Application Report

SPRAAC3 – November 2005

ARM Compiler Tips and Code Size Optimization Using DSP/BIOS Link

Aarti Goyle, George Mock
Arnie Reynoso *SDO Applications – SDO Customer Support*

ABSTRACT

This application note demonstrates techniques for use with DSP/BIOS Link on TI's Code Generation ARM (TMS470). The techniques shown optimize code size and take advantage of new compiler features.

DSP/BIOS Link is a standard for communication between TI DSPs and GPPs (General Purpose Processors). It provides for both data streaming and control and status messaging. The software examples used in this document are based on DSP/BIOS Link 1.20.03 release for the DA295 platform.

All techniques described here can be applied to any ARM-based system to reduce code size when using the TI ARM compiler. This document covers issues such as changes in the Application Binary Interface (ABI), managing code that executes in both ARM and Thumb mode, and linker trampolines. It also compares and discusses code sizes using several different compiler build options.

Contents

1 Introduction 2
2 Hardware and Software Setup 2
3 Building the Baseline DSP/BIOS Link and Message Application 3
4 ABI Changes in TMS470 Code Generation Tools 3
4.1 Difference Between TIABI and TI_ARM9_ABI 4
4.2 Migrating Projects from CCStudio 2.2x to CCStudio 3.1 4
5 Rebuilding Baseline DSP/BIOS Link and Message Application 5
6 Trampolines 6
6.1 Impact of Trampolines on DSP/BIOS Link 6
6.2 Background on Trampolines 7
6.3 Version Notes on Enabling Trampolines 8
6.4 Trampoline Details in Link Map File 8
7 Mixing Arm Mode and Thumb Mode Code 9
7.1 Issue Encountered When Mixing Thumb and Arm Modes 9
7.2 Detecting Thumb and Arm Mode Conflicts 10
8 Code Size Data 11
8.1 Collecting the Data 11
8.2 Optimization Levels: –o3 Option 12
8.3 Thumb Mode: –mt Option 12
8.4 Automatic Inlining: –oi0 Option 13

8.5 Functions in Subsections: –ms Option 14
8.6 Branch Chaining: –ab=0 Option 15
9 Performance Data....16
9.1 Data Collection 16
9.2 Analysis 17
10 Conclusions....17
11 References....17

Figures

Figure 1. Build Options Dialog Showing --abi Option....5
Figure 2. CPSR Register Bits After Undefined Instruction....10
Figure 3. Code Size vs. Compiler Options....11
Figure 4. Code Size for Thumb Builds....12
Figure 5. Performance Impact of Compiler Options....16

Tables

Table 1. CCStudio and Code Generation Tools Versions....8

1 Introduction

This document shows the effects of some new code generation options available with the latest TMS470 (ARM) code generation tools. It examines the code size and performance of DSP/BIOS Link on the GPP side for the DA295 EVM with PrOS. The steps involve rebuilding the GPP-side DSP/BIOS Link with several different TMS470 code generation tool options and measuring the impact of each option on the Link footprint using internal utilities. This document uses the DSP/BIOS Link Message application as the test application.

2 Hardware and Software Setup

Hardware

- DA295 EVM
- Spectrum Digital XDS510 USB emulator

Software Packages	Chosen Install Directory
• Code Composer Studio v3.1	c:\CCStudio_v3.1
• DSP/BIOS Link for DA295 EVM with PrOS 1.20.02	c:\dsplink
• DA295 PSP	c:\asp-psp
• DSP/BIOS 5.10.02	c:\bios_5_10_02

If the software packages are installed elsewhere, modify the examples in this document accordingly to use the appropriate directories.

3 Building the Baseline DSP/BIOS Link and Message Application

To provide baseline statistics, we built the DSP/BIOS Link kernel and the Message application using the default options in CCStudio 3.1. See the *DSP/BIOS Link User's Guide* (LNK 093 USR) for details on the build process for the DSP/BIOS Link kernel on the GPP and DSP sides as well as details for the Message application.

Next, we built a sample project in CCStudio 3.1 following these steps:

1. Use CCStudio Setup to choose the DA295 EVM.
2. Start CCStudio and open the ARM-side debugger.
3. Open the hibari_sample.pjt project in C:\asp-psp\system\hibari\pros\hibari_evm\build.
4. Build the hibari_sample.pjt project.

The build results in several linker warnings such as the following:

```
>> warning: Cannot find state of callee routine at addr 24 called from module
            ../../lib/Debug/hibari_prKernel32.lib
>> warning: Cannot find state of callee routine at addr 4 called from module
            ../../lib/Debug/hibari_prKernel32.lib
>> warning: Cannot find state of callee routine at addr 4 called from module
            ../../../../../os/lib/Debug/hibari_palos_pros.lib
>> warning: Cannot find state of callee routine at addr 28 called from module
            ../../../../../os/lib/Debug/hibari_palos_pros.lib
```

These warnings are generated because the default ABI (Application Binary Interface) has changed since v4.1.0 of the TMS470 Code Generation Tools. A TMS470 project containing a mix of C and assembly code written for older toolsets generates these warnings when built with the default options in CCStudio 3.1. These warnings also appear when you build a project with the default options in CCStudio 3.1 and link with libraries built with older versions of the code generation tools.

You should not ignore these warnings. The following sections provide background information on the ABI and on options you can use to eliminate the linker warnings. Section 5, "Rebuilding Baseline DSP/BIOS Link and Message Application" specifically discusses how to eliminate linker warnings that occur when you build the sample application.

