ANALOG DEVICES Engineer To Engineer Note EE-132

Technical Notes on using Analog Devices' DSP components and development tools
Phone: (800) ANALOG-D, FAX: (781) 461-3010, EMAIL: dsp.support@analog.com, FTP: ftp.analog.com, WEB: www.analog.com/dsp

Placing C Code and Data Modules in SHARC memory using VisualDSP++ ™

Introduction

When developing C code for a DSP application, you often need to explicitly place code and data at specific addresses in memory.

This EE Note presents two techniques to do this. The first technique involves editing the LDF files to explicitly locate entire object files in memory segments. The second technique shows how to use the SECTION keyword in your C source code to explicitly place individual variables and functions in specific memory segments.

In the pre-VisualDSP releases of the SHARC Tools (Release 3.3) you could easily place C code in an alternate memory segment defined in the architecture file by using the ***-mpmcode*** compiler switch. This switch allowed you to specify which architecture file segment contained that specific code or data object file. An example command line using this switch is shown below:

g21k filename.c -mpmcode=alt_pmco -a alt.ach

In VisualDSP++ the *-mpmcode* switch is no longer valid. Instead, users can use one of the methods described below to achieve the same results.

The job of the C compiler is to translate the user's C code into assembly code. In doing this, the compiler needs to place code and data in memory segments that are defined in the LDF file. The compiler uses a predetermined set of memory segments to place code and data modules. The default LDF file that comes with the VisualDSP++ tools use these memory segments which are listed and described below:

seg_pmco	Default segment name for code storage.
seg_dmda	Default segment name for data memory global variables and static variables.
seg_pmda	Default segment name for program memory data variables.
seg_rth	Segment name for system initialization code storage and Runtime Header.
seg_init	Segment name for Runtime Initialization storage.

Because the compiler uses these segment names during compilation, the LDF file must be edited in order to place code or data in other memory segments. These changes are made to the MEMORY and SECTIONs portion of the LDF file. The example below shows how to manipulate your LDF file to change the placement of your code.

Code Example

Consider the following 2 code modules:

```
/* main.c */

extern int sum(int a,int b);
int ext_a=0xa,int ext_b=0xb;


void main(void)
{
      int i=5;
      int j=3;
      int total;
```

```
      i = sum(i,j);
      total=ext_a+ext_b;
      idle();
}

/* ext.c */

int sum(int a, int b)
{
return a+b;
}
```

Main.c is a code module that will call the function **sum** from the file named **ext.c**. If we were to compile this code with any of the default LDF files, all of the code modules would be placed in **seg_pmco**, and all of the data variables would be placed in **seg_dmda**.

Our goal is to place the external function **sum** in a code segment a called **alt_pmco,** and to place the two variables, ext_a and ext_b in a code segment called **mem_dmda.** There are two ways to do this: Modify the LDF files, or use the SECTION keyword in the C source code. We will show examples of both methods.

Option #1 – Modifying the LDF File

We'll use this method to change the linker's placement of the object file **ext.doj**.

First, we will add two new segments to the MEMORY section of the LDF file. The two memory segments are named **alt_pmco** (for code storage) and **mem_dmda** (for 16-bit data storage – this will be used later).

```
MEMORY
{
seg_rth  { TYPE(PM RAM) START(0x00020000)
END(0x000200ff) WIDTH(48) }

seg_init { TYPE(PM RAM) START(0x00020100)
END(0x000217ff) WIDTH(48) }

seg_pmco { TYPE(PM RAM) START(0x00021800)
END(0x00025fff) WIDTH(48) }

alt_pmco { TYPE(PM RAM) START(0x00030000)
END(0x00033fff) WIDTH(48) }

seg_dmda { TYPE(DM RAM) START(0x00036000)
END(0x00037fff) WIDTH(32) }

seg_heap { TYPE(DM RAM) START(0x00038000)
END(0x00039fff) WIDTH(32) }

seg_stak { TYPE(DM RAM) START(0x0003a000)
END(0x0003dfff) WIDTH(32) }

mem_dmda { TYPE(DM RAM) START(0x0007c000)
END(0x0007ffff) WIDTH(16) }
```

