

# Use of the Text and Visual Linkers

## 1 Introduction

This is a very slightly updated version of a memo written summer of 2003. The main change is the addition of cmd file memory allocation allowing the EECS452.cmd file to be used with the large memory model. Last semester no use was made of the visual linker and this is expected to be the same this semester.

A linker is a program used to combine object files to create a complete program in a form that can be loaded into a computer's memory and placed into execution.

TI supplies two linkers:

- Visual Linker
- text linker

The linker to be used by a Code Composer Studio project can be selected following the menu path:

```
Tools
  Linker Configuration
    Use the Visual Linker
    Use the text linker
```

The choice is global to CCS rather than for a specific project and is not recorded within a project's project file. This means that if you are working with a set of projects some of which use one linker and the other linker you need to keep switching back and forth.

Which linker to use?

TI has invested a lot of effort in developing the Visual Linker. However, it does not seem to be used by TI. Searches were made of the ti directory and the subdirectories, created when the C5510DSK was installed, seeking text linker command files (extension .cmd) and Visual Linker recipe files (extension .rcp).

The search on \*.cmd resulted in 156 files being found. Many of these were generated by the TI DSP/BIOS configuration tools used to automate program creation.

The search on \*.rcp resulted in 4 being found. Only one applied to the C55x processor.

Another indicator of importance is the lack of a TI manual for the Visual Linker whereas the text linker is documented in the Assembly Language Tools manual.

In spite of the somewhat negative mixed message we will make at least some use of the Visual Linker in EECS 452. Similarly, we may or may not make use of command files generated by the DSP/BIOS configuration program.

## 2 Memory on the C5510 DSK

The C55x uses byte addressing for executable code and word addressing when working with data. Depending upon what is being described, much of the TI documentation uses either and/or both byte and word addresses. Both the text linker and the Visual Linker use *only* bytes for memory addresses and section sizes. This is both for code and data.

The C55x has a *unified memory map*. This means that code and data share the same physical memory. However, separate busses are used to access code and data. This a variation on the original Harvard architecture where data and code were placed in physically separate memory units each having their own busses.

One of the banes of a unified memory map is that it is possible for a program to accidentally destroy itself. Physically separate data and memory would (maybe) minimize the effects of a wild pointer. Then again, in either case the program would be wrong. The use of a unified memory map is not a problem with a correctly written program.

MMR:	0x000000	—	0x0000BF	on-chip registers & reserved
DARAM:	0x0000C0	—	0x00FFFF	8 8K byte blocks
SARAM:	0x010000	—	0x04FFFF	32 8K byte blocks
EXTERN P0:	0x050000	—	0x3FFFFFF	DSK SDRAM
EXTERN P1:	0x400000	—	0x5FFFFFF	DSK Flash
	0x600000	—	0x7FFFFFF	DSK CPLD
EXTERN P2:	0x800000	—	0xBFFFFFF	DSK daughter card
EXTERN P3:	0xC00000	—	0xFFFFFFFF	DSK daughter card

All addresses above are in bytes.

The on-chip memory is 163,840 words or equivalently 327,680 bytes. The C5510DSK off-chip memory is 4,194,304 words. This can be viewed as being two 2,097,152 word units. In the first unit the first 163,840 words are overlaid by the on-chip memory meaning that only 1,933,312 external SDRAM words are accessible. The section unit address space is overlaid by the flash memory and the CPLD. The second SDRAM 2M word unit can be accessed but not readily. Unless required by a project, we will ignore the presence of the second 2M word block of off-chip memory.

DARAM block addresses start at multiples of 0x2000 (bytes). SARAM block addresses also start at multiples of 0x2000 (bytes).

Accesses to off-chip memory are of single type. The data path between the processor and the external memory is 32-bits wide.

When accessing the off-chip memory the EMIF needs to be programmed to supply a sufficient wait states to insure proper operation.

### 3 What happens if we don't place things correctly?

The question is: can you simply place all of the sections into memory without worrying about where code and data are being placed?

