TEXAS INSTRUMENTS

Application Report
SPRAA46A–December 2008

Advanced Linker Techniques for Convenient and Efficient Memory Usage

George Mock *Software Development Systems*

ABSTRACT

The linker is the code generation development tool responsible for linking together all the object files and libraries into the final executable form. The linker offers many features, including some recent additions, which make it easy to use system memory efficiently. This application note gives practical advice on how to use three of these features: automatic section splitting, copy tables, and trampolines. Automatic section splitting distributes code or data across separate memory ranges. Copy tables are a convenient way to manage code or data overlays at runtime. Trampolines change how function calls are implemented such that far call memory models are no longer necessary. Detailed examples show all of these linker features in action.

Project collateral and source code discussed in this application report can be downloaded from the following URL: https://www.ti.com/lit/zip/spraa46a.

Contents

1 Introduction 2
2 Linker Basics 3
3 Automatic Section Splitting 4
4 Copy Tables 7
5 Example of Automatic Section Splitting and Copy Tables 10
6 Trampolines 14
7 Feature Availability 18
8 Summary 19
9 Acknowledgements 19
Appendix A Placing Functions in Subsections Can Cause Code Growth 20
Appendix B Trampolines Improve Performance on Reference Frameworks Example 22

List of Figures

1 Overview of Linking 3
2 Automatic Section Splitting 5
3 Section Splitting and Copying at Runtime 11
4 Trampolines 15
5 Enabling Trampolines 17

List of Tables

1 Assembly Language Tools User's Guides 3
2 Put Functions in Subsections 7
3 Feature Availability 18
4 Code Generation Tools Versions by CCS Version 18

Copyright © 2008, Texas Instruments Incorporated

1 Introduction

It is tempting to view the linker as just a necessary utility; the tool that links all the object files and libraries together into the final executable. In fact, the linker contains a number of features which make it easy to use memory efficiently. This application note gives practical advice on using three of these features.

Automatic section splitting makes it easy to distribute code or data across multiple memory ranges. These memory ranges do not have to be adjacent. The algorithm that distributes the pieces of code or data among the memory ranges focuses entirely on saving memory.

Copy tables are a feature which makes it very straightforward to manage code or data overlays. Overlays are used to share a relatively small block of fast memory among multiple pieces of code or data. The code or data is first loaded into slow memory, then as required at runtime, copied into the fast memory block for execution. Before copy tables, managing the details of the runtime copy was cumbersome. With copy tables, managing the overlays is much easier.

Trampolines are a feature that changes how function calls are implemented. Some CPU architectures use different code sequences to implement function calls. The correct sequence to use depends on the distance in memory between the function call and the destination of the call. Without trampolines, managing this detail is either error prone, or tends to waste memory and cycles. Trampolines automatically implement a near perfect solution to this problem. In common cases, switching from various large code model build options to using trampolines results in notable performance improvement. Appendix B walks through an example based on Reference Frameworks Level 3 in which trampolines improve performance by almost 10%!

A detailed example binds all the practical advice together. This example is presented in the form of a Code Composer Studio™ project. You will see how the project is built, the details of what linker did, and watch the example code execute.

Code Composer Studio is a trademark of Texas Instruments.
All other trademarks are the property of their respective owners.

Copyright © 2008, Texas Instruments Incorporated

2 Linker Basics

Figure 1 illustrates how the linker combines (links) several object files and libraries together.

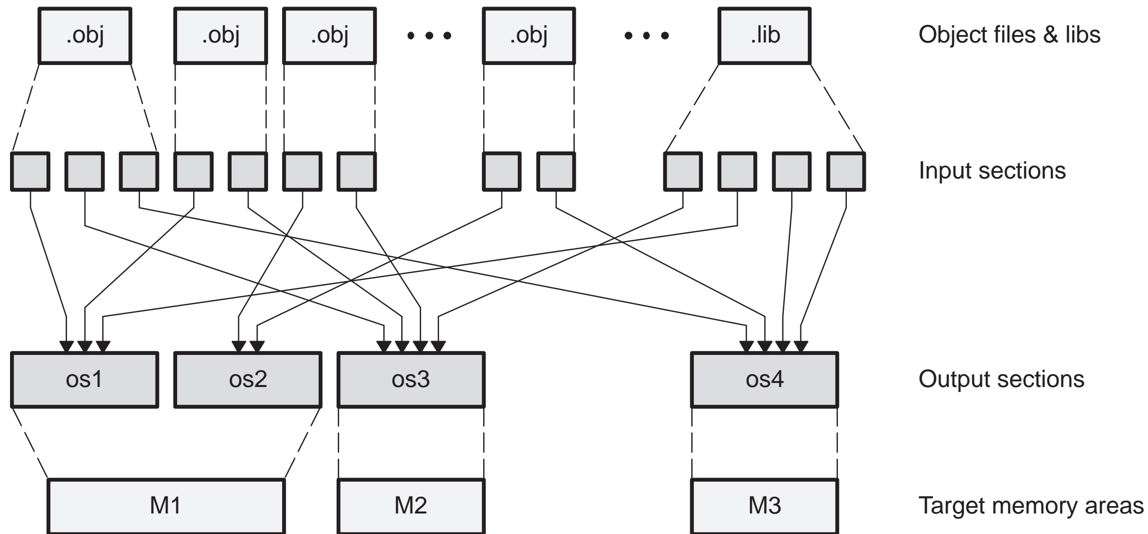

Figure 1. Overview of Linking

Each object file is viewed as a collection of input sections. The same is true for each member object file brought in from an object library. The input sections are combined together into output sections. Next, the output sections are allocated to target memory. The final output object file can also be viewed as a collection of sections. These sections, however, are different in two ways: 1) They are output sections, not input sections, and 2) These sections are allocated to target memory.

The basics of using the linker are described in the Linker Description chapter of the Assembly Language Tools Guide. There is one such guide for each TI processor family. Table 1 lists the User's Guides for the TI processors most commonly used today.

Table 1. Assembly Language Tools User's Guides

Title	Literature Number
TMS320C6000 Assembly Language Tools User's Guide	SPRU186
TMS470R1x Assembly Language Tools User's Guide[(1)]	SPNU118
TMS320C55x Assembly Language Tools User's Guide	SPRU280
TMS320C54x Assembly Language Tools User's Guide	SPRU102
TMS320C28x Assembly Language Tools User's Guide	SPRU513

(1) The guide applies to both ARM7 and ARM9.

Users of ARM7 and ARM9 devices should refer to the TMS470R1x User's Guide.

Copyright © 2008, Texas Instruments Incorporated

The remainder of this application note presumes the following basic knowledge, as described in the documentation.

- The MEMORY directive assigns names to ranges of target memory
- The SECTIONS directive
  - Collects input sections into output sections
  - Allocates output sections to memory ranges as defined in the MEMORY directive
- The MEMORY and SECTIONS directives must be defined in a linker command file
- The specific syntax of the MEMORY and SECTIONS directives

To access any of these User's Guides online, visit the TI main web page http://www.ti.com and enter the literature number in the Keyword Search text box. The appropriate User's Guide is also bundled with Code Composer Studio™. To see it select Help | User Manuals. The online help in Code Composer Studio™ also documents the linker. Select Help | Contents | Code Generation Tools | Using the Linker.

3 Automatic Section Splitting

3.1 Problem Description

The C/C++ compiler places all code in the section .text by default. The simplest way to place such code in memory is to create a single output section, also named .text, and allocate it to an appropriate area in target memory. It might look like this in the SECTIONS directive …

```
.text > M1
```

This statement instructs the linker to collect all of the input sections with the name .text into an output section also named *.text* , and allocate the *.text* output section into the memory range M1. Such a statement requires M1 to be large enough to contain *.text*.

