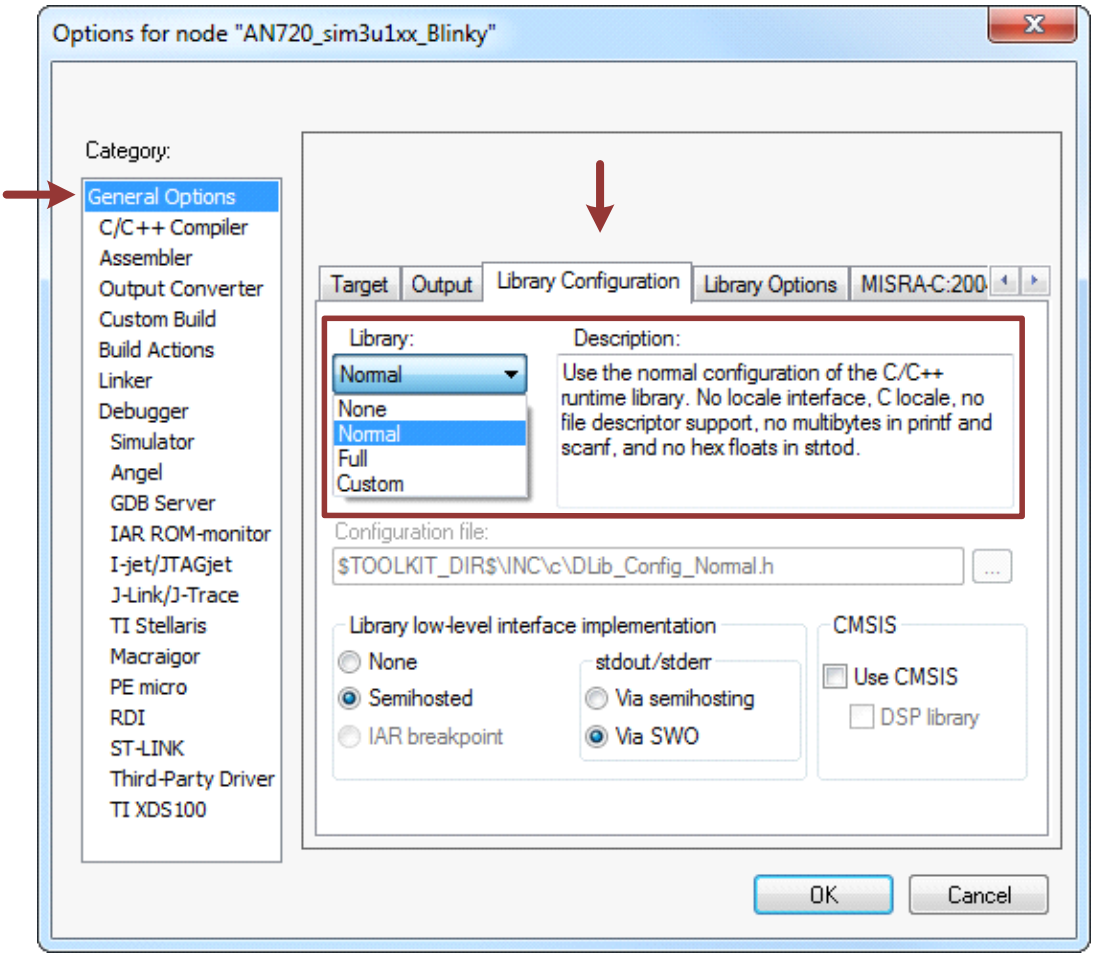


Figure 16. Using the IAR Embedded Workbench IDE to Select the Project Library

Using the **sim3u1xx_Blinky** examples in the si32HAL 1.0.1 software package, Table 30 shows the relative Debug build sizes with the **Normal** and **Full** library options using **Semihosting**. The example optimization settings are **None**, and printf() uses the **Large without multibytes** option. Table 31 shows the Debug build sizes for CoreMark, and Table 32 shows the relative CoreMark speed scores for each of these library options.