The answer is that the pipeline is aware of what types of memory are being accessed by each instruction and adjusts its operation (i.e., timing) accordingly.

For example: executing, in a SARAM block, a multiply instruction with both operand values contained in the same block gives the desired value. This instruction will make a program fetch and two data fetches. Because this is being done in a SARAM block three cycles will be necessary and the pipeline does what is needed to cause this to be the case.

If instead, the two operand values are located in a DARAM block or in two other individual SARAM blocks, then the instruction will be executed in one cycle.

The concern with "proper" placement is not one of correctness of the computational result but with the efficiency of execution. Performance is the name of the game that we are playing here. Otherwise why use a DSP chip in the first place?

### 4 What memory requirements are imposed by C

As noted above a program will execute correctly (not necessarily as quickly as hoped for) regardless of where code and data are placed in memory. However, with careful placement, a program's execution time will generally be significantly reduced.

The following guidelines are from *TMS320C55x DSP Programmer's Guide*, SPRU376A. This is an excellent guide mainly focused on how to get C to generate code that executes efficiently.

- The `.stack` and `.sysstack` should be placed into DARAM. Alternatively they should be placed into separate SARAM blocks. Because these two stacks share a common data page pointer (SPH) both must reside in the same 64K word memory page.

- The `.bss` and `.stack` sections should be either be placed into the same DARAM block or into separate SARAM blocks.
- Use the `DATA_SECTION` pragma to define sections that can be placed appropriately into specific DARAM or SARAM blocks at link time.

For example, use separate data sections to hold the arrays containing the input data and the coefficient values for a FIR filter. These sections then can be placed in DARAM or SARAM areas in order optimize execution.

## 5 The text linker

The text linker is described in the *TMS320C55x Assembly Language Tools User's Guide*, SPRU280G.

There are quite a few linker command files included with the C5510DSK CCS files. However, I did not stumble one specific to the C5510DSK. So, we will make one using what information that we have on hand.

The small memory model will be used and we will use the sample linker file given in the *TMS320C55x DSP Programmer's Guide* as our starting point. This file is reproduced in Figure 1.

As a reminder, when working with the linkers all addresses and sizes are in terms of bytes.

Several of the linker files included with CCS were examined to get a feel for what what might or might not be important when setting up a generic C5510 command file for EECS 452 lab use. The resulting file is

```

/*****
LINKER command file for LEAD3 memory map.
Small memory model
*****/

stack    0x2000 /* Primary stack size */
sysstack 0x1000 /* Secondary stack size */
heap     0x2000 /* Heap area size */

c          /* Use C linking conventions: autoinit vars at runtime */
u _Reset   /* Force load of reset interrupt handler */

MEMORY
{
  PAGE 0:    /* Unified Program/Data Address Space */
    RAM (RWIX) : origin = 0x000100, length = 0x01ff00 /* 128Kb page of RAM */
    ROM (RIX)  : origin = 0x020100, length = 0x01ff00 /* 128Kb page of ROM */
    VECS (RIX) : origin = 0xffff00, length = 0x000100 /*256byte int vector*/
  PAGE 1:    /* 64Kword I/O Address Space */
    IOPORT (RWI) : origin = 0x000000, length = 0x020000
}

SECTIONS
{
  .text > ROM PAGE 0 /* Code */

  /* These sections must be on same physical memory page */
  /* when small memory model is used */
  .data    > RAM      PAGE 0 /* Initialized vars */
  .bss     > RAM      PAGE 0 /* Global & static vars */
  .const   > RAM      PAGE 0 /* Constant data */
  .system  > RAM      PAGE 0 /* Dynamic memory (malloc) */
  .stack   > RAM      PAGE 0 /* Primary system stack */
  .sysstack > RAM      PAGE 0 /* Secondary system stack */
  .cio     > RAM      PAGE 0 /* C I/O buffers */

  /* These sections may be on any physical memory page */
  /* when small memory model is used */
  .switch  > RAM      PAGE 0 /* Switch statement tables */
  .cinit   > RAM      PAGE 0 /* Autoinitialization tables */
  .pinit   > RAM      PAGE 0 /* Initialization fn tables */

  vectors  > VECS     PAGE 0 /* Interrupt vectors */

  .ioport  > IOPORT PAGE 1 /* Global & static IO vars */
}