Suppose M1 is not large enough to contain *.text*. But there are other memory ranges M2 and M3 that, when combined together with M1, are large enough to hold .text. Suppose further that M1, M2, and M3 are not contiguous. What is the best way to allocate the .text section into the memory ranges M1, M2, and M3?

3.2 Overview

The simplest solution is to use the linker feature for automatically splitting sections.

```
.text >> M1|M2|M3 /* splitting */
.no_split > M4 /* not splitting */
```

This statement instructs the linker to create the output section *.text* as before. Note the split operator >> is different from the usual memory placement operator >. The output section *.text* is split into pieces that are allocated into the different memory ranges M1, M2, and M3. See Figure 2.

Copyright © 2008, Texas Instruments Incorporated

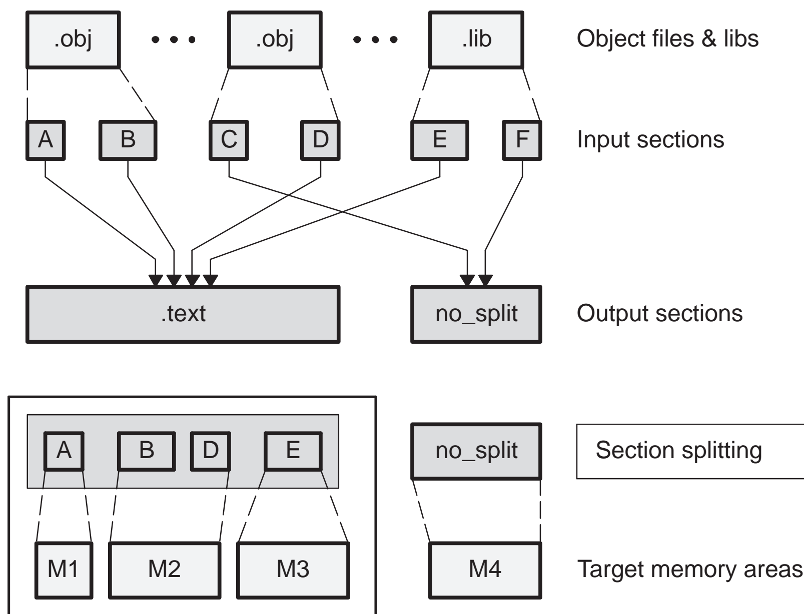

Figure 2. Automatic Section Splitting

The example in Figure 2 builds two output sections. The input sections A, B D, and E are combined into the output section *.text*. The *.text* section is split as it is allocated to the memory ranges M1, M2, and M3. The input sections C and F are combined into the output section no_split, which is not split as it is allocated in to memory range M4.

Section splitting always takes place on input section boundaries. For example, an input section which contains the code for a function does not get split. In general, the linker does not have enough information about an input section to split it.

It is worth noting the difference between the >> operator and the > operator. The following example, while legal, is **not** section splitting.

```
.text > M1|M2|M3
```

By using > instead of >>, the *.text* section is not split. Instead, .text is allocated to the first memory range, M1, M2, or M3 (in order) which can completely contain it. This is called allocation to multiple memory ranges.

3.3 Splitting Data Sections

Section splitting is not limited to sections containing code, such as *.text*. Sections containing data may be split as well. Be especially careful, however, when splitting data sections. Sometimes, code presumes data is contained in a single contiguous block. Such data cannot be split.

Consider, as one example, the *.bss* section in the C6000 C/C++ compiler tools. Global variables are defined in the *.bss* section. The register *DP (B14)* points to the beginning of the .bss section, and global variables are accessed at an offset from the *DP*. The *.bss* section, then, cannot be split. If it were split, some DP+offset global variable accesses would work, and others would not.

Any attempt to split a compiler generated data section which does not allow splitting causes the linker to issue a warning and ignore the split operator. The C6000 .bss section is such a section. No such warning is issued when other **user defined** data sections are split.

Copyright © 2008, Texas Instruments Incorporated

3.4 Limitations on Splitting

At the time of this writing, the linker documentation which lists limitations on using the split feature has some errors. These limitations are listed at the end of a section titled **Automatic Splitting of Output Sections Among Non-Contiguous Memory Ranges**. A complete and correct list of which sections may not be split is as follows:

- Certain sections created by the C/C++ compiler, including:
  - *.cinit,* which contains the autoinitialization table for C/C++ programs
  - *.pinit*, which contains the list of global constructors for C++ programs

  Other sections created by the C/C++ compiler may not be split. Exactly which data sections cannot be split varies by compiler. For instance, the C6000 compiler does not allow the .bss section, which defines global variables, to be split.
- An output section with an input section specification that includes an expression to be evaluated. The expression may define a symbol that is used in the program to manage the output section at run time.
- An output section that has a START(), END(), or SIZE() operator applied to it. These operators provide information about a section's load or run address and size. Splitting the section may compromise the integrity of the operation.
- An output section that is a GROUP member. The intent of a GROUP directive is to force contiguous allocation of GROUP member output sections.
- The run allocation of a UNION. Note splitting the load allocation of a UNION member is allowed.

If you use the >> operator on any of these sections, the linker issues a warning and ignores the operator.

The bullets listed above comprise complete documentation on the limitations of section splitting. Therefore, it mentions situations not described elsewhere in this application note.

3.5 Order of Section Splitting

To further illustrate how automatic section splitting uses memory efficiently, this section describes the algorithm for how the input sections are allocated across multiple memory ranges.

Finding the optimal allocation of input sections across the different memory ranges is a difficult problem. In computer science literature, it is called the *bin packing* problem. The linker handles this problem with a relatively simple solution that takes little time to compute, and produces a good, often optimal, allocation for cases that tend to occur in practice.

The input sections are sorted in this order:

- Aligned sections by size
- Blocked sections by size
- Remaining sections by size

Aligned and blocked sections both relate to allocating the section with reference to an address that is divisible by a given power of 2. An aligned section is required to start on such an address. A blocked section is required to either not cross such an address, or, start on such an address. Aligned sections are more restricted in their placement than blocked sections.

The memory ranges are considered in the order given in the link command statement. A list of available segments is maintained for each memory range. Note that allocating input sections with differing requirements of alignment and size can create multiple available segments within a single memory range.

Allocation proceeds by available memory segments. Each available segment is filled, from the sorted input sections list, until it is full or no more input sections can fit. Then the next available memory segment is considered. In general, allocating larger aligned sections early tends to create holes that smaller unaligned sections may fill later.

Note future releases of the linker may make minor changes to this algorithm aimed at even more efficient use of memory.

Copyright © 2008, Texas Instruments Incorporated

3.6 Use Subsections to Save Memory

All other things being equal, several small input sections tend to get a tighter allocation than a few large input sections, and thus use less memory.

For C/C++ code, there is an easy automated way to separate the code into smaller input sections. Some compilers have a build switch that places each function in a subsection. The default section for code is

```
.text
```

The functions in subsections build switch causes each function to be placed in a subsection named

```
.text:name of function
```

The compilers which support this switch are listed in Table 2.

Table 2. Put Functions in Subsections

Compiler	Build Switch	Alias
TMS320C6000	--gen_func_subsections	-mo
ARM	--gen_func_subsections	-ms
TMS320C55x	--gen_func_subsections	-mo

Subsections are described in detail in Chapter 2 of the Assembly Language Tools User's Guides. The main effect on automatic section splitting is that the linker sees input sections that are much smaller than otherwise. Without the functions in subsections switch, all the functions in a source file are combined into a single input section. With the functions in subsections switch, each function in the application is placed in its own input section. In most cases, changing to many smaller input sections results in a tighter allocation, thus using less memory.