4 ABI Changes in TMS470 Code Generation Tools

The v4.1.0 release of the ARM (TMS470) Code Generation Tools introduced a new ABI (Application Binary Interface) for the compiler.

An ABI defines how an object file is compiled. It covers items such as language specifics, calling conventions, and the object file format. Object files that conform to the same ABI, even if built by different compiler vendors, can be linked together into a single cohesive executable.

For v4.1.x of the TMS470 compiler, changes to the ABI are limited to function naming conventions and linking dual-mode code. These changes were made to take advantage of new capabilities present in V5E (ARM9) and higher architectures. The new ABI is called **TI_ARM9_ABI**. The old ABI in compiler releases prior to 4.1.0 is now referred to as **TIABI**.

4.1 Difference Between TIABI and TI_ARM9_ABI

Function names are generated differently with the new ABI. Under the older TIABI, Thumb mode routine symbols require a “$” prefix and Arm mode routine symbols require a “_” prefix. To allow calls from functions in the opposite mode, the code generator created function veneers to handle the mode change. For example, consider an Arm mode function calling a Thumb mode function called foo(). The Arm mode function calls foo() with a BL _foo instruction. The veneer for “foo”, which has _foo as the entry point label changes the mode and then correctly calls $foo. The return from $foo goes back through the veneer to change the mode back from Thumb to Arm.

With the TI_ARM9_ABI (the default with v4.1.0 and higher tools), the function naming convention is uniform. All function names are prefixed with an “_” (underscore). The linker determines the mode of both the caller and callee and generates the necessary code to make the call. For example, if the target is an ARM9 (V5E) or higher architecture, the linker replaces a BL instruction with a BLX instruction. If the target is an ARM7 (V4), or the linker cannot determine the mode of the caller, the linker generates the necessary veneer code. The compiler does not generate function veneers in the assembly file.

Since the v4.1.x compilers, by default, no longer use the “$” and “_” prefixes to mark the state of a function, this information is passed to the linker using two new assembler directives: .thumbfunc and .armfunc.

For example:

```
.thumbfunc _foo            ; foo is a THUMB mode function
```

or

```
.armfunc _foo              ; foo is an ARM mode function
```

4.2 Migrating Projects from CCStudio 2.2x to CCStudio 3.1

Typical applications fall into one of the following categories. The changes you need to make to migrate code to use the new tools are described in the following list for each category.

- **All C code.** Rebuild everything for TI_ARM9_ABI. The run-time library in the project should be the appropriate one built for TI_ARM9_ABI. Valid choices include rts32.lib and rts16.lib. The list of available run-time libraries is documented in the CGTNew.txt file included with the code generation tools (in the c:\CCStudio_v3.1\tms470\docs directory).

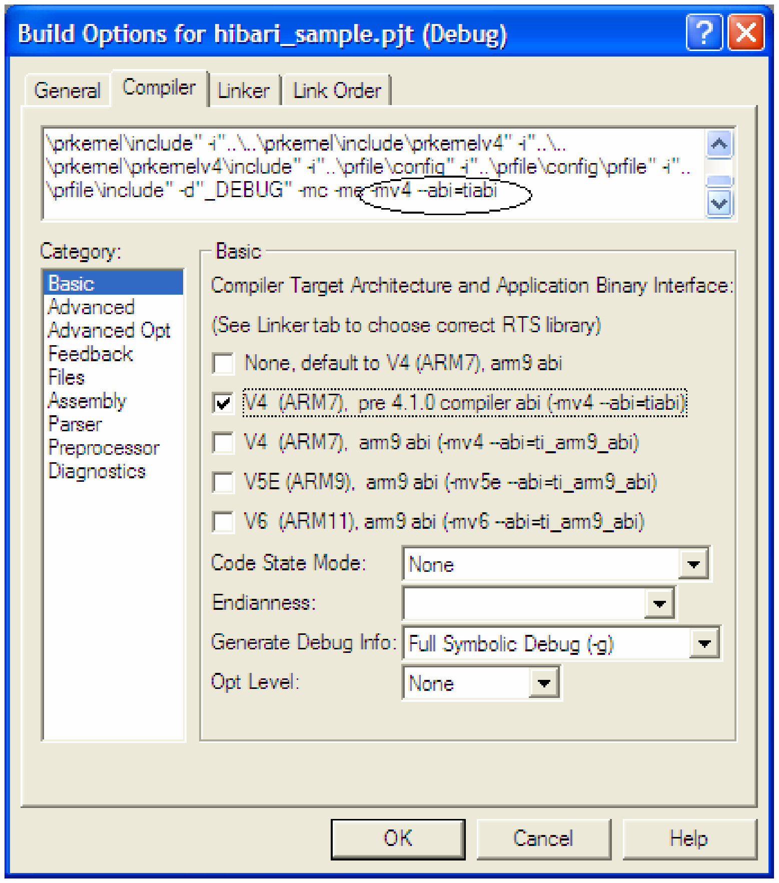
- **Mix of C and assembly code, no old objects.** If the application has a mix of C and assembly code but does not link in object files or libraries built with older tools, change the assembly files to work with the new ABI. To do so, identify all 32-bit and 16-bit routines in assembly source using the ".armfunc" and ".thumbfunc" assembler directives. For example:

  ```
  .thumbfunc _foo
  _foo:
  ```

  The new tools provide a predefined symbol, __TI_ARM9ABI_ASSEMBLER, which supports writing assembly code that works with both the new ARM9 ABI and the older TIABI. The assembly source files associated with this application note contain example functions that can be built with any version of the code generation tools and can run under any ABI.