```

Figure 1: Starter text linker command file. From SPRU376A.

```

/*****
LINKER command file for EECS 452 C5510DSK memory map.
Small memory model --- Version 1.0 25Jul2003 KM
Added large pages --- Version 1.01 18Nov2003 KM

Appears to work ok for large memory model as well.
Linker represents addresses and allocations using 8-bit bytes!!!!!!

*****/

-stack 0x2000 /* Primary stack size .. fills one 8KB block */
-sysstack 0x1000 /* Secondary stack size .. fills one half 8KB block */
-heap 0x2000 /* Heap area size .. fills one 8KB block */

-c /* Use C linking conventions: auto-init vars at runtime */
-u _Reset /* Force load of reset interrupt handler */

MEMORY
{
PAGE 0: /* ---- Unified Program/Data Address Space ---- */
MMR_RSVD : origin = 0x000000, length = 0x0000BF /* 192 bytes MMR reserved */
VECT (RWIX) : origin = 0x000100, length = 0x000100 /* 256 byte interrupt vector */
DARAM (RWIX) : origin = 0x000200, length = 0x00FD00 /* almost 64KB of DARAM */
SARAM0 (RWIX) : origin = 0x010000, length = 0x010000 /* 64KB of SARAM */
SARAM1 (RWIX) : origin = 0x020000, length = 0x020000 /* 128KB of SARAM */
SARAM2 (RWIX) : origin = 0x040000, length = 0x010000 /* 64KB of SARAM */
/* SDRAM has 0xB0000 37776 KB of SDRAM .. notall allocated here */
SDRAM0 (RWIX) : origin = 0x050000, length = 0x010000 /* 64KB of SDRAM */
SDRAM1 (RWIX) : origin = 0x060000, length = 0x020000 /* 128KB of SDRAM */
SDRAM2 (RWIX) : origin = 0x080000, length = 0x020000 /* 128KB of SDRAM */
SDRAM3 (RWIX) : origin = 0x0A0000, length = 0x020000 /* 128KB of SDRAM */
SDRAM4 (RWIX) : origin = 0x0C0000, length = 0x020000 /* 128KB of SDRAM */
SDRAM5 (RWIX) : origin = 0x0E0000, length = 0x020000 /* 128KB of SDRAM */
SDRAM6 (RWIX) : origin = 0x100000, length = 0x020000 /* 128KB of SDRAM */
SDRAM7 (RWIX) : origin = 0x120000, length = 0x020000 /* 128KB of SDRAM */
SDRAM8 (RWIX) : origin = 0x140000, length = 0x020000 /* 128KB of SDRAM */
SDRAM9 (RWIX) : origin = 0x160000, length = 0x020000 /* 128KB of SDRAM */
SDRAM10 (RWIX) : origin = 0x180000, length = 0x020000 /* 128KB of SDRAM */
SDRAM11 (RWIX) : origin = 0x1A0000, length = 0x020000 /* 128KB of SDRAM */
FLASH : origin = 0x400000, length = 0x80000
VECS (RIX) : origin = 0xffff00, length = 0x000100 /* 256-byte int vector*/
}

SECTIONS
{
.text > SARAM0 PAGE 0 /* Code */

/* These sections must be on same physical memory page */
/* when small memory model is used */

.data > DARAM PAGE 0 /* Initialized vars */