One downside of using the functions in subsections switch is that it can cause code size to increase. Details of this code growth are described in Appendix A. In some cases, the code growth due to using the functions in subsections switch is more than the memory saved by a tighter allocation. Whether this technique saves memory in any particular system can only be determined by direct experimentation.

4 Copy Tables

4.1 Problem Description

Rare is the memory system where all the memory is as fast as desired. Most memory configurations contain a relatively small block of fast memory. Achieving good performance demands efficient use of this fast memory.

Some systems supply a cache for this purpose. A cache is a small fast memory holding recently accessed code or data, designed to speed up subsequent access to the same code or data. The contents of the cache are automatically controlled by hardware. What if the system does not have a cache? Or, more direct control over the contents of fast memory is needed?

Such systems can be addressed by another efficiency technique which shares the fast memory among several chunks of code or data. Code or data is loaded in slower memory, and then copied into the fast memory block as needed for execution. This technique is called overlaying sections. For a general overview of overlaying sections, see the section titled **Run-Time Relocation** in chapter 2 of the Assembly Language Tools User's Guide.

Copyright © 2008, Texas Instruments Incorporated

Two steps are required to overlay sections:

1. The linker allocates the sections to a single memory range for running, but different memory ranges for loading.
2. The application, at runtime, manages the process of copying a section, as needed, from the load memory range to the run memory range.

   Implement step 1 with the UNION directive. Details are in the section titled **Overlaying Sections with the UNION Statement** in the linker chapter of the Assembly Language Tools User's Guide.

Implementing step 2 requires three pieces of information about each section in the overlay:

- Load address
- Run address
- Length

The assembler and linker provide support for managing this information in various forms. Such support includes the .label directive, and the linker operators *START( )* , *END( )* , and *SIZE( )* . While useful, these constructs impose a large responsibility upon the user. Further, some operations, such as splitting the load allocation over multiple memory ranges, are not even possible.

Copy tables are a new and improved method for managing overlays. They are introduced in the linker bundled with Code Composer Studio™ version 3.00.

4.2 Overview

There are two elements to copy tables:

- The *table* directive in the linker command file creates the copy tables
- The function copy_in performs the copy at runtime, using information in the linker created copy tables. The function *copy_in* is a runtime support function supplied with the compiler.

4.2.1 The Table Directive

The table directive instructs the linker to create a copy table for an output section. For example:

```
SECTIONS{
...UNION {
.func1 {func1.obj (.text) } load = SLOW_MEM, table (_func1_copy_table)
.func2 {func2.obj (.text) } load = SLOW_MEM, table (_func2_copy_table)
} run = FAST_MEM
.ovly > SLOW_MEM /* allocate copy tables */
}
```

Each instance of the table directive instructs the linker to create a copy table that contains the information necessary for copying the func1 or func2 code from SLOW_MEM to FAST_MEM. The symbol supplied in the table directive is assigned the address of the table. Each copy table is allocated space in the subsection .ovly:symbol name. Note this statement:

```
.ovly > SLOW_MEM /* allocate copy tables */
```

collects all the copy table input sections into an output section named .ovly and allocates it space in SLOW_MEM.

If it were written in assembly, the copy table for func1 would be similar to:

```
 .sect ".ovly:_func1_copy_table" ;name the section
 .global _func1_copy_table ; name of table is global
 _func1_copy_table: ;field sizes specific to C6000
 .short 12 ; size of one copy table record
 .short 1 ; how many copy table records
 .word load address
 .word run address ; linker supplies these values
 .word length
```


Copyright © 2008, Texas Instruments Incorporated

4.2.2 The Function copy_in

The function copy_in is one of the compiler runtime-support functions. The argument to copy_in is the address of a linker created copy table. It performs 1 or more memory to memory block copies as set out in the table, and returns no result. Here is an example based on the previous linker command file:

```
#include <cpy_tbl.h>
void func1(), func2();
extern far COPY_TABLE func1_copy_table, func2_copy_table; /* may not need "far" */
void main()
{ copy_in(&func1_copy_table);
func1();
copy_in(&func2_copy_table);
func2(); }
```

Note this statement below which includes the system header file that declares the *COPY_TABLE* structure and the `copy_in` function.

```
#include <cpy_tbl.h>
```

This statement:

```
extern far COPY_TABLE func1_copy_table, func2_copy_table; /* may not need "far" */
```

declares the copy tables. These are declarations of structures that are defined by use of the table() directive in the link command file. Note the copy table symbols, when written in C/C++, are not prepended with an underscore. This conforms to the convention for naming global objects that are referenced in C/C++. Such symbols are written in assembly with a prepending underscore and in C/C++ without the underscore. Use of the far keyword depends on which compiler is being used. Not all TI compilers support a far data model, also known by terms such as large data model. Even among compilers that support far, it does not necessarily follow that the copy tables reside in far memory. Consult the relevant Compiler User's Guide for details on the far data model.

These statements:

```
copy_in(&func1_copy_table);
func1();
```

call the copy_in function to copy the func1 code from SLOW_MEM to FAST_MEM, then call func1. Recall the argument to copy_in is the address of a copy table. That is why &func1_copy_table is used instead of simply func1_copy_table. Note the func1 code must be copied from the load address to the run address before calling func1. Management of this detail is left entirely to the application. There is no error checking by the compiler or linker to insure a given address has been copied over with the correct code or data prior to execution at that address.

No method exists for executing func1 from SLOW_MEM. All of the symbols associated with func1, including the function entry point, are relocated on the presumption that func1 will execute from FAST_MEM.

To see the source code for the definition of the COPY_TABLE structure or the copy_in function, extract it from the source library for all the compiler runtime support functions. Execute the following at a command prompt:

```
ar6x -x rts.src cpy_tbl.h cpy_tbl.c C6000 Specific
```

This example uses the C6000 archiver utility ar6x. Use the archiver utility in the code generation toolset of interest. The file rts.src is a source library of all the runtime support functions bundled with the compiler. It can be found in the *\lib* directory of the compiler distribution. In Code Composer Studio™ releases, it can be found in the directory *install base\device name\cgtools\lib.*

4.3 Alternative Methods of Performing Copy

The table directive does not require that the RTS function copy_in be used to perform the runtime copy operation. Other methods, such as DMA transfers, may be used as well.

Copyright © 2008, Texas Instruments Incorporated

4.4 Function copy_in Does Not Work for Some C6000 Devices

All C6000 devices are based on a Harvard architecture, consisting of a path from the DSP's program fetch pipeline to a dedicated program memory (or in some cases, a program cache) and a path from the DSP's data load/store pipeline to a dedicated data memory (or in some cases, a data cache). Lower levels of memory may include an L2 SRAM/Cache, L3 on chip SRAM, or external memory accessible via the External Memory Interface (EMIF).

For devices with mapped Program Memory (including C620[1-5] and C6701), the program memory is not directly accessible via Load/Store instructions, since these instructions are serviced via the data load/store pipeline which is connected to data memory/cache. In this case, the function copy_in cannot be used to relocate code to Program Memory. For these devices, DMA transfers must be used to copy blocks of code. However, if code is relocated to locations other than Program Memory, the copy_in function can be used.