- **Mix of C code and old object code.** To rebuild the application for TI_ARM9_ABI, first rebuild the object files with the new tools, if possible. Then rebuild the entire application. If it is not possible to rebuild the object files or libraries, build the application with the --abi=tiabi compiler option to maintain backward compatibility. Use a run-time support library that is built for TIABI. Examples include rts32tiabi.lib and rts16tiabi.lib.

  The --abi option is available in the CCStudio 3.1 Build Options dialog as shown in Figure 1.



Build Options for hibari_sample.pjt (Debug)

General | Compiler | Linker | Link Order

```
\prkernel\include" -i"..\..\prkernel\include\prkernelv4" -i"..\..
\prkernel\prkernelv4\include" -i"..\prfile\config" -i"..\prfile\config\prfile" -i"..
\prfile\include" -d"_DEBUG" -mc -me -mv4 --abi=tiabi
```

Category: Basic, Advanced, Advanced Opt, Feedback, Files, Assembly, Parser, Preprocessor, Diagnostics

Basic

Compiler Target Architecture and Application Binary Interface:

(See Linker tab to choose correct RTS library)

- [ ] None, default to V4 (ARM7), arm9 abi
- [✓] V4 (ARM7), pre 4.1.0 compiler abi (-mv4 --abi=tiabi)
- [ ] V4 (ARM7), arm9 abi (-mv4 --abi=ti_arm9_abi)
- [ ] V5E (ARM9), arm9 abi (-mv5e --abi=ti_arm9_abi)
- [ ] V6 (ARM11), arm9 abi (-mv6 --abi=ti_arm9_abi)

Code State Mode:	None
Endianness:	
Generate Debug Info:	Full Symbolic Debug (-g)
Opt Level:	None

OK | Cancel | Help

Figure 1. Build Options Dialog Showing --abi Option

5 Rebuilding Baseline DSP/BIOS Link and Message Application

There are two options for avoiding the linker warnings encountered in Section 3, "Building the Baseline DSP/BIOS Link and Message Application":

- Modify all assembly code for TI_ARM9_ABI compatibility
- Build with --abi=tiabi to maintain backward compatibility

For this baseline project, we rebuilt the DSP/BIOS Link kernel and the Message application with the --abi=tiabi compiler option to maintain backwards compatibility. The run-time library rts32e.lib was replaced with rts32etiabi.lib.

Use the following steps modify the make files for DSP/BIOS Link as well as hibari_sample.pjt in CCStudio to use the --abi switch:

1. Edit DA300.mk in the C:\dsplink\make\PrOS\ directory.
2. Add --abi=tiabi to the standard flags for the compiler, STD_CC_FLAGS.
3. Rebuild the GPP-side DSP/BIOS Link.
4. Rebuild the GPP-side Message application. The resulting library is MESSAGEGPP.LIB.
5. Use CCStudio Setup to select the DA295 EVM.
6. Start CCStudio and open the GPP-side project.
7. Open the hibari_sample.pjt project in the C:\asp-psp\system\hibari\pros\hibari_evm\build directory.
8. Change the project options on hibari_sample.pjt and all dependent projects. Right-click on each .pjt file and select **Project->Build Options**. In the Compiler tab, choose the Basic category. Then, select the V4 (ARM7), pre 4.1.0 compiler abi. Click OK.
9. Remove rts32e.lib from the project.
10. Add rts32etiabi.lib to the project. This file can be found in the C:\CCStudio_v3.1\tms470\cgtools\lib directory.
11. Rebuild the hibari_sample.pjt project.

As you can see in the build output, the linker warnings that appeared earlier are now gone. The output of this build is used as the baseline for code size measurements in this application note.

6 Trampolines

Trampolines are a linker feature that changes how function calls are implemented. The correct sequence to use for a function call depends on how far away in memory the caller is from the callee. Compiler-based solutions for this issue tend to be costly in terms of speed and code size. Trampolines offer a linker-based solution that optimizes function call implementations.

Support for trampolines is available in TMS470 code generation tools v2.17 and higher.

6.1 Impact of Trampolines on DSP/BIOS Link

Using the trampolines feature is not a necessity for the GPP-side DSP/BIOS Link and Message applications. The code size of the application is well within the 22-bit address range supported by near calls for the TMS470. This can be verified by checking the link map file for the sample project, hibari_sample.map. The size of the .text section is 0x00029d14. Also the entire code section is placed in SDRAM, so everything is within near range. However, if the application were to grow larger, or if some code sections were placed in memory that is “far” from other sections that call them, then it would be necessary to build with trampolines enabled.

The next section gives an overview of trampolines and when they are beneficial to use.

6.2 Background on Trampolines

The TMS470 CPU supports PC-relative call and branch instructions whose range is limited to a 22-bit address range. When these instructions are used, the destination address must be near enough to the instruction that the difference between the call and the destination fits within the 22-bit range. If the callee is too far away from the caller, the linker generates an error.

The alternative to a PC-relative call is generally an "absolute" call, which is implemented by loading the callee address into a register, and calling via that register. This approach is more expensive in terms of speed and size.

In the TMS470 code generation tools, a sub-optimal way to solve this problem using the compiler is by using the “far mode” (-ml) shell option which tells the compiler that *all* of the calls generated are "far" calls. This option again uses larger call sequences than necessary since all calls are treated as far calls instead of just the specific ones that need to be, resulting in larger and slower code.