```

```

.bss      > DARAM  PAGE 0 /* Global & static vars */
.const   > DARAM  PAGE 0 /* Constant data */
.system  > DARAM  PAGE 0 /* Dynamic memory (malloc) */
.stack   > DARAM  PAGE 0 ALIGN = 0x2000 /* Primary system stack */
.sysstack > DARAM  PAGE 0 ALIGN = 0x2000 /* Secondary system stack */
.cio     > SARAM0 PAGE 0 /* C I/O buffers */

/* These sections may be on any physical memory page */
/* when small memory model is used */

.switch  > SARAM0 PAGE 0 /* Switch statement tables */
.cinit   > SARAM0 PAGE 0 /* Auto-initialization tables */
.pinit   > SARAM0 PAGE 0 /* Initialization fn tables */

vectors  > VECT   PAGE 0 /* Interrupt vectors */

/* Allocate pages in SARAM for when using large memory model */

SARAMA   > SARAM0 PAGE 0
SARAMB   > SARAM1 PAGE 0
SARAMC   > SARAM2 PAGE 0

/* Allocate pages in SDRAM for when using large memory model */

SDRAMA   > SDRAM0 PAGE 0 /* 32K word page */
SDRAMB   > SDRAM1 PAGE 0 /* 64K word page */
SDRAMC   > SDRAM2 PAGE 0 /* 64K word page */
SDRAMD   > SDRAM3 PAGE 0 /* 64K word page */
SDRAME   > SDRAM4 PAGE 0 /* 64K word page */
SDRAMF   > SDRAM5 PAGE 0 /* 64K word page */
SDRAMG   > SDRAM6 PAGE 0 /* 64K word page */
SDRAMH   > SDRAM7 PAGE 0 /* 64K word page */
SDRAMI   > SDRAM8 PAGE 0 /* 64K word page */
SDRAMJ   > SDRAM9 PAGE 0 /* 64K word page */
SDRAMK   > SDRAM10 PAGE 0 /* 64K word page */
SDRAML   > SDRAM11 PAGE 0 /* 64K word page */
}

```

For most of the lab programs the amount of DARAM should be much more than adequate. The memory locations most likely to be involved in calculations were placed in DARAM. The code was placed in SARAM. The stack and system stacks were aligned to start on block boundaries.

Even after working with the interrupt vectors with the C5402DSKs I still retain a bit of confusion on why movement to low memory. Though I do it and the DSP/BIOS configuration programs does this as well. I followed suit. There will be more to come on this topic in future notes.

The size of the flash memory area was extracted from a DSP/BIOS configuration program generated command file and was just copied here.

The heap is not very large. The heap is where `calloc` and its relatives obtain

memory to be allocated. If dynamic memory allocation is being used the size and location of the heap and its section may need to be changed.

Setting the sizes of the two stacks has not been discussed anywhere that I've seen. If these are not sized adequately things won't work right. There is no system present here. Setting things up properly is our responsibility. Being able to set things up properly requires knowing what is being done and how things work.

The IOPORT lines were removed because they caused a problem when using Visual Linker to convert the command file into a recipe file. A check of the C/C++ manual did not find any mention of a `.ioport` section.

The command file shown in Figure ?? can be used, as is, allowing the linker to place components pretty much filling the space available going from low to high addresses. Greater control can be exercised if desired. See the linker chapter in the *Assembly Language Tools User's Guide*.

## 6 The Visual Linker

The only information that I'm aware of describing the Visual Linker is that provided with Code Composer Studio. It is assumed here that the reader has gone through the CCS Visual Linker tutorial.

When initially creating a recipe file the Visual Linker needs to know something about the memory on the target processor. This can be obtained in one of two ways. The Visual Linker can extract the information from a text linker command file or a memory template file (extension `.mem`) can be used.

The above command file in Figure ?? seems to work reasonably well for this task so we will focus here on the `.mem` template file.

A search was made on the `ti` directory for files having the `.mem` extension. Five were found that were for use with the C55x processor. The two most relevant to the C5510DSK were a generic one for the C5510 and one for the C5510EVM. Starting with the generic version adding information specific to the C5510DSK using the EVM version seemed to be a reasonable thing to do.