For devices whose Program Memory (including C621x/C671x/C64x) always operates as a cache, the lower level memory regions (including L2 SRAM or EMIF address range) are accessible via Load/Store instructions, since the Data Memory controller and Program Memory controller have equal access to these lower levels of memory. For these devices, the copy_in function can always be used.

For more information on internal program memory configurations, consult the *TMS320C6000 DSP Peripherals Overview* (SPRU190), chapter 16.

4.5 Overlays and Cache

Cache can have an effect on code or data that is copied from one memory block to another. All the details of such effects are beyond the scope of this application note. This section outlines some of the potential problems.

Cache for memory that contains code, also known as program cache, generally does not account for the possibility that code stored in memory can change as the application executes. Yet that is exactly what occurs with code overlays. For a CPU with a program cache, any block copy for a code overlay should be accompanied by commands that inform the cache about the change to program memory. If special steps are not taken, calling a function just copied in may result in execution of whatever remains in the cache from before.

Cache for memory that contains data is not handled the same way. If the CPU itself performs the overlay memory block transfer, such as when calling copy_in, then there are no cache problems. The memory block transfer is no different than any other CPU operation which modifies memory. If some other mechanism, such as a direct memory access (DMA) peripheral, performs the transfer, then the transfer operation should be accompanied by commands to inform the cache about the change to data memory.

4.6 Use Copy Tables for Boot Time Initialization

To support loading code from non-volatile memory such as flash or ROM into RAM at boot time, the table directive supplies a special operand BINIT. For example:

```
.bcode: load = ROM, run= RAM, table(BINIT)
```

For more information see the application note *Creating a Second-Level Bootloader for FLASH Bootloading on TMS320C6000 Platform with Code Composer Studio™ 2.2* (SPRA999A). While that application note is specific to C6000, users of other devices will still benefit from the illustration of when and how to use the BINIT form of the table directive.

5 Example of Automatic Section Splitting and Copy Tables

5.1 Overview

The example uses linker copy tables, which are introduced in Code Composer Studio™ version 3.00. Thus, the example can only be run under Code Composer Studio™ version 3.00 or greater. The example was developed using the C6416 Device Cycle Accurate Simulator as the execution platform. Running the example on a different device or execution platform requires changes.

Copyright © 2008, Texas Instruments Incorporated

The application report is accompanied by a zip file that can be downloaded from the following URL: http://www-s.ti.com/sc/techlit/spraa46.zip. Extract the contents of the zip file into an empty directory. Two directories and one file are created. This example uses the contents of the directory *linker_example*. The file *DSK6713.gel* and directory *referenceframeworks* are used in the example in Appendix B.

The files in the linker_example directory are:

- *demo.c* – Code for main and functions 1-3
- *demo.cmd* – Linker command file
- *c6416_cache.c* – Contains a routine for handling details of C6416 program cache behavior in connection with code overlays
- *func4.c* – Code for function 4
- *func5.c* – Code for function 5
- *func6.c* – Code for function 6
- *func7.c* – Code for function 7
- *linker_example.pjt* – Code Composer Studio™ project file

Focus on how the code for functions 1-7 is linked. There are examples of both section splitting and copy tables, sometimes in combination. Figure 3 is a visual summary. Most of what is displayed in Figure 3 is implemented in the linker command file *demo.cmd*.

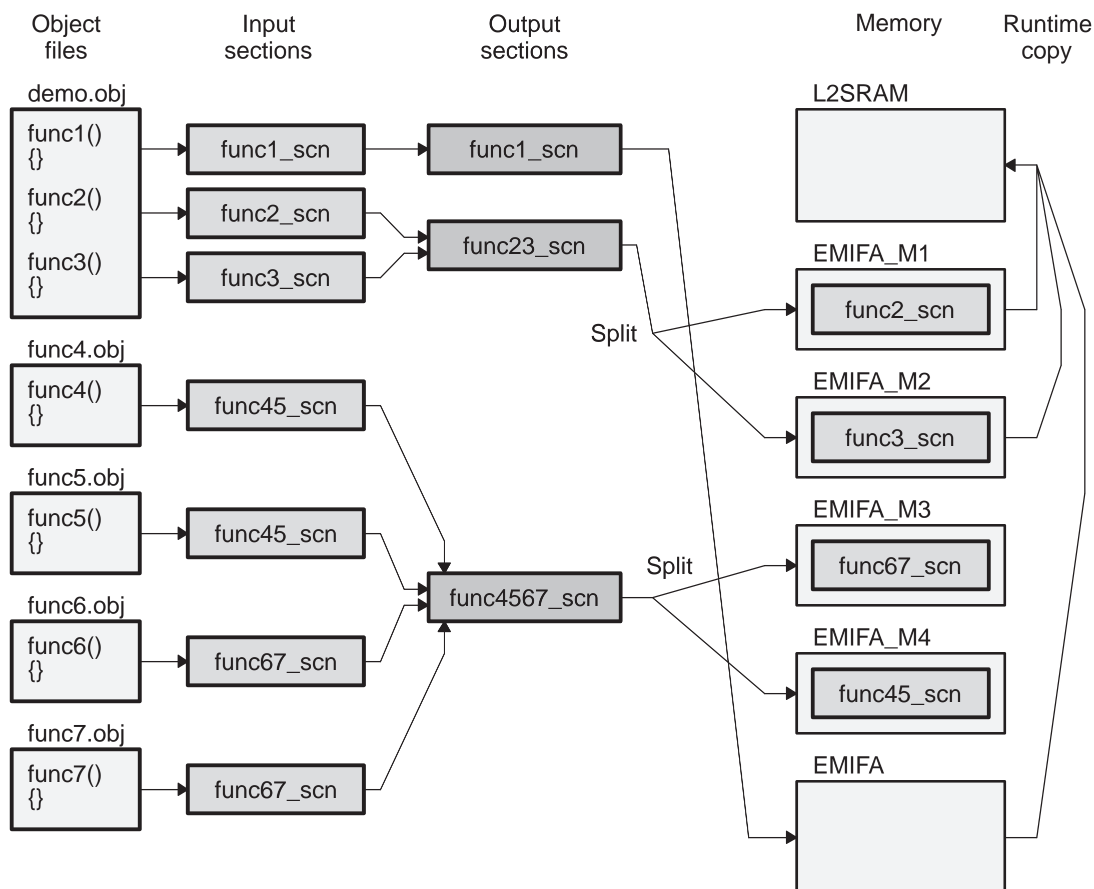

Figure 3. Section Splitting and Copying at Runtime

Copyright © 2008, Texas Instruments Incorporated

5.2 Memory Configuration Details

On chip memory of the C6416 is a block of SRAM called L2. The term L2 comes from the fact all or part of the SRAM can be configured as level 2 cache. The example uses the default configuration, which has none of the block used as L2 cache. The external memory on the C6416 is accessed through an external memory interface (EMIF). The example uses the first of such memory interfaces EMIFA. The first part of EMIFA external memory is broken into artificially small blocks in order to illustrate section splitting.

5.3 Function by Function Description

Here is what happens to the code associated with each function as written in the files *demo.c* and *func[4-7].c*.

The code for function 1 is placed in the input section *func1_scn*. That is linked into the output section also named *func1_scn*. That output section is linked for loading into the general EMIFA external memory. At runtime, this code is copied into L2 SRAM for execution.

The code for function 2 is placed into the input section *func2_scn*. The code for function 3 is placed into the input section *func3_scn*. These two input sections are linked into the output section *func23_scn*. The load allocation of the output section is split across the memory ranges EMIFA_M1 and EMIFA_M2. At runtime, the functions are copied together into L2 SRAM for execution.

Note function 1 and functions 2-3 are executed from the same address range in L2SRAM.