Next, we make a small modification in the SECTIONS portion of the LDF file. This change will allow us to place the object file created by the compiler from the file **ext.c** (called **ext.doj**) in the alternate memory segment **alt_pmco** that we just created. Keep in mind that we still want the main code module to reside in **seg_pmco.** The boldfaced text shows the changes to the LDF.

```
PROCESSOR p0
{
LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST)
OUTPUT( $COMMAND_LINE_OUTPUT_FILE)
SECTIONS
    {
dxe_seg_rth
{
INPUT_SECTIONS( $OBJECTS(seg_rth)
$LIBRARIES(seg_rth))
} > seg_rth

dxe_seg_init
{
INPUT_SECTIONS( $OBJECTS(seg_init)
$LIBRARIES(seg_init))
} > seg_init

dxe_seg_pmco
{
INPUT_SECTIONS( main.doj(seg_pmco)
$LIBRARIES(seg_pmco))
} > seg_pmco

dxe_alt_pmco
{
INPUT_SECTIONS( ext.doj(seg_pmco))
} > alt_pmco

dxe_alt_dmda
```

```
{
INPUT_SECTIONS( $OBJECTS(alt_dmda))
} > mem_dmda

dxe_seg_dmda
{
INPUT_SECTIONS( $OBJECTS(seg_dmda)
$LIBRARIES(seg_dmda))
} > seg_dmda
```

Here we made two changes. Under **dxe_alt_pmco** we changed $OBJECTS (a macro that refers to all of the object files on the command line) to **ext.doj.** In addition, under **dxe_seg_pmco** we changed $OBJECTS to **main.doj.** If you build the code example with the new LDF file, **main.doj** is placed in **seg_pmco**, while **ext.doj,** is placed in **alt_pmco**.

Note that both changes had to be made – if you left the macro $OBJECTS in front of **seg_pmco**, you would find that **ext.doj** would not be placed in **alt_pmco**. Both changes have to be made in order for the linker to appropriately place all object files.

Examining the MAP file created by the linker will confirm placement of your code module **alt_pmco.**

Option #2: Using the SECTION keyword

The second way to control placement of your code or data is to use the SECTION keyword (formerly referred to as the SEGMENT keyword in previous releases of VisualDSP).

In this example, we want to place the two variables, ext_a and ext_b in an alternate memory section called **mem_dmda.** To do so, we change their declaration from

```
int ext_a=0xa, ext_b=0xb;
```

to

```
static section ("alt_dmda") int ext_a=0xa;
static section ("alt_dmda") int ext_b=0xb;
```

The updated file, **main.c**, is shown below:

```
/* main.c */
extern int sum(int a,int b);

static section ("alt_dmda") int ext_a=0xa;
static section ("alt_dmda") int ext_b=0xb;


void main(void)
{
      int i=5;
      int j=3;
      int total;
      i = sum(i,j);
      total=ext_a+ext_b;
      idle();
}
```

Using this syntax in our C code, the two variables ext_a and ext_b will now be forced into the memory segment **mem_dmda.** Note that for this to work, we need to have the following line in the SECTIONs portion of the LDF file:

```
dxe_alt_dmda
{
INPUT_SECTIONS( $OBJECTS(alt_dmda))
} > mem_dmda
```

By using the SECTION command in our C source code as shown above, we are forcing the compiler to create an input section called **alt_dmda.** The line above in the LDF file instructs the compiler to take whatever it placed in **alt_dmda** and redirect to the memory segment called **mem_dmda. Alt_dmda** is the name used in your source code and **mem_dmda** is the name of the memory segment in the MEMORY portion of the LDF file. These two names can be the same, however for clarification purposes they are shown to be different in this example.

Again, examination of the MAP file generated by the linker will confirm placement of the data variables in **mem_dmda.**

This keyword can also be used to explicitly place functions in specific memory segments (See page 2-

64 of the VisualDSP++ C Compiler and Library manual for details).

See the associated zip file for the VisualDSP++ project that contains all of the code modules described in this EE Note.

Conclusion

This EE Note described two methods of explicitly placing C code and data in specific memory segments. All of the code mentioned is included in an associated zip file. Note that the attached code example is based on the ADSP-21060 SHARC® processor, however the principles discussed in this EE note are applicable for all Analog Devices' DSP products (ADSP-21xx, 2106x, 2116x. etc).

For more information, consult the additional references mentioned below.

Additional References

1. VisualDSP++ *C Compiler Guide and Reference*
2. VisualDSP++ *Linker and Utilities Manual*
3. **EE-69***: Understanding and Using Linker Description Files.*