The linker now provides a more optimal solution to this problem with trampolines. With this feature, only those calls that need to be “far” are transformed by the linker. When trampolines are enabled, the compiler always uses the smaller call sequence. The linker then determines whether the call sequence can reach its destination or not. If it cannot, it redirects the call to a linker-generated trampoline. The trampoline code section contains a sequence of instructions that performs a transparent long branch to the original callee. Because the trampoline uses a branch instead of a call, the called function returns not to the trampoline, but to the original call sequence. Multiple calls to the same function may reuse the same trampoline.

Consider the following example, in which a function "bar" calls a function "foo".

The compiler initially generates:

```
bar:
      ...
      call    foo                  ; call the function "foo"
      ...
```

If "foo" is out-of-range from the call to "foo”,   then the linker does the following:

- Change the original call to "foo" into a call to "foo_trampoline":

```
bar:
      ...
      call    foo_trampoline       ; call a trampoline for foo
      ...
```

- Generate a trampoline code section, "foo_trampoline", containing code that executes a long branch to the original callee, "foo":

```
foo_trampoline:
      long_branch     foo
```

Since the trampoline needs to be placed such that it can be reached by a small call sequence, it is usually appended to the end of the output section.

Trampolines are not enabled by default. If the application encounters a situation where "long" calls are required, the linker generates an error, and suggests that the --trampolines option be used.

Please note that the cl470 (v4.1.2) help summary says the following:

```
--trampolines[=off,on]          Generate far call trampolines (Default:on)
```

This seems to imply that trampolines are on by default; however, this is not the case. What this really means is that the option can be --trampolines=on OR --trampolines=off. If you use just "--trampolines" as the option, it turns trampolines on. The trampoline feature itself is not enabled by default in any release of the ARM tools.

6.3 Version Notes on Enabling Trampolines

Table 1 shows which version of the TMS470 code generation tools is included in which version of CCStudio.

Table 1. CCStudio and Code Generation Tools Versions

CCStudio Product Version	TMS470 CGT version
2.20	2.24
2.21	2.25
2.21 + Update Advisor patch for 2.52 CGT	2.52
3.1	4.1.2

- In v2.17 to v2.25 of the TMS470 code generation tools, the equivalent linker option is --large_model. It is not documented in cl470 help summary, but is in the readme.txt file.
- In v2.54 of the TMS470 code generation tools, the linker option is --large_model. It is documented in cl470 help summary.
- In v4.1.2 of the TMS470 code generation tools, CGTLinker.txt refers to the option as --large_model, but the cl470 help summary lists it as --trampolines. Both options should work, but --trampolines is the recommended option moving forward.
- In the Build Options dialog of CCStudio 3.1, there is a check box for enabling trampolines (--trampolines) in the Linker tab Advanced category.

6.4 Trampoline Details in Link Map File

Any trampolines generated by the linker are displayed in the link map file. The statistics include the name and address of the callee, the address of trampoline, and from where the call originated. Please see the SPRAA46 application note at http://focus.ti.com/docs/apps/catalog/resources/appnoteabstract.jhtml?abstractName=spraa46 for more details on the trampoline data provided in the map file.

7 Mixing Arm Mode and Thumb Mode Code

A valid technique for reducing code side for a given ARM library or application is to use the “Thumb mode” feature of the compiler (the -mt flag). This mode uses a reduced instruction set of 16-bits vs. 32-bits for standard “Arm mode”. Thumb mode is optimized for code density from C code, but contains only a subset of the functionality of the ARM instruction set.

By default, the TI ARM compiler supports dual-state interworking. This allows code compiled in both Arm mode (the default) and Thumb mode to work correctly. It is the called function’s responsibility to handle the proper state changes required by the calling function (call it by name). It is also the responsibility of the calling function to handle the proper state change required to indirectly call a function (call it by address). If a function doesn’t support dual-state interworking, it cannot be called by functions requiring a state change and cannot indirectly call functions that support dual-state interworking.

The TI ARM compiler supports (by default) full dual-state interworking. At link time, the linker determines whether to link in the additional code (veneer) that allows functions requiring state changes to call the function. There are other compiler options that can set different support for interworking functions. See the *TMS470R1x Optimizing C/C++ Compiler User's Guide* (SPNU151) for more information regarding this topic.

Before compiling C code for Thumb mode to reduce code size, make sure all functions being called are either Thumb mode functions or support dual-state interworking. When using or writing ARM assembly functions, you need to make sure the assembly code supports dual-mode interworking if the function is expected to be called from the opposite mode.

7.1 Issue Encountered When Mixing Thumb and Arm Modes

To save code size, DSP/BIOS Link was compiled in Thumb mode (with the –mt option). This includes both DSP/BIOS Link library and applications on the GPP-side. The GPP-side (ARM7) operating system (PrOS) was compiled in Arm mode.

The dual-state interworking features of the tools support combining such code that executes in opposite modes. This approach is valid since the compiler and linker should generate and link in the correct veneers to support interworking.