The code for functions 4 and 5 is placed in the input section *func45_scn*. The code for functions 6 and 7 is placed in the input section *func67_scn*. Those two input sections are linked into the output section *func4567_scn*. The output section is split across the memory ranges EMIFA_M3 and EMIFA_M4. These functions are executed from external memory. They are not copied to on-chip memory for execution.

5.4 Build and Run

1. Configure Code Composer Studio™ to use the **C6416 Device Cycle Accurate Simulator** as the execution platform.
2. Start Code Composer Studio™
3. Open the project. Select **Project | Open**, browse to the*linker_example* directory, highlight the file *linker_example.pjt*, then click **Open**.
4. Build the project. Select **Project | Rebuild All** or click the icon.
5. Load the program. Select **File | Load Program**. Browse to the Debug sub-directory of *linker_example*, select the file name *linker_example.out*, then click **Open**.
6. Run the example. Select **Debug | Run** or hit F5. Output displays in the **Stdout** tab of the **Output** window.

Correct output is as follows:

```
hit func1, run address is ed80, x is 0
hit func2, run address is ed80, x is 1
hit func3, run address is ede0, x is 3
hit func4, x is 6
hit func5, x is 10
hit func6, x is 15
hit func7, x is 21
linker example PASSED!
```

Each function prints a message when it executes, and also increments the global variable x by the function number. Note how functions 1 and 2 run at the same address. Functions 4-7 do not print a run address since they are not part of the code overlay in on chip memory.

Copyright © 2008, Texas Instruments Incorporated

5.5 Inspect the Map File

The linker option –m *name of file* creates a linker map file. The map file displays, in text tabular form, information about the executable output file created by the linker. To see the map file select **File | Open**, set the **Files of Type** drop-down box to **Memory Map Files (*.map)**, browse to the *Debug* sub-directory of *linker_example*, select the file *linker_example.map*, and then click **Open**.

Here are excerpts from the map file which relates to code for functions 1-7.

```
22 SECTION ALLOCATION MAP
23
24 output attributes/
25 section page origin length input sections
26 ------- ---- ------ ------- ---------------
27 .func23_scn:1
28 * 0 80000000 00000060 RUN ADDR = 0000ed80
29 80000000 00000060 demo.obj (.func2_scn)
30
31 .func23_scn:2
32 * 0 80000100 00000060 RUN ADDR = 0000ede0
33 80000100 00000060 demo.obj (.func3_scn)
34
35 .func4567_scn:1
36 * 0 80000200 000000c0
37 80000200 00000060 func7.obj (.func67_scn)
38 80000260 00000060 func6.obj (.func67_scn)
39
40 .func4567_scn:2
41 * 0 80000400 000000c0
42 80000400 00000060 func4.obj (.func45_scn)
43 80000460 00000060 func5.obj (.func45_scn)
...
60 .func1_scn
61 * 0 80001300 00000060 RUN ADDR = 0000ed80
62 80001300 00000060 demo.obj (.func1_scn)
63
64 .ovly 0 80001360 0000002c
65 80001360 0000001c <linker> (.ovly:func23_ctbl) [fill =0]
66 8000137c 00000010 <linker> (.ovly:func1_ctbl) [fill =0]
...
221 LINKER GENERATED COPY TABLES
222
223 _func1_ctbl @ 8000137c records: 1, size/record: 12, table size: 16
224 .func1_scn: copy 96 bytes from load addr=80001300 to run addr=0000ed80
225
226 _func23_ctbl @ 80001360 records: 2, size/record: 12, table size: 28
227 .func23_scn:1: copy 96 bytes from load addr=80000000 to run addr=0000ed80
228 .func23_scn:2: copy 96 bytes from load addr=80000100 to run addr=0000ed80
```

Note the line numbers on the left are not displayed in the Code Composer Studio™ editor.

Lines 27 and 31, as well as lines 35 and 40, illustrate the representation of an output section that is split. The form is name of *output section:number*. Split sections are numbered individually, starting with 1. The last column shows which input sections are placed in a given split. See lines 29, 33, 37-38 and 42-43.

Output sections which are part of an overlay are displayed at their load address. The run address is an attribute in the last column. See lines 28, 32, and 61. Note that the run address for *func23_scn:1* and *func1_scn* is the same. The run address for *func23_scn:2* is different because function 2 and function 3 are copied at the same time.

The *.ovly* section, which contains the linker generated copy tables, is displayed starting on line 64.

The details of the copy tables start on line 221. There is an entry for each instance of the table directive. One begins on line 223 and the other on line 226. Each entry displays the name of the copy table, the name of the output section that is copied, and the length, load address, and run address for each record used to perform the copy. Recall that the load allocation of *func23_scn* is split across two memory ranges. Thus there are two records in the corresponding copy table *func23_ctbl*, one for each block of code that is copied. The records are shown on lines 227-228.

Copyright © 2008, Texas Instruments Incorporated

5.6 Handling Cache Effects

The example only copies code sections. Thus, only program cache, not data cache, needs to be considered. The file *c6416_cache.c* contains a routine for informing the program cache that a change to program memory has occurred. Specific details are contained in the comments to that source code, and are not further described in this application note.

6 Trampolines

6.1 Problem Description

The C6000 and 470 CPUs each use different instructions sequences to call functions. One sequence takes less code space and cycles, but is limited by the distance allowed between the call sequence and the destination of the call. The other sequence has no distance limitation, but takes more code space and cycles.

Compare the two methods of performing calls in C6000 assembly:

```
; destination must be +/- 1M words from current PC
CALL _near_call
...
; destination can be anywhere
MVKL _far_call,A3
MVKH _far_call,A3
CALL A3
```

The ideal solution always uses the smallest instruction sequence that can perform the call.

Compiler based solutions to this problem include things like the *far* keyword, and memory models controlled by build time options. Such solutions fall short for two main reasons. First, it is difficult to consistently use the *far* keyword only where needed. Second, memory model options often use larger call sequences than necessary.

The linker replaces all these compiler based solutions with a new feature called trampolines.

Copyright © 2008, Texas Instruments Incorporated

6.2 Overview

Trampolines are summarized visually in Figure 4.

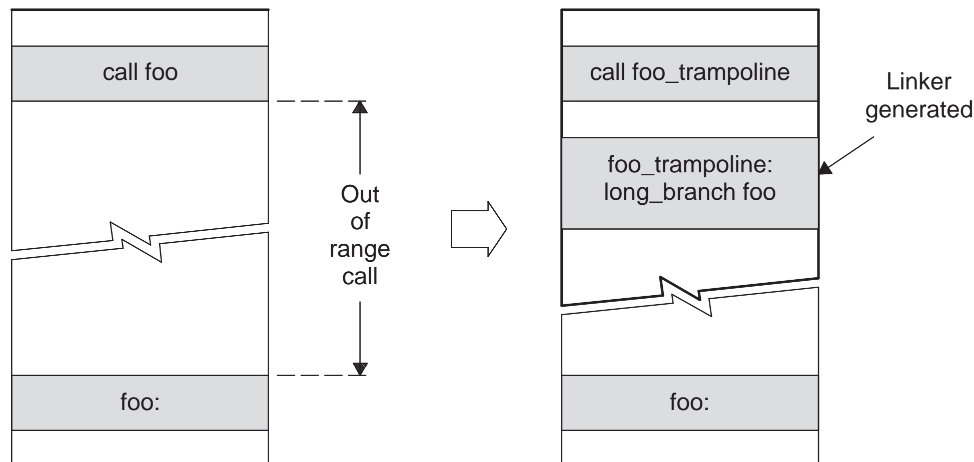

Figure 4. Trampolines

The compiler, or assembly language coder, always uses the smaller call sequence. Whenever the linker encounters a small call sequence that cannot reach the destination, it redirects the call to a linker generated trampoline. The trampoline uses a large branch instruction that can always reach the destination. 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.