Table 30. IAR Toolchain Library Usage Comparison—sim3u1xx_Blinky Debug

Library	Code (bytes)	Read Only Data (bytes)	Read-Write Data (bytes)	Zero-Initialized Data (bytes)
Normal	7310	90	1036	N/A
Full	10408	238	3864	N/A

Table 31. IAR Toolchain Library Usage Comparison—CoreMark Debug Size

Library	Code (bytes)	Read Only Data (bytes)	Read-Write Data (bytes)	Zero-Initialized Data (bytes)
Normal	12866	1750	3160	N/A
Full	14836	1890	5872	N/A

Table 32. IAR Toolchain Library Usage Comparison—CoreMark Debug Speed

Library	CoreMark Score
Normal	CoreMark 1.0: **58.056593** / IAR ANSI C/C++ Compiler V6.50.1.4354/W32 for ARM Iterations=3000 / STACK
Full	CoreMark 1.0: **53.463034** / IAR ANSI C/C++ Compiler V6.50.1.4354/W32 for ARM Iterations=3000 / STACK

7.4. Function Library Usage

Function libraries such as floating point math and **printf()** can significantly increase the size of a project. If a project is constrained by size, a careful analysis of the usage of these large libraries may be required. For example, floating point can often be approximated well by fixed point math, eliminating the need for the floating point libraries.

The **printf()** library is often needed by projects for debugging or release code. If **printf()** is used for debugging purposes, using a defined symbol in the project to remove **printf()** when compiling a release build can dramatically reduce the size of a project. To define a symbol to differentiate between a Debug project and a Release project, see “7.6. Reset Sequence”. The code can then use **#ifdef**...**#endif** preprocessor statements to remove debugging code or **printf()** calls.

The removal of debugging **printf()** statements can dramatically reduce the code size of a project. A simple way to do this is to redefine the printf function at the top of the file containing the **printf()** calls using the following statement:

```
#define printf(args...)
```

This method preserves the **printf()** statements for later use, if needed. The **printf()** define can also be encapsulated with preprocessor **#if** statements to automatically include this define when building with a Release configuration.

To verify that all instances of **printf()** have been removed, search the map file for the project for the printf library. In the **sim3u1xx_Blinky** example, this means adding the statement to both the main.c and gCpu.c files.

The IAR tools have many options for the **printf()** library used by code. To access these options:

1. With the project open and not in debug mode, select the project name in the Workspace window.
2. Go to **Project→Options...**.
3. Under **Category** on the left, select **General Options**.
4. Click the **Library Options** tab.

The **Printf formatter** section of this tab has options for **Auto**, **Full**, **Full without multibytes**, **Large**, **Large without**

multibytes, **Small**, **Small without multibytes**, and **Tiny**. A box under the options list provides a brief description for each setting. A similar menu is also available for **scanf()**.