However, we found that by default the PrOS assembly code does not support dual-mode interworking. To correct the problem, we needed to use the pre-defined symbol - dDUAL_STATE_INTERWORKING in order for the assembly interrupt functions to return to the correct processor mode. To make this change to PrOS, follow these steps:

1. Modify the .pjt files in the PrOS directory tree that start with the name “hibari” or “Hibari”.
2. If the .pjt file contains an option like the following, that project may include some assembly code that checks the DUAL_STATE_INTERWORKING symbol.

```
-d"_DUAL_STATE_INTERWORKING=0"
```

3. If you find the previous option in a .pjt file, change it to the following:

```
-d"_DUAL_STATE_INTERWORKING=1"
```

4. Rebuild the entire PrOS application.

7.2 Detecting Thumb and Arm Mode Conflicts

The method used to detect interworking conflicts between DSP/BIOS Link and PrOS is a useful illustration for detecting other conflicts.

In this case, the ARM assembly interrupt functions in PrOS weren't handling processor state changes correctly. All interrupts on the ARM are handled in Arm mode, so the assembly functions didn't need to handle the state change (from Thumb to Arm). However, they did need to ensure that the proper processor state was set before returning. Failure to do this led to the processor attempting to execute Thumb mode instructions in Arm mode, which caused the hardware to issue an undefended instruction exception vector interrupt.

We validated the problem using the following recommended technique for exiting the exception handler for an undefined instruction. The undefined instruction interrupt trap handler was set to execute the following instruction:

```
MOVS PC, R14
```

This instruction restores the Current Program Status Register (CPSR) and returns to the instruction following the undefined instruction. Inspecting the T bit—bit 5, which indicates 16- vs. 32-bit mode—in the CPSR register indicated that the processor was in Arm mode (see Figure 2). T is set to one for 16-bit mode, zero for 32-bit mode. In this case, the instruction following the undefined instruction was a Thumb mode instruction. Therefore the exception was caused due to executing a Thumb mode instruction in Arm mode.

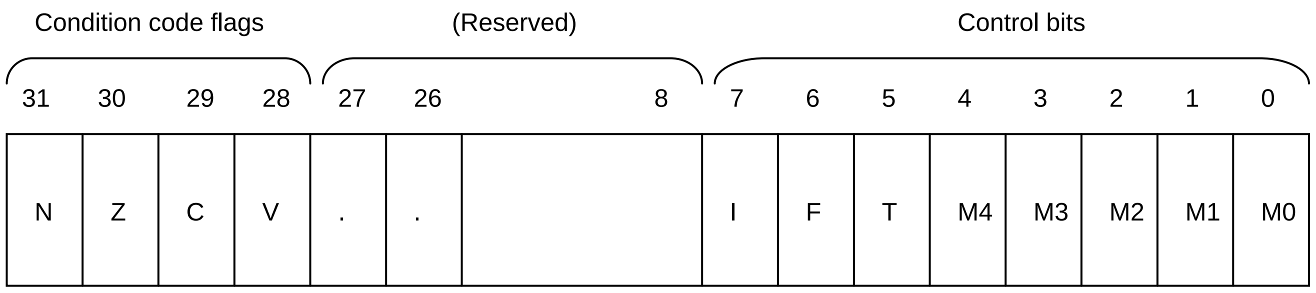


Figure 2. CPSR Register Bits After Undefined Instruction

8 Code Size Data

This section reviews the code size data observed when building DSP/BIOS Link with different sets of compiler options.

Each bar in Figure 3 represents one build. The compiler options for that build are shown in the key to the right. Each build uses the baseline options of –me --abi=tiabi. The –me option indicates little-endian byte ordering. The --abi=tiabi option uses the older TIABI application binary interface. The decision to use TIABI is discussed in Section 4, “ABI Changes in TMS470 Code Generation Tools”. The other options shown in the key are discussed in subsections that follow.

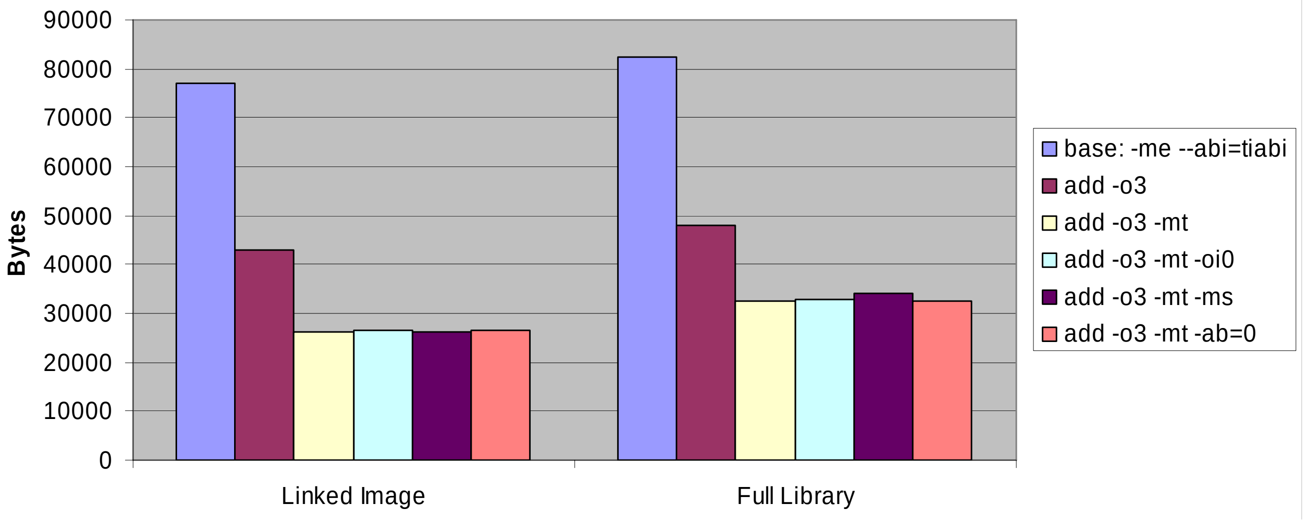


Figure 3. Code Size vs. Compiler Options

The “Linked Image” data set shows the amount of code size, in the final linked image, used by the DSP/BIOS Link functions. The “Full Library” data set shows the code size of the entire DSP/BIOS Link library. The linked image code size is smaller because it only counts the size of Link functions actually called by the Message example. The extra size in the full library is due to functions that are not called in the Message example. The code sizes are close, which indicates that the Message example uses most of the routines supplied by DSP/BIOS Link. Some interesting differences, however, are discussed in the following subsections.