A trampoline is always placed where it can be reached by a small call sequence. It is usually appended to the end of the output section, thought it may also be placed between input sections. If an input section is so big that a small call sequence cannot reach the end, then an error message is issued. Very large input sections, which must come from a single file, are rare.

In practice, most small function calls can reach their destinations. Relatively few calls require a trampoline.

Trampolines are not enabled by default. Use the linker option *--trampolines* to enable them. This option is available only for the C6000 and 470 toolsets. For C6000, support starts with Code Composer Studio™ version 3.00. For 470, it is supported in the currently available Code Composer Studio™ version 2.20. Note that under Code Composer Studio™ release 2.20 the option is *--large_model*.

6.3 Trampolines Improve Performance and Code Size

Consider complicated systems where the source is broken out into sub-modules that are built separately. Alternatively, consider building a library of related functions. In both cases it is often not known where the code may be placed in memory, and thus whether near calls can always reach their intended destinations. Such code is commonly built with command line options that cause calls to always use the larger encoding. On C6000 the options include

```
-ml1, -ml2, and -ml3
```

On 470, it includes the

```
-ml
```

option. While using these options increases cycles and code size, that cost is outweighed by the convenience of being able to place the code anywhere in memory.

Trampolines make those build options unnecessary, and improved performance often results. In practice, most calls are to related functions. Such functions are typically grouped in the same file or library and thus are typically near one another in memory. Therefore, near calls can reach these destinations. Using trampolines means that such calls are performed optimally. The costs of using trampolines are only borne by the less common calls to different modules or libraries that can be far away in memory.

Copyright © 2008, Texas Instruments Incorporated

Appendix B gives an example which illustrates the performance difference. This example is based on Reference Frameworks level 3 (RF3), and executes on a DSK6713. It is included in the zip file distributed with this application report. The build option*–ml3* is replaced by the build options *—trampolines –ml0*, and a 10% performance increase results. That is a very significant increase in performance for so little effort.

Using trampolines saves on code size as well. Recall the example of a C6000 near call and far call given in section 6.1. The near call is 8 bytes smaller than the far call. That rate of savings becomes a noticeable code size reduction when nearly all the calls change from far to near.

6.4 Basic Trampolines Example

The example from section 5 can make use of trampolines. The application in the example places the functions for functions 4-7 in external memory, while everything else is in internal memory. The address range distance between internal and external memory is very large, around 0x7ff00000 bytes. So the calls to functions 4-7, as well the calls that functions 4-7 make to the compiler runtime support functions, have to cross a large distance.

The example handles this by building with the far calls memory model switch *-ml1*. While that works, it has the unfortunate side effect of making every function call use more instructions, which is unnecessary most of the time.

The steps below remove the far calls build flag, note the errors that result, and then use trampolines to fix those errors.

1. Open up Code Composer Studio™ and the linker example project.
2. Remove the far calls build option. Select **Project | Build Options | Tab:Compiler | Category: Advanced.** In the Memory Models drop-down box select **Near Calls & Data**. Click **OK**.
3. Build the project. Select **Project | Build** or click the icon .
   Note several link-time errors in the **Build** tab of the **Output** window. For instance:
```
>> error: relocation overflow occured at address 0x00000068 in section '.text' of input file
'C:\projects\c6x\linker_example\Debug\demo.obj'. The 29-bit PC-relative displacement 536863840
at this location is too large to fit into the 21-bit PC-Relative field; the called function is
out of range from this call site. Consider using the '--trampolines' linker option to generate
trampolines to bridge the distance between far caller/callee pairs.
```
4. Add the switch for trampolines. Select **Project | Build Options | Tab:Linker**. Click to place the cursor at the end of the options box at the top, then type:`--trampolines`. See Figure 5 , where the option is circled in red. Click **OK**.

Copyright © 2008, Texas Instruments Incorporated

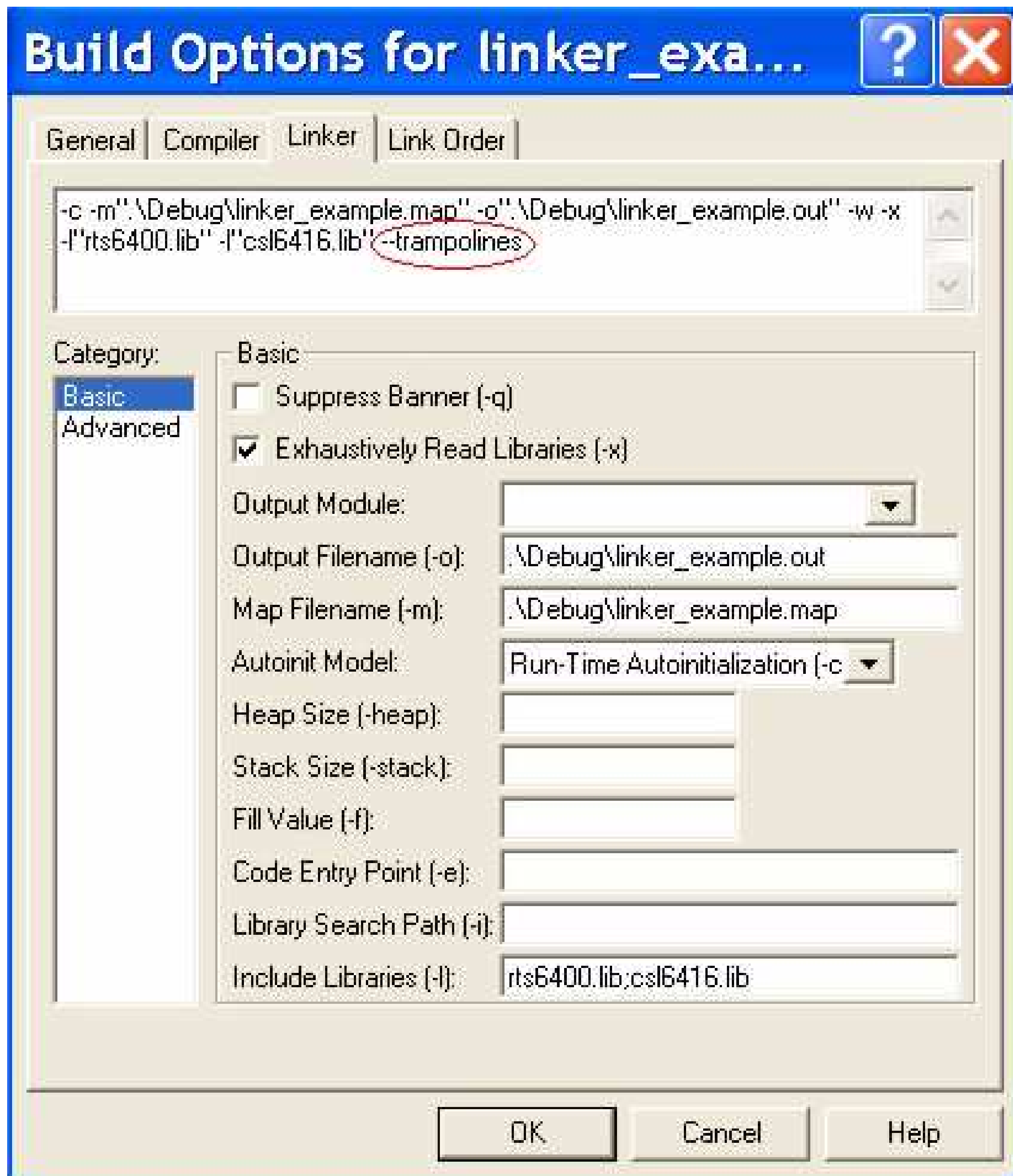

Figure 5. Enabling Trampolines

At this writing the Code Composer Studio™ build interface does not directly support adding the switch *--trampolines*, so it has to be added manually.