The generic memory template file is listed below (too large to fit into a figure). This can be found in the directory

```

\ti\bin\utilities\vislink\Templates\c55x\c5510\generic

/* TMS320C5510 core only                                     */
/*                                                         */
/* Use this description if the particular C5510 board is not yet known, */
/* or as a starting point for creating a custom description.      */
/*                                                         */
/* When making memory alterations:                            */
/* 1. Take care to declare Read/Write/eXecute/Initializable (RWXI) */
/* properties correctly! THIS IS *REAL* IMPORTANT!             */

```

```

/* 2. The visual linker defines "Initializable" as persistent memory */
/* (i.e. FLASH, ROM). Use "Write" only if you keep variables */
/* (as opposed to code & constants) in the region at run-time. */
/* 3. Never give a region eXecute capabilities if it is declared */
/* in a data space. Regions that may hold code MUST be declared */
/* in code space. They may be shared with a data space */
/* using the SHARED_ADDRESSES directive of the .mem file. */
/* 4. Currently, there is no visual interface to create a */
/* SHARED_ADDRESSES directive. This must be done by hand in the */
/* .mem file using a text editor. */
/* If you start with an existing .mem file, you probably do not */
/* need to alter the SHARED_ADDRESSES directive unless you have new */
/* devices that swap in and out of memory (usually at boot time). */
/* 5. If writing the .mem file by hand, declare the primary code space */
/* 1st, the primary data space 2nd and the primary IO space 3rd. */
/* Others may follow in any order. */
/* 6. Its best if SPACE names contain one of the following substrings: */
/* {"prog", "code", "data", "io"}. The visual linker looks for */
/* these strings (case-insensitive) to help determine semantics for */
/* the SPACE. */
/* 7. An annoying warning is avoided if read-only region names contain */
/* "rom" or "ROM" */
/* 8. An annoying warning is avoided if non-readable non-writeable */
/* (not allocatable) region names contain "reserved" or "RESERVED" */
/* or "RSVD" */
MEMORY C5510 (family = c55x) {

    /* MP=0 */
    SPACE memory_space PAGE 0
        (OVERLAY = memory_space,
         SWAP_IN =
             "DEBUG_Global(""old_ST3"" ) = ST3;"
             "ST3 = (ST3 & 0xF7FF); /* MP=0 */"
         SWAP_OUT =
             "ST3 = DEBUG_Global(""old_ST3"" );"
         IS_SWAPPED_IN =
             "((ST3 & 0x800) == 0x000) /* MP==0 */"
        ):

        /* memory mapped registers (part of 1st block of DARAM) */
        MMR_RSVD ( ): o= 0x00000 e= 0x000BF

        /* On-chip Dual Access RAM, divided into 4 blocks of 16K bytes: */
        /* 0-3FFF, 4000-7FFF, 8000-CFFF, D000-FFFF */
        DARAM (RWX ): o= 0x000C0 e= 0x0FFFF

        /* On-chip Single Access RAM, divided into 16 blocks of 16K bytes: */
        /* 10000-13FFF, 14000-17FFF, 18000-1CFFF, 1D000-1FFFF */
        /* 20000-23FFF, 24000-27FFF, 28000-2CFFF, 2D000-2FFFF */
        /* 30000-33FFF, 34000-37FFF, 38000-3CFFF, 3D000-3FFFF */