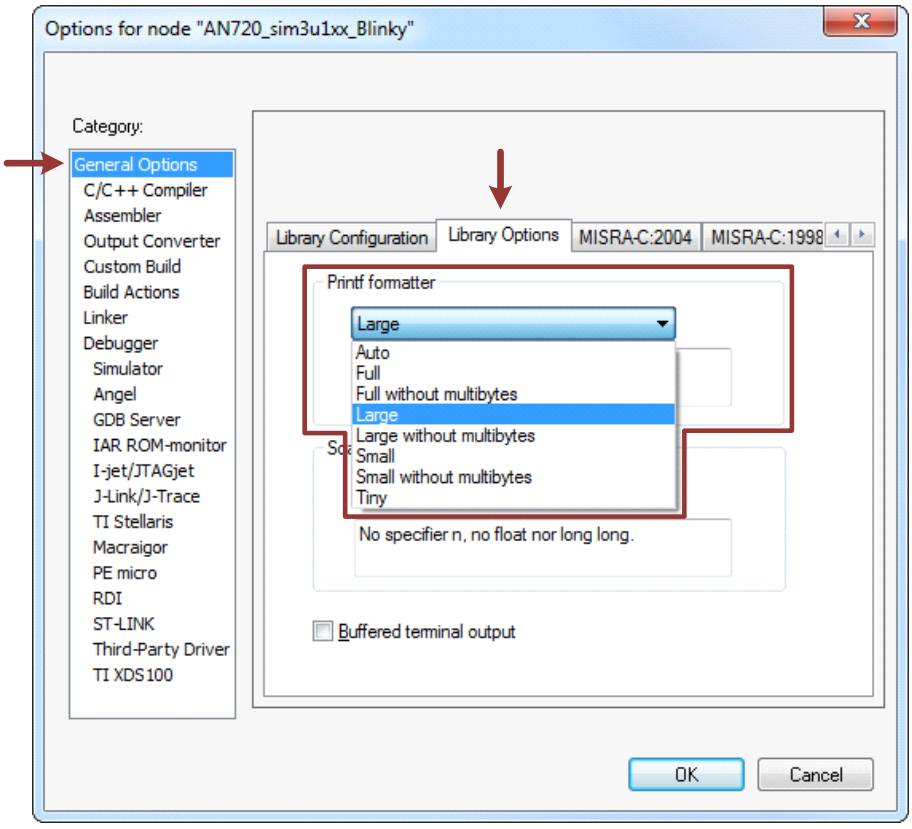


Figure 17. Printf() Formatter Options in IAR Embedded WorkBench

Using the **sim3u1xx_Blinky** default example in the si32HAL 1.0.1 software package, Table 33 shows the relative build sizes with the different no-multibyte **printf()** settings and no code optimization. This section also does not include the CoreMark tests since **printf()** is not part of the CoreMark benchmark.

Table 33. IAR printf() Comparison—sim3u1xx_Blinky Debug

Library	Code (bytes)	Read Only Data (bytes)	Read-Write Data (bytes)	Zero-Initialized Data (bytes)
Auto printf included	2426	90	1036	N/A
Full without multibytes printf included	8098	90	1036	N/A
Large without multibytes printf included	7310	90	1036	N/A
Small without multibytes printf included	3394	90	1036	N/A
Tiny printf included	2426	90	1036	N/A
printf not included	1716	32	1036	N/A

7.5. Toolchain Optimization Settings

Each toolchain has multiple optimization settings that can affect the resulting code size. With the IAR toolchain, code optimization can be set by following these steps:

1. With the project open and not in debug mode, select the project name in the Workspace window.
2. Go to **Project→Options...**.
3. Under **Category** on the left, select **C/C++ Compiler**.
4. Click the **Optimizations** tab.

Figure 18 shows the optimization settings for the IAR Embedded Workbench IDE. Level **None** has the least optimization, while **High** has the most optimization. The **High** setting has three additional settings: **Balanced**, **Size**, and **Speed**. On any of the optimization settings, the **Enabled transformations** algorithms can be customized for the particular project.

Declaring a variable as **volatile** will prevent the compiler from optimizing out the variable.