8.1 Collecting the Data

The “Linked Image” data was collected by running a Perl script over an XML version of the linker map file. An XML version of the map file was created using the --xml_link_info=*filename* linker option. The Perl script searched the list of input sections in each output section. Input sections from a library were separated out, and their sizes were accumulated both by library name and by the kind of section (code, data, or uninitialized data). Since the sizes of the data sections were very similar in all the builds, there are not shown here.

The “Full Library” data was collected by running another Perl script over the XML output obtained by running the Object File Display utility on the Link library. The script accumulated the sizes of all the sections in the library, both by file name and by the kind of section. Again, only the sizes of the code sections are shown.

Details on these XML files and writing scripts to glean data from them were the topic of a presentation at the Texas Instruments Developer Conference in Spring 2005. The archived presentations can be found at http://www.ti.com/tidc05pres. The title of this presentation is “Cool New Code Generation Utilities”.

8.2 Optimization Levels: –o3 Option

Five levels of optimization are available with the ARM compiler: none, 0, 1, 2, and 3. The level is indicated by the number following the–o compiler option. Level 3 is the highest optimization level. It lowers code size by about 44% versus the baseline build. We used the –o3 compiler option in all builds other than the baseline build.

8.3 Thumb Mode: –mt Option

Thumb mode is discussed in Section 7, “Mixing Arm Mode and Thumb Mode Code”. You select Thumb mode by using the build option –mt. Building with Thumb mode brings overall code size savings to about 66%.

The following chart shows the differences in the Thumb builds more clearly. The following subsections discuss these differences.

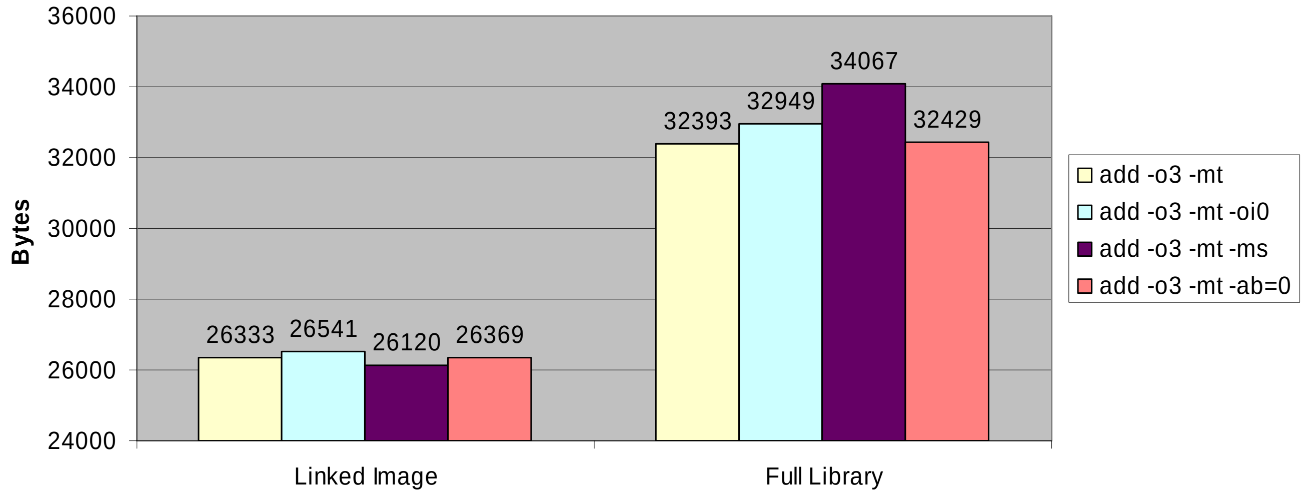


Figure 4. Code Size for Thumb Builds

8.4 Automatic Inlining: –oi0 Option

Inlining a function replaces a call to a function with the actual function in the body of the calling function. Inlining a call speeds performance both by avoiding the overhead of the call and by improving optimization in the code context around the call. This speed increase comes at the expense of increased code size, especially when the inlined function is large, or is called many times.

When you use –o3 to build at optimization level 3, the compiler inlines fairly small functions by default. You can further control the space/speed trade-off of inlining by using the –oi compiler switch, which allows you to specify a number for the inlining threshold.

The compiler internally measures each function in terms of the number of operations it performs. All functions below the threshold of the –oi option are automatically inlined. To see the size of the functions, use the compiler switch –on2 in combination with –o3. This switch creates a text file with a base name the same as the original source file, and the extension .nfo. The compiler-generated information in that file includes the size of the functions.