5. Build again. Select **Project | Build** or click on icon for incremental build.

   Only the linker will run, and the build will complete. All the places where errors occurred in the previous build are automatically changed to using trampolines.

6. Load the program. Select **File | Load Program**. Browse to the Debug sub-directory of linker_example, select the file name linker_example.out, then click **Open**.

7. Run the example. Select **Debug | Run** or press F5.

   The same output as before will appear in the **Stdout** tab of the **Output** window.

Copyright © 2008, Texas Instruments Incorporated

6.5 Inspect the Map File

Here is the part of the map file which displays the trampolines generated.

Far Call Trampolines

callee	addr	tramp	addr	call addr	call info
_printf	00007f20	.T$0005	800002c0	8000020c 8000026c	func5.obj (.func45_scn) func4.obj (.func45_scn)
_printf	00007f20	.T$0006	800004c0	8000040c 8000046c	func6.obj (.func67_scn) func7.obj (.func67_scn)
_func4	80000260	.T$0001	000008480	00006fe8	demo.obj (.text)
_func5	80000200	.T$0002	000084a0	00006ff0	demo.obj (.text)
_func6	80000400	.T$0003	000084c0	00006ff8	demo.obj (.text)
_func7	80000460	.T$0004	000084e0	00007000	demo.obj (.text)

The column contents are as follows:

- callee – function called
- addr – address of callee
- tramp – automatically generated name for the trampoline
- addr – where the trampoline resides in memory
- call addr – list of addresses from which a call using the trampoline originates
- call info – the object file and input section that contain the originating call. If the object file is from a library, the name of the library is shown as well.

The first four lines in the table are for calls to *printf* from functions 4-7. Each function calls *printf* one time. Recall that functions 4-7 are all linked into the output section *func4567_scn*. Under normal circumstances, this means all four calls to *printf* would share a single trampoline. However, the output section *func4567_scn* is split into two memory regions. Thus, there is a trampoline for each part of the split.

The remaining entries are for calls to functions 4-7 from the main routine in *demo.c*.

7 Feature Availability

Table 3. Feature Availability

Feature	Target(s)	CCS Version
Section Splitting	All	2.20
Copy Tables	All	3.00
Trampolines	ARM	2.20
	C6000	3.00

- Feature – which linker feature
- Target(s) – which CPU devices have support for the feature. The term **All** means C6000, 470, C55x, C54x, and C28x. These features are not supported for older devices.
- CCS Version – the version of Code Composer Studio™ which supports the feature. All later versions of Code Composer Studio™ support the feature as well. Code Composer Studio™ 2.20 is generally available today. Code Composer Studio™ 3.00 is planned for releases on various targets throughout 2004.

Table 4 indicates which version of the code generation tools is bundled with the given version of Code Composer Studio™. This information is for developers who perform builds outside of Code Composer Studio™.

Table 4. Code Generation Tools Versions by CCS Version

CCS Version	C6000	ARM	C55x	C54x	C28x
2.20	4.32	2.24	2.56	3.83	3.07
3.00	5.00	3.00	3.00	4.00	4.00

Copyright © 2008, Texas Instruments Incorporated

8 Summary

This application note describes three ways the linker makes it easy to use memory efficiently. Automatic section splitting places pieces of code or data into distinct memory ranges. Copy tables make implementing overlays a snap. Trampolines eliminate the need for far call memory models. The example gives you all the details on how to use these features in your application.

This is not the end of the line for linker features dedicated to managing memory well. Stay tuned for more to come.

9 Acknowledgements

Figures 1, 2, and 4 were provided by Todd Snider. The example in section 5 is based on work first done by Ning Kang. The reference frameworks example in Appendix B was provided by Alan Campbell. Thank you for these invaluable contributions.

Copyright © 2008, Texas Instruments Incorporated

Appendix A Placing Functions in Subsections Can Cause Code Growth

When the compiler places functions in individual subsections, it also marks these sections for conditional linking. If the linker sees no references to the symbols defined in such a section, it removes the section. In effect, if a function is never called, it is removed. In this manner, placing functions in subsections can save code size.

If very few functions are removed, however, placing functions in subsections can cause code size to grow. The reasons for this code growth are target specific, as detailed below.

A.10 TMS320C6000

On C6000 each input section that contains code must be aligned on a 32-byte boundary. Such alignment can create holes between the input sections. Placing functions in subsections increases the number of input sections that require 32-byte alignment, which increases the possibility of holes between sections, thus potentially causing overall code size to grow.

A formula for estimating the code growth in bytes is 16 * (functions – files) – sizeof (functions removed).

- 16 – the average size of the hole created by aligning on a 32-byte boundary. About half of the time the hole will be larger, the other half of the time it will be smaller.
- functions – the number of functions in the system, not counting those removed because they are not called.
- files – the number of files in the system
- sizeof(functions removed) – how much space is saved by removing functions that are not called

Note the term (functions – files) really represents the number of additional input sections that result from placing each function in a subsection. When the functions are not placed in subsections, the number of input sections is the same as the number of files. When functions are placed in subsections, the number of input sections is the same as the number of functions. Thus the number of additional input sections is (functions – files).

The rest of this section explains why input sections which contain code must be aligned on a 32-byte boundary.

Here are some basic C6000 technical terms and relationships:

- All instructions on C6200, C6400, and C6700 devices are 4-bytes long
- Instructions are fetched (read) in groups of 8 instructions called a fetch packet. A fetch packet is 32-bytes long. A fetch packet boundary is a 32-byte boundary.
- An execute packet is a group of instructions which execute in parallel. An execute packet can contain 1 to 8 instructions.
- On C6200 and C6700 devices, an execute packet may not span a fetch packet boundary.
- On C6400 devices, an execute packet can span a fetch packet boundary with one exception: an execute packet that can be the target of a branch may not span a fetch packet boundary.

There are two main reasons why an input section which contains code must be aligned on a fetch packet boundary (32-byte boundary).

The assembler performs error checking to insure an execute packet does not span a fetch packet boundary. Such checking requires the assembler to know the location of the fetch packet boundaries. Such knowledge can be assured only by first aligning the section on a fetch packet boundary.

An input section that contains instructions must end on a fetch packet boundary. If it does not, when the last fetch packet in the section is read it will pick up the first few bytes of the next piece of memory. For that to be safe, that next piece of memory must exist, and it must contain valid C6000 instructions. Absent such conditions, a system crash could result. Since these conditions cannot be guaranteed, the assembler insures the section ends on a fetch packet boundary. This assurance is implemented by aligning the section on a fetch packet boundary, then padding the end of section with the appropriate number of NOP instructions.

Copyright © 2008, Texas Instruments Incorporated

A.11 TMS320C55x

On TMS320C55x there are two different sizes of call instructions. The smaller call instruction, while clearly preferred, is limited by the distance in memory between the call and the destination of the call. The larger call instruction has no limit on the distance to the call destination.