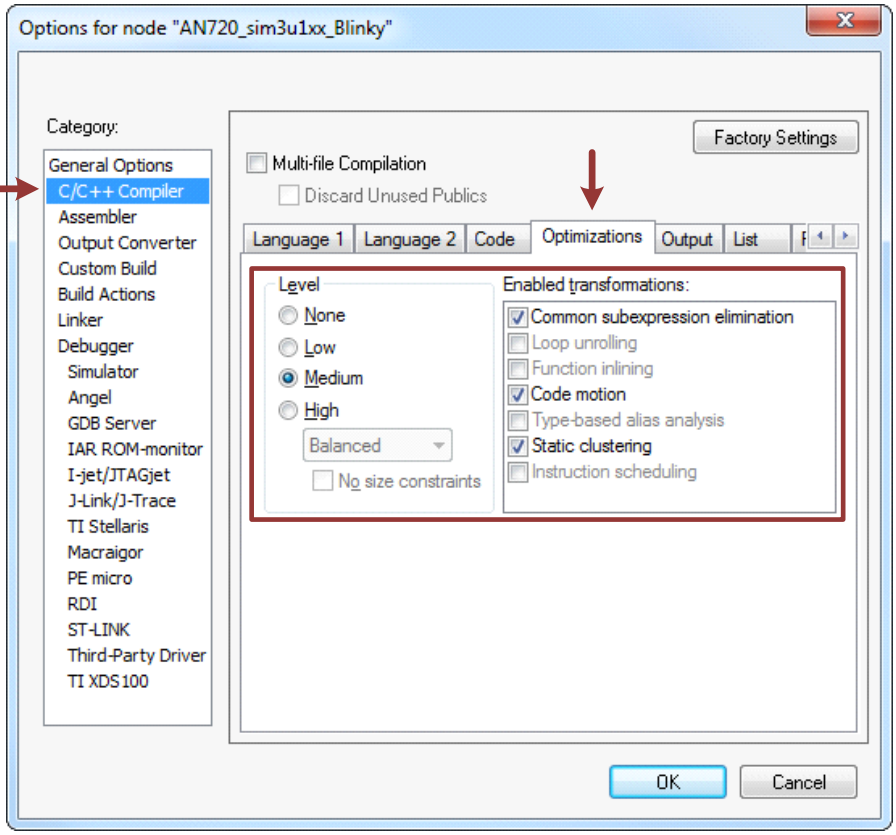


Figure 18. Setting the Project Optimization in the IAR Embedded Workbench IDE

Using the **sim3u1xx_Blinky** example in the si32HAL 1.0.1 software package, Table 34 shows the relative Debug build sizes with the different optimization level settings. Table 35 shows the CoreMark Debug build sizes, and Table 36 lists the CoreMark speed scores for these optimization levels using the default **Enabled transformations** settings.

Table 34. IAR Toolchain Optimization Comparison—sim3u1xx_Blinky Debug

Library	Code (bytes)	Read Only Data (bytes)	Read-Write Data (bytes)	Zero-Initialized Data (bytes)
None	7310	90	1036	N/A
Low	7298	90	1036	N/A
Medium	7100	32	1036	N/A
High Balanced	7020	32	1036	N/A
High Size	7020	32	1036	N/A
High Speed	7004	32	1036	N/A

Table 35. IAR Toolchain Optimization Comparison—CoreMark Debug Size

Library	Read Only Code (bytes)	Read Only Data (bytes)	Read-Write Data (bytes)	Zero-Initialized Data (bytes)
None	12866	1750	3160	N/A
Low	12570	1750	3160	N/A
Medium	12752	363	3160	N/A
High Balanced	12404	363	3160	N/A
High Size	12220	363	3160	N/A
High Speed	13148	363	3160	N/A

Table 36. IAR Toolchain Optimization Comparison—CoreMark Debug Speed

Library	CoreMark Score
None	CoreMark 1.0: **58.056593** / IAR ANSI C/C++ Compiler V6.50.1.4354/W32 for ARM Iterations=3000 / STACK
Low	CoreMark 1.0: **65.223347** / IAR ANSI C/C++ Compiler V6.50.1.4354/W32 for ARM Iterations=3000 / STACK
Medium	CoreMark 1.0: **75.953463** / IAR ANSI C/C++ Compiler V6.50.1.4354/W32 for ARM Iterations=3000 / STACK
High Balanced	CoreMark 1.0: **106.334820** / IAR ANSI C/C++ Compiler V6.50.1.4354/W32 for ARM Iterations=3000 / STACK
High Size	CoreMark 1.0: **88.438785** / IAR ANSI C/C++ Compiler V6.50.1.4354/W32 for ARM Iterations=3000 / STACK
High Speed	CoreMark 1.0: **119.070769** / IAR ANSI C/C++ Compiler V6.50.1.4354/W32 for ARM Iterations=3000 / STACK

7.6. Reset Sequence

The speed of the reset sequence of a device can be an important factor, especially for devices like the SiM3U1xx/SiM3C1xx that require a reset to exit the lowest power mode.

After the hardware jumps to the reset vector and loads the stack pointer address, the core must initialize the memory of the device. This involves copying data from flash to RAM and zero-filling any zero-initialized segments. Then, the reset code typically calls a system initialization function and jumps to main.

This reset sequence may take different times based on the library used with the project. The startup code should always be compiled with the fastest speed optimization to ensure it takes as little time as possible.

The si32HAL examples have a ~500 ms delay added to a pin reset event to prevent code from switching to a non-existent clock source and disable the device. This delay can be removed by defining the **si32HalOption_disable_pin_reset_delay** symbol in the project.