```

```

/* 40000-43FFF, 44000-47FFF, 48000-4CFFF, 4D000-4FFFF */
SARAM      (RWX ): o= 0x010000 e= 0x04FFFF

EXTERNAL_CE0(RWX ): o= 0x050000 e= 0x3FFFFF
EXTERNAL_CE1(RWX ): o= 0x400000 e= 0x7FFFFF
EXTERNAL_CE2(RWX ): o= 0x800000 e= 0xBFFFFF
EXTERNAL_CE3(RWX ): o= 0xC00000 e= 0xFF7FFF

PDR0M      (RX  ): o= 0xFF8000 e= 0xFFFFF

SPACE IO_space PAGE 2:

PERIPHERAL_0(RW ): o= 0x00000 e= 0x003FF /* FHEA */
PERIPHERAL_1(RW ): o= 0x00400 e= 0x007FF /* EMULATOR/TEST */
PERIPHERAL_2(RW ): o= 0x00800 e= 0x00BFF /* EXT. MEMORY INTERFACE */
PERIPHERAL_3(RW ): o= 0x00C00 e= 0x00FFF /* DMA */

PERIPHERAL_4(RW ): o= 0x01000 e= 0x013FF /* TIMER #0 */
PERIPHERAL_5(RW ): o= 0x01400 e= 0x017FF /* INSTRUCTION CACHE */

PERIPHERAL_7(RW ): o= 0x01C00 e= 0x01FFF /* CLOCK GENERATOR */

PERIPHERAL_8(RW ): o= 0x02000 e= 0x023FF /* TRACE FIFO */
PERIPHERAL_9(RW ): o= 0x02400 e= 0x027FF /* TIMER #1 */
PERIPHERAL_10(RW ): o= 0x02800 e= 0x02BFF /* SERIAL PORT #0 */
PERIPHERAL_11(RW ): o= 0x02C00 e= 0x02FFF /* SERIAL PORT #1 */

PERIPHERAL_12(RW ): o= 0x03000 e= 0x033FF /* SERIAL PORT #2 */
PERIPHERAL_13(RW ): o= 0x03400 e= 0x037FF /* GPIO */
PERIPHERAL_14(RW ): o= 0x03800 e= 0x03BFF /* ID */

/* MP=1 */
SPACE memory_space_MP1 PAGE 3
(OVERLAY = memory_space,
 SWAP_IN =
  "DEBUG_Global("old_ST3" ) = ST3;"
  "ST3 = (ST3 | 0x800); /* MP=1 */"
 SWAP_OUT =
  "ST3 = DEBUG_Global("old_ST3" );"
 IS_SWAPPED_IN =
  "((ST3 & 0x800) == 0x800) /* MP==1 */"
):

// RSVD_2_CE3 ( ): o= 0xFF8000 e= 0xFFFFF
}

/* Table of address ranges vs. sets of spaces that share them. */
/* For each entry in the table, */
/* if a region's parent appears in the space set */
/* and the region is fully contained in the address range, */
/* then the region is shared with the other spaces of the set. */

```

```
SHARED_ADDRESSES C5510 {  
    o= 0x0000 e= 0x0FF7FFF {memory_space, memory_space_MP1}  
}
```

There is a fair resemblance to a text linker command file.

Some of the comments are technically correct but are a bit misleading. The smallest granularity of a DARAM block is 8K bytes not 16K bytes. Similarly for the SARAM blocks. Indicating and using a more coarse granularity should pretty much be harmless.

The DARAM starts at 0x00060. In an earlier version of the EECS 452 text linker file this also was done. If the interrupt vector is to be repositioned having the allocation already made makes things simpler.

The SDRAM section named in the EECS452 text linker command file occupies the address space associated with the EXTERNAL\_CE0 section.

No space has been allocated in this template for the stacks or the heap. This can be done during the creation of the Visual Linker recipe or later using the properties pages of the associated output section folders in the memory view. Running a test run indicates that the default stack sizes are 0x3E8 bytes and the heap size is 0x7D0. Just one trial run was made. The defaults may change depending on program size. Don't know at this point.

For all practical purposes the generic C5510 memory template should suit our purposes in EECS452 quite well. We might change the name of the template section, EXTERNAL\_CE0, to SDRAM but it hardly seems the worth the effort. For the present both names will be used interchangeably.

As a side note, I have occasionally had CCS and the Visual Linker go into a malfunction mode where things that once worked no longer worked. It wasn't even possible to shut Windows 2000 down at this point. This can be corrected by using the task manager to terminate the `cc_app.exe` and `visuallinker.exe` tasks that somehow got detached from what is going on. We had similar problems with the CCS/Visual combination with the C5402DSKs.