**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**



# Accessing Short Word Memory In C

*Last modified: July 20, 1999*

## Introduction

The ADSP-2106x SHARC family processors' C runtime environment natively processes 32-bit datatypes. The DSP hardware can be configured to process 48, 40, 32, and 16-bit data, although the C runtime environment only natively deals with 32-bit data. This EE note will explain how to place variables (and other datatypes) into 16-bit, or short-word memory, to allow for a more efficient use of memory and allows for tighter granularity of the data as well.

As mentioned earlier, short word memory addressing is used to access 16-bit data located in the internal memory of the SHARC. The short word address space is an alias of the normal word address space. Each single 32-bit memory address in normal word space becomes two 16-bit words in short word address space. See page 5-12 of the SHARC User's Manual for the respective memory map. Each memory address in Normal word memory space can be accessed as two short words by doubling the address. For example, say we had a value of 0x12345678 stored at memory address 0x28000. This is the first location of memory block 1 on a 21062. A read memory location 0x56000 would return the LSW of the value stored at 0x28000 (0x5678) and an access to location 0x56001 would return the MSW of the value stored at 0x28000 (0x1234). For more information on this concept, read sections 5.2 and 5.3.4 of the SHARC User's Manual.

The C compiler does not directly support short word memory space, so we need to take care of a few things manually. Fortunately for us, the development tools for the SHARC family compiler allow us to place variables (and other datatypes) into whichever memory segment you choose.

## VisualDSP Development Tools Example

The VisualDSP development tools allow you to easily place variables into an explicit memory segment that you declare in your linker description file (LDF), using the "segment" qualifier. For example, in your C source, you would include the following statement:

```
Static segment("alt_dmda") int
```

The above statement will tell the compiler (and linker) to place the array labeled "shortArray" into the memory segment "alt_dmda" which is declared in your LDF file.

## Setting Up Your LDF

Setting up alternate segments in your LDF file is a very straightforward procedure. Here is an excerpt from the LDF used for this EE note (a complete version of this file and all of the sources used for the project are included in the appendix at the rear of this document.)

```
MEMORY
{
.
.
.
alt_dmda { TYPE(DM RAM) START(0x0007c000) END(0x0007ffff)
WIDTH(16) }

PROCESSOR p0
{
   LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST)
   OUTPUT( $COMMAND_LINE_OUTPUT_FILE )

   SECTIONS
   {
        .
        .
        .
        alt_dmda
        {
            INPUT_SECTIONS( $OBJECTS(alt_dmda) )
        } >alt_dmda
        .
        .
        .
```

From this LDF example, we see that our segment is declared in a short-word segment in the "memory" field of command specifies the physical memory of the target system, with the *"sections"* portion of the LDF corresponding to each memory segment declared. For more information on these linker description file commands, please refer to pages 5-33

through 5-35 of the VisualDSP User's Guide & Reference Manual, first edition.

**V3.3 210xx Family Development Tools Example**

For the v3.3 version of our development tools, the placement of variables into alternate memory segments is performed in a different manner. Unfortunately for us, the *"static segment"* qualifier which we used for the VisualDSP development tools example is not available for the v3.3 tools. To accommodate this, we therefore need to do things in a more tedious manner. There is one main difference to our approach here; we must compile the source file separately to place it in the desired segment by using the "-mpmdata=" compiler switch. For more information on this compiler switch, please refer to sections 3.2.5 and 3.2.6, on pages 3-8 and 3-9 of the ADSP-21000 Family C Tools Manual, third edition.

```
g21k short-33.c -c -mpmdata=alt_dmda -a 21060c.ach -save-temps -g
```

For the above example, we see that we have compiled our source file named short-33.c, which will reside in the segment named "alt_dmda" which we have declared in our architecture file. Here is an excerpt of this file below:

```
.SEGMENT/RAM/BEGIN=0x0007c000 /END=0x0007ffff /DM/WIDTH=16
alt_dmda;
```

(For our configuration, our short-word memory space begins at 2*0x3E000 and ends at 2*0x3FFFF, or 0x7C000 to 0x7FFFF.)

Also note the inclusion of the *"-c"* compiler switch. This switch stops the compiler from invoking the linker, so only an object file is created, not an executable. This important step is needed because we need to compile our different segment's sources separately and then link them all together to create the final executable. A complete example will be provided in the appendix at the rear of this EE note.

**Manipulating Signed 16-bit Data**

If our 16-bit data is signed data, we can use the Short Sign Extend mode of the SHARC which will sign-extend our 16-bit values when they are loaded into 32-bit/40-bit the register file.

```
asm("bit set MODE1 SSE:");
```

One important statement to mention here is that the SHARC family DSP processors work natively with 32-bit datatypes. When using short-word memory accesses, you must ensure that you deal with your signed data values accordingly. Writes from the register file to 16-bit memory work as desired (whether the data is a negative or a positive value). Reading a negative value from memory into the register file should be sign-extended to retain the sign information otherwise undesired calculations and values will result.