To define a symbol in the IAR Embedded Workbench IDE:

1. With the project open and not in debug mode, select the project name in the Workspace window.
2. Go to **Project→Options...**.
3. Under **Category** on the left, select **C/C++ Compiler**.
4. Click the **Preprocessor** tab.
5. Add the symbol in the **Defined symbols** box.

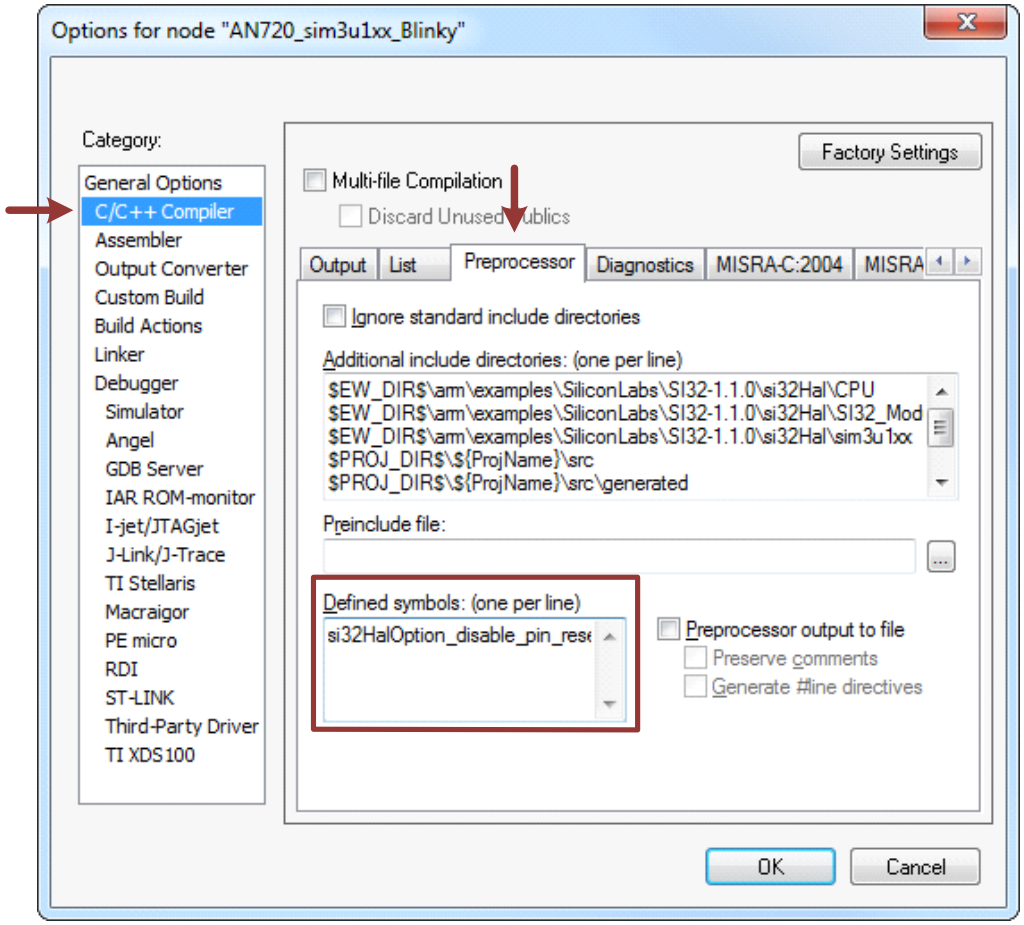
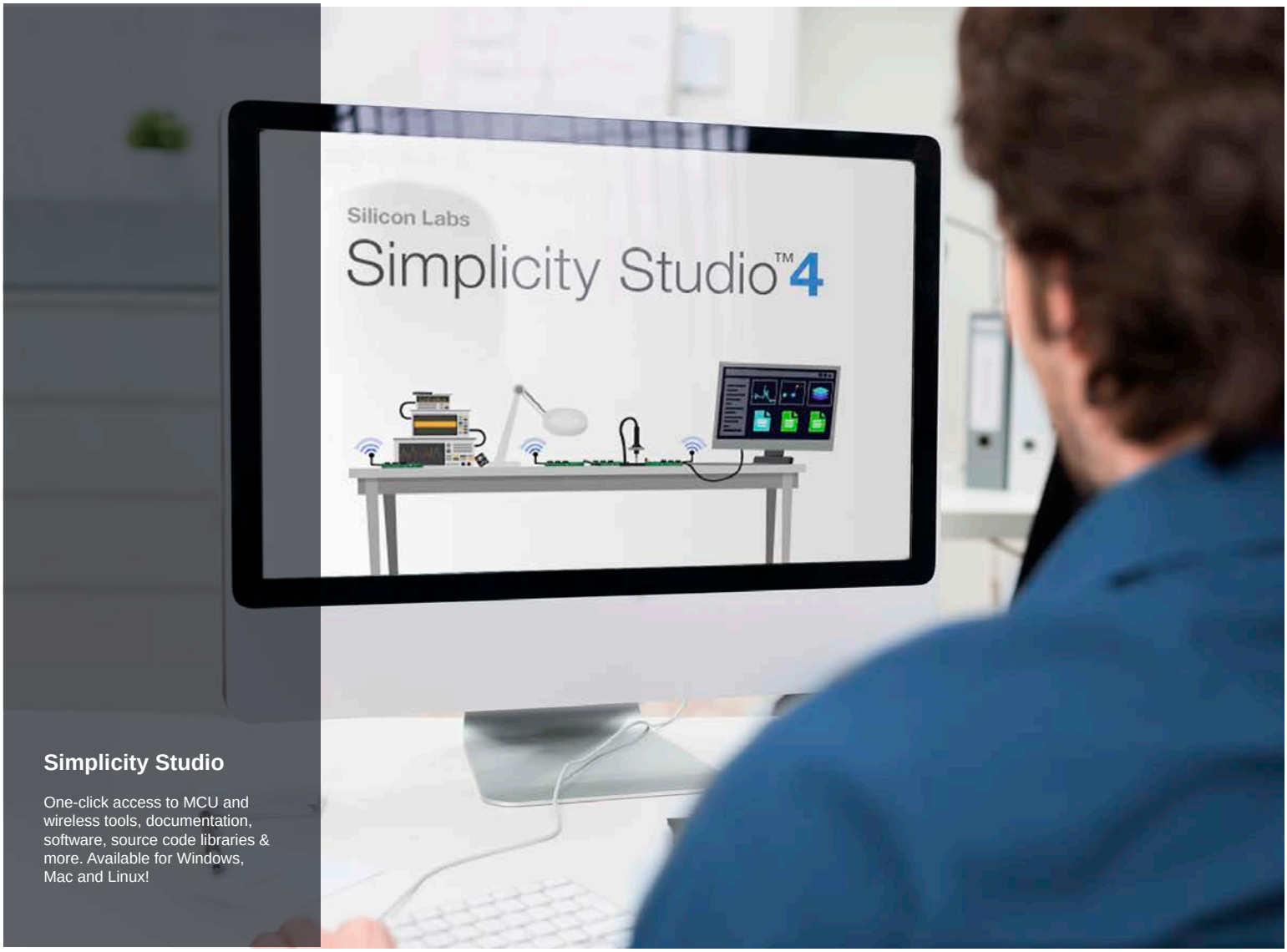


Figure 19. Adding a Project Define Symbol in the IAR Embedded Workbench IDE

Table 37 shows the reset time comparison for the toolchain libraries using the fastest speed optimization on the start up code. This time was measured using the **sim3u1xx_Blinky** example in Debug mode from the fall of a port pin at the beginning of the Reset IRQ handler to the fall of a port pin at the beginning of main() on an oscilloscope. This test requires modification of the si32HAL startup sequence file **startup_<device>_iar.s**.

Table 37. IAR Toolchain Library Usage Comparison—sim3u1xx_Blinky Debug Reset Sequence

Library	Reset Time (µs)
Normal with printf()	72
Full with printf()	500
Normal with printf() removed	72
Full with printf() removed	72



Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com