The compiler switch –oi0 sets the inline threshold to 0, which disables all automatic inlining. It seems that disabling inlining (–oi0) would further reduce code size. However, it results in a code size increase of about 0.8% in the linked image, and about 1.7% in the whole library. Why?

The answer is best described by example. Consider this code:

```
static void call_once(int *ptr1, int *ptr2)
{
    if (!ptr1 || !ptr2)
    {
        /* error handling */
        return;
    }

    /* operations with ptr1 and ptr2 */
}
```

This function is called only once in the file, and the call is similar to:

```
call_once(&local_var1, &local_var2);
```

For functions called only one time, the compiler uses a higher threshold to decide whether to inline. Such is the case with this example function. Once the function is inlined, the call site is optimized in the surrounding context. In this case, the pointers are replaced with the addresses of the local variables. Thus, the test for NULL pointers and subsequent error handling code are all removed.

Inlining a function does not mean the original function can be removed in all cases. If some other source file in the application may call the function, then the original cannot be removed. In the case of the call_once function, the inlined function is declared static. Thus, no other calls may exist and the original call_once function is removed.

In this case, all that remains at the one site where the function was called is:

```
    /* operations with ptr1 and ptr2 */
```

Thus, whether inlining increases or decreases your code size depends upon whether you have a number of static functions called only once and how well they can be optimized when they are inlined.

Other functions cannot be optimized as much after inlining, so the code size savings for these functions is more modest. All that is avoided is the function call and return overhead. Nonetheless, the savings becomes noticeable when added up over several functions, as is the case with the DSP/BIOS Link code.

8.5 Functions in Subsections: –ms Option

By default, the compiler puts the assembly code for each function in the section .text. If you use the –ms compiler option, each function is instead placed in its own subsection with the name .text:*_name_of_function*. These subsections are marked for conditional linking. If the linker does not see any references to a subsection, it is removed. In effect, if a function is never called, it is deleted. Functions that are never called can be a common occurrence when combining code developed at different times for multiple purposes.

If very few functions are removed, however, placing functions in subsections can cause code size to grow. Why?

The ARM instruction set has limited support for constant literals. Constant literals include both integer constants and addresses. When a constant literal is needed, it is placed in a constant table in memory, and then a reference is made to the table entry. This table entry reference is accessed at an offset from the program counter (PC). The size of the offset is limited, which in turn limits the distance in memory between the constant table and the function which references it. Thus, the constant table must be placed in the same section as the function which refers to the constant.

If there are multiple references to the same constant, even from different functions, these references can share one constant table entry, provided the functions are all in the same section with the constant table. When each function is placed in its own subsection, there can be no sharing of constant table entries between functions. References to the same constant result instead in duplicate entries in separate constant tables. Thus, overall code size may grow.

In the case of the DSP/BIOS Link example, the –ms option does reduce the size of the Link library functions in the final link; that is, when examining the Linked Image size data. Comparing the –o3 –mt build with the –o3 –mt –ms build shows a code size savings of about 0.8%.

However, the same comparison for the Full Library size data shows a code size increase due to the –ms option. The size of the library is simply the sum of the all the code in the library. In that sense, there is no notion of removing a function from the library. Since none of the functions are removed, and placing each function in its own subsection causes more duplication in the constant tables, then overall code size increases.

8.6 Branch Chaining: –ab=0 Option

Branch chaining is a code transformation aimed at improving code size, though at the cost of increased cycles. Consider the following assembly pseudo-code:

```
               long_branch   label
               ...
               short_branch  label
               ...
label:
```

Branch chaining changes the code to:

```
               short_branch  next_branch
               ...
next_branch:
               short_branch  label
               ...
label:
```

The second sequence is smaller, but takes longer. On ARM devices such a modification can save 2-4 bytes of code size.

A key property of branch chaining is the length of the chain. The length is the number of branches that may be chained together to implement the original long branch.

Branch chaining is enabled by default when building Thumb mode code. The default length is 10. Thus, branch chaining up to a length of 10 is enabled in all builds compared here except the one that disables it with the –ab=0 option. The option to change the length is –ab=*number*. Another way to disable branch chaining is to use the --disable_branch_chaining option.

Branch chaining is never enabled when building Arm mode code.

Assembly coders can automatically create branch chains by using special pseudo-branch instructions in place of the usual branch instructions. Details are in the readme file that accompanies the ARM compiler release, and will eventually be covered in the documentation.

Branch chaining saves about 0.14% code size in the DSP/BIOS Link example application. So, the –o3 –mt –ab=0 build is slightly larger than the –o3 –mt build since branch chaining is disabled.

Even though it provides minimal savings in code size, it rarely makes sense to disable branch chaining. Branch chaining is only implemented in Thumb mode code, which is rarely speed-critical. Any speed-critical Thumb code, however, could save a few cycles by disabling branch chaining.

9 Performance Data

This section reviews the performance impact of the compiler options described in the previous section.