For calls to functions defined in the same input section, the assembler always encodes the smaller call instruction (with extremely rare exceptions). If the call is to a function defined in a different input section, even if that input section is in the same file, the assembler always encodes the larger call instruction.

Placing functions in subsections means no calls (except recursive calls) are to a function in the same input section. Thus the larger call instruction is always encoded. Always encoding the larger call instruction can cause overall code size to grow.

A.12 TMS470R1x

The TMS470R1x instruction set does not support any 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 input 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 input 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.

Copyright © 2008, Texas Instruments Incorporated

Appendix B Trampolines Improve Performance on Reference Frameworks Example

B.1 Introduction

Reference Frameworks for eXpressDSP Software are provided as starterware for developing applications that use DSP/BIOS and the TMS320 DSP Algorithm Standard (also known as XDAIS). Reference Frameworks are available for different levels of system complexity. The example here is based on Reference Frameworks Level 3 (RF3) as described in the application note *Reference Frameworks for eXpressDSP Software: RF3, A Flexible, Multi-Channel, Multi-Algorithm, Static System* (SPRA793).

The appendix walks through building the example code with two different option sets, and compares performance. The example executes on a DSK6713.

The first build uses –ml3. Under –ml3, all calls are far calls that can reach any destination, but cost more in cycles and memory. The example, built with these options, imposes a CPU load of about 10.5%.

The second build uses —trampolines –ml0 in place of –ml3. The advantage of using trampolines instead of –ml3 is described in section 6.3. The example, built with these options, imposes a CPU load of about 9.5%. That is a performance improvement of just under 10%! No source changes required!

Note changing from –ml3 to –ml0 also changes how global data is handled. That difference is discussed in section B.5.

The files are in the zip file supplied with this application report. Extract the contents of that zip file to an empty directory. The example makes use of the file DSK6713.gel and the directory referenceframeworks. The directory linker_example is used in section 5.

The directions below use the name linkex to represent the directory where the zip files are extracted.

B.2 System Setup

Required components:

- DSK6713
- C6000 Code Composer Studio™ v3.00
- An emulator such as the XDS-560
- Audio cable
- Speakers or headphones

At this writing, the only way to control the DSK6713 with Code Composer Studio™ v3.00 is through an emulator. Drivers to support a USB interface will be released at a later date.

Connect the audio cable to the sound output jack on the PC, and the line in jack on the DSK6713. Connect the speakers or headphones to the headphone or line out jacks on the DSK6713.

The default GEL file supplied with the emulator needs is not specifically configured to the DSK6713. Replace it with the DSK6713.gel file supplied with the example.

1. Open Code Composer Studio™ Setup
2. Configure the system to use an emulator connected to a C671x target system.
3. In the leftmost column, right click on the name of the emulator and select **Properties**.
4. Select the tab **Startup GEL File(s).**
5. Click the **“..”** box on the far right, browse to linkex, select the filename DSK6713.gel, then click **Open**, then click **Finish**.
6. Save the configuration. Select **File | Save**.
7. Quit Code Composer Studio™ Setup. Select **File | Exit.**

Copyright © 2008, Texas Instruments Incorporated

B.3 Build With -ml3 and Check Performance

Building the reference frameworks example involves building several libraries as well as a Code Composer Studio™ project. Batch files are supplied to perform these builds.

1. Open a DOS command window
2. Configure the path and environment variables for using Code Composer Studio™ command line build tools. Run the batch file *CCS install directory*\DosRun.bat.
3. Change directory to linkex\referenceframeworks.
4. Build the libraries and application example with –ml3. Run the batch file build6000_rf3_dsk6713_ml3.bat.
5. Do not close the DOS window.
6. Start Code Composer Studio™
7. Open the reference frameworks example project build with –ml3. Select **Project | Open**. Browse to linkex\referenceframeworks\apps\rf3\dsk6713, select the file app_ml3.pjt, then click **Open**.
8. Connect to the DSK6713. Select **Debug | Connect**.
9. Load the program. Select **File | Load Program**. Browse to the Debug subdirectory of the project directory, select app.out, then click **Open**.
10. Start playing an audio source on the PC. For example, open Windows Media Player to play music from a CD.
11. Back in Code Composer Studio™, run the application. Choose **Debug | Run** or hit F5.
12. Listen for the audio from the PC source to begin playing through the speakers or headphones connected to the DSK6713.
13. Check the performance level by using the CPU Load Graph. Choose **DSP/BIOS | CPU Load Graph**. Note the CPU load level is about 10.5%.
14. Halt the application. Choose **Debug | Halt**.
15. Close the project. Select **Project | Close**.

B.4 Build With --trampolines -ml0 and Check Performance

1. Return to the DOS window.
2. Rebuild the libraries and application with –-trampolines and –ml0. Run the batch file build6000_rf3_dsk6713_ml0_trampolines.bat.
3. Return to Code Composer Studio™.
4. Open the reference frameworks example project built with –-trampolines –ml0. Select **Project | Open**. Browse to linkex\referenceframeworks\apps\rf3\dsk6713, select the file app_ml0_trampolines.pjt, then click **Open**.
5. If not connected to the DSK6713, select **Debug | Connect.**
6. Load the program. Select **File | Load Program**. Browse to the Debug subdirectory of the project directory, select app.out, then click **Open**.
7. Make sure the audio started in step 10 above is still playing.
8. Back in Code Composer Studio™, run the application. Choose **Debug | Run** or hit F5.
9. Listen for the audio from the PC source to begin playing through the speakers or headphones connected to the DSK6713.
10. Check the performance level by using the CPU Load Graph. Choose **DSP/BIOS | CPU Load Graph**. Note the CPU load level is now about 9.5%.

Copyright © 2008, Texas Instruments Incorporated

B.5 Compare Performance

A CPU load of 9.5% is almost 10% less than a CPU load of 10.5%. That is quite a performance improvement just for changing build options!

The measurement conditions for the CPU load measurement are:

- Reference Framework Level 3 running on a 225Mhz c6713 DSK
- Sampling rate of 44.1 khz with 80 samples per frame. This equates to a frame rate of 44100 / 80 = 551 Hertz.

The relatively high frame rate means that function calls are being made frequently, as compared to a G723 system which has a frame rate of 8000 / 240 = 33Hz. The more a function is called, the more the impact of employing trampolines to make that a near call rather than a far call.

On C6000 Performance Audio systems, which can run at up to 96khz, using trampolines to change far calls to near calls can have even more impact.

B.6 Note on Data: -ml0 Works as Well as -ml3

Up to this point, the example has focused on how code and calls are handled. Another difference between the build options –ml3 and –ml0 is how data is handled. Note this discussion only covers global and static variables. Variables dynamically allocated on the stack or from the heap are not related to this issue in any way.

Similar to the difference between near and far calls, there is a distinction between near and far data. Far data takes more instructions to access than near data. The limitation of near data is that the total size of all near data cannot exceed 32K bytes.

Under –ml3, all global data defaults to far access. Under –ml0, aggregate data defaults to far access, and scalar data defaults to near access. Aggregate data is arrays, structures, and unions. Scalar data is simple variables declared with integer, double, etc.

It may seem that –ml0 imposes difficult constraints on how data is handled that is best avoided by using –ml3. That simply is not the case.

Under –ml0, only scalars are declared near. And the total size of all near data is limited to 32K bytes. The largest scalar types use up 8 bytes: long long, double, and long double. Even if all the scalars were one of those 8-byte types, it would take 4,097 differently named scalars to exceed the 32K byte limit. Exceeding such a large limit is highly unlikely.

Copyright © 2008, Texas Instruments Incorporated

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2023, Texas Instruments Incorporated