# Appendix A: VisualDSP Example

```
// Link for 21060 system
ARCHITECTURE(ADSP-21060)
SEARCH_DIR( $ADI_DSP\21k\lib )
MAP (SUM.map)

// The lib060.dlb must come before libc.dlb because libc.dlb has some 21020 specific code and data
$LIBRARIES = lib060.dlb, libc.dlb;

// Libraries from the command line are included in COMMAND_LINE_OBJECTS.
$OBJECTS = 060_hdr.doj, $COMMAND_LINE_OBJECTS;

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

PROCESSOR p0
{
   LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST)
   OUTPUT( $COMMAND_LINE_OUTPUT_FILE )

   SECTIONS
   {
                // .text output section
                seg_rth
                {
                        INPUT_SECTIONS( $OBJECTS(seg_rth) $LIBRARIES(seg_rth))
```

**Example LDF file for VisualDSP Example**

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

seg_pmco
{
        INPUT_SECTIONS( $OBJECTS(seg_pmco) $LIBRARIES(seg_pmco))
} >seg_pmco

alt_pmco
{
        INPUT_SECTIONS( $OBJECTS(seg_pmco) )
} >alt_pmco

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

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

stackseg
{
        // allocate a stack for the application
        ldf_stack_space = .;
        ldf_stack_length = MEMORY_SIZEOF(seg_stak);
} > seg_stak

heap
{
        // allocate a heap for the application
        ldf_heap_space = .;
        ldf_heap_end = ldf_heap_space + MEMORY_SIZEOF(seg_heap) - 1;
        ldf_heap_length = ldf_heap_end - ldf_heap_space;
} > seg_heap
```

**Example LDF file for VisualDSP example (ontinued)**

```
/* Short-Word memory space in C example for VisualDSP development tools */

static segment("alt_dmda") int x;
static segment("alt_dmda") int shortArray[10]={1,2,3,4,5,6,7,8,9,10};
static segment("alt_dmda") int total;

void main (void){

        int i;

        x=5;
        total=0;

        for(i=0; i<10; i++)
                total+=shortArray[i];

        total+=x;

        asm("idle;");

} /* end of main */
```

**Example source code for VisualDSP short-word memory example**

# Appendix B: v3.3 Development Tools Example

```
.SYSTEM              Example_ADSP21060_C_Architecture;
.PROCESSOR = ADSP21060;



!  ---------------------------------------------------------------
!  Internal memory Block 0
!  ---------------------------------------------------------------

.SEGMENT/RAM/BEGIN=0x00020000 /END=0x000200ff /PM/WIDTH=48       seg_rth;
.SEGMENT/RAM/BEGIN=0x00020100 /END=0x000217ff /PM/WIDTH=48       seg_init;
.SEGMENT/RAM/BEGIN=0x00021800 /END=0x00025fff /PM/WIDTH=48       seg_pmco;

!  ---------------------------------------------------------------
!  Internal memory Block 1
!  ---------------------------------------------------------------

.SEGMENT/RAM/BEGIN=0x00030000 /END=0x00033fff /PM/WIDTH=48       alt_pmco;
.SEGMENT/RAM/BEGIN=0x00036000 /END=0x00037fff /DM/WIDTH=32       seg_dmda;
.SEGMENT/RAM/BEGIN=0x00038000 /END=0x00039fff /DM/WIDTH=32 /cheap seg_heap;
.SEGMENT/RAM/BEGIN=0x0003a000 /END=0x0003dfff /DM/WIDTH=32       seg_stak;
.SEGMENT/RAM/BEGIN=0x0007c000 /END=0x0007ffff /DM/WIDTH=16       alt_dmda;

.ENDSYS;
```

**Example Architecture File for v3.3 Development Tools**

```
g21k main.c -c -a 21060c.ach -save-temps -g
g21k short-33.c -c -mpmdata=alt_dmda -a 21060c.ach -save-temps -g
g21k main.o short-33.o -a 21060c.ach -o short-33.exe -map -g -save-temps
```

**Example batch file for v3.3 development tools**

Here we see from the above batch file, that the build process must be performed in multiple stages. The code and data segments that are to reside in alternate memory segments other than the default seg_pmco and seg_dmda segments, must be compiled and linked separately. The use of the "-c" compiler switch stops the compiler before the linking stage for each source file. Also, note the use of the "-mpmdata=" compiler switch, which tells the linker which alternate segment this code module will reside. (For more information, please refer to pages 3-8 and 3-9 of the "ADSP-21000 Family C Tools Manual", third edition.

```
/* Main.c: Short-Word memory space in C example for v3.3 development tools*/


extern int x;
extern int shortArray[];
extern int total;


void main (void){
        int i;


        x=5;
        total=0;


        for(i=0; i<10; i++)
                total+=shortArray[i];


        total+=x;
        asm("idle;");


} /* end of main */
```

**Main source file (Main.c) for v3.3 development tools**

```
/* Short-33.c: Short-Word memory space in C example for v3.3 development tools*/
int x;
int shortArray[10]={1,2,3,4,5,6,7,8,9,10};
int total;
```

**Short-word memory segment file (Short-33.c) for v3.3 development tools**