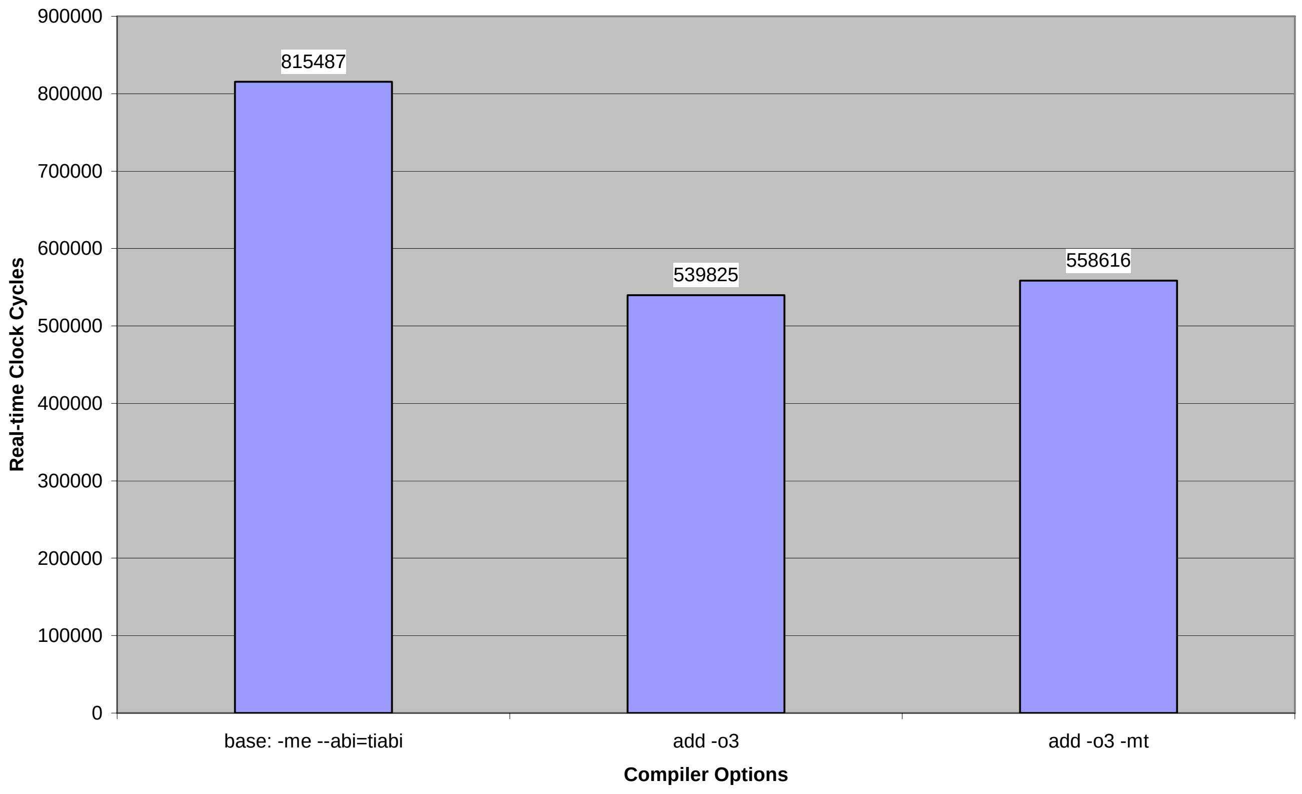


Figure 5. Performance Impact of Compiler Options

Each bar in Figure 5 represents the cycles used (in Real-Time Clock cycles) to execute the example with different build options. Smaller values indicate better performance. Options described in Section 8, “Code Size Data” that are not shown in Figure 5 caused negligible (<1%) performance impacts and therefore are not shown here.

9.1 Data Collection

The Probe module in DSP/BIOS Link (ARM side configuration option) was enabled to gather the performance data. The Probe module stores timestamps at pre-determined locations within the normal flow of execution. The timestamp data is then analyzed offline to determine performance.

The data in Figure 5 was gathered using the standard message example and placing appropriate Probe API calls within the example. The data gathered represent the cycles required to send and receive 10 messages from the ARM to the DSP and back.

9.2 Analysis

Optimization in the ARM compiler not only lowers code size by about 44% versus the baseline build. It also increases performance by 33%. Both performance and code size benefit from optimization build options.

Adding the Thumb mode option (-mt) causes a slight increase in cycle execution. Figure 5 shows that there is a 3% increase in cycle execution when using Thumb mode. Why?

If every Arm mode (32-bit) instruction had an equivalent Thumb mode (16-bit) instruction, code size should decrease by 50% without impacting performance. The reality is that the Thumb mode instruction provides a subset (arithmetic and logical operations, load/store data movements, and conditional and unconditional branches) of the functionality provided by the Arm mode instruction set. Hence, you can achieve good code size reduction using Thumb mode code. This subset of Thumb mode instructions provides most of the functionality required in a typical application. Therefore, a slight performance penalty is paid since, for a given program, a few of the Thumb mode instructions executed is more than the corresponding number of Arm mode instructions executed.

The other options described in Section 8, “Code Size Data” caused negligible (<1%) performance impacts and therefore not shown in Figure 5. The options, especially function subsections (-ms) and branch-chaining (-–ab=number), typically have an impact of 10s of cycles depending on the placement of functions and the depth of function calls. As for inlining, performance can be impacted by the removal of function calls and improvements in optimization when inlining occurs.

10 Conclusions

The techniques described in this application note are general and can be applied to any TI ARM development platform. Not every option described is suitable for all applications. You will need to determine the best set of options to accomplish both the speed and performance necessary for their application and libraries. In some case, you will need to obtain performance data in order to determine the direct effect of these options.

11 References

1. *TMS470R1x User’s Guide* (SPNU134)
2. *TMS470R1x Assembly Language Tools User’s Guide* (SPNU118)
3. *TMS470R1x Optimizing C/C++ Compiler User's Guide* (SPNU151)
4. *DSP/BIOS Link User’s Guide for DA295* (LNK 093 USR)