Notes on using Analog Devices' DSP, audio, & video components from the Computer Products Division
Phone: (800) ANALOG-D or (617) 461-3881, FAX: (617) 461-3010, EMAIL: dsp_applications@analog.com

Boot Paging I: FAQ : Boot Pages on the 21xx Family DSP's (excluding 218x and CSP01).

Last Modified: *7/21/1997*

Introduction

This Engineer's Note addresses commonly asked questions about the boot page capabilities of the ADSP-21xx family Digital Signal Processors (DSP). The questions cover the basic concepts of boot pages, hardware and software implementation of boot pages, and programming techniques for effective use of boot pages while programming in both 21xx assembly language and C.

What are Boot Pages?

Boot Pages provide a means of executing programs larger that require more memory than the internal Program Memory (PM) of the DSP. A large program can be segmented into a maximum of 8 parts or "pages", and stored in an external memory device (i.e. EPROM, RAM, etc.) connected to the DSP. Individual pages can then be automatically loaded into PM at any time through simple software functions.

For example, a control system that uses an ADSP-2103 varies the speed of a motor based on the data received from an external probe. The program driving the DSP performs test functions on the system and does a wide variety of signal analysis on the incoming data. The code for this program is much larger than the internal PM of the 2103, so it is divided into 6 pages or segments of code. The first page (page zero) is automatically loaded at RESET and contains boot and test code, as well as instructions to load the second page into PM. The second page then retrieves data from probe and loads either the third or the fifth page into PM, depending on the instructions and data the 2103 received.

How are boot pages loaded from an external memory device into Program Memory?

There are two ways that boot pages are loaded into PM: asserting the MMAP pin (logical zero) on the DSP while resetting the DSP, and setting the BFORCE field of the SYSCON (DSP System Control Register) to a logical one. The first method always loads boot page zero into PM upon chip reset. The second method occurs at any time and is initiated entirely through software. On the processor cycle following the assertion of the BFORCE bit of SYSCON, the boot page designated by the 3-bit BPAGE field of SYSCON is automatically loaded into PM and begins executing as soon as it is finished loading.

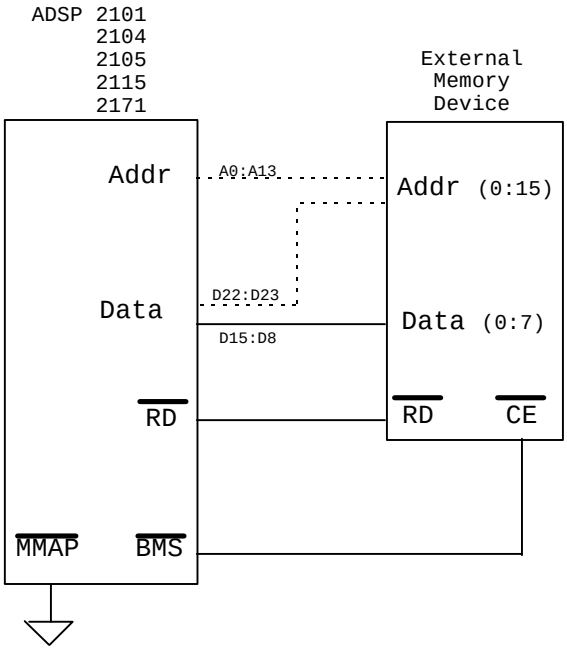
What types of external memory devices can be used and how do they connect to the ADSP-2100 family?

The most commonly used memory device for this application is the EPROM (2764, 27128, 27256, 27512). However, any type of memory device can be used as long as it is fast enough and compatible with the address bus, data bus and memory control signals of the ADSP-2100 family. This includes flash memory, dual-ported RAM, or another processor.

The ADSP-2100 family DSPs can access up to 64 Kbytes of external boot page memory. This block of memory is segmented in eight pages. Each page is a maximum of 8 Kbytes long and a minimum of 32 bytes long. The memory must also be at least 8-bits wide (memory wider than 8 bits can be used but the extra bits are discarded).

64 Kbytes of memory require 16 address bits to decode. The ADSP-2100 external address bus is only 14 bits wide so 2 bits are taken from bits 22 and 23 of the external data bus. The external data bus connects to the most significant address bits of the external memory device.

The figure below shows how an external memory device is connected to the DSP.



The number of wait states that the external memory device requires is defined in the 3-bit BWAIT field of SYSCON. This value defaults to 3 (7 for the ADSP 217x and msp5x).

How is the 24-bit program data stored in the 8-bit external memory?

Each 24-bit instruction takes up 4 locations of the external 8-bit memory leaving one 8-bit byte unused. In the first word of each page this fourth byte contains the page length. The page length is calculated with the following formula:

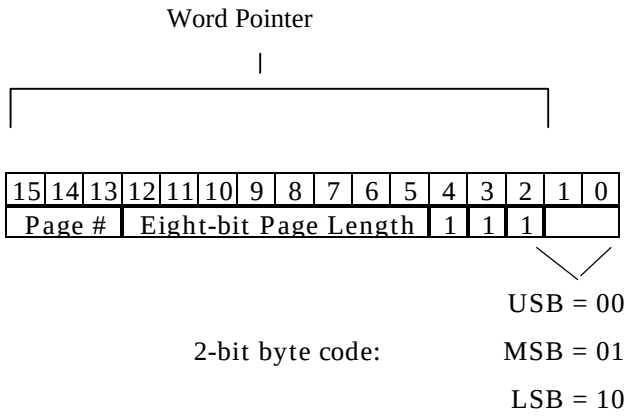
Page Length = (number of 24bit PM words / 8) - 1

Below is a boot page memory map demonstrating this allocation where:

USB = Upper Significant Bits (23-16)
MSB = Middle Significant Bits (15-8)
LSB = Low Significant Bits (7-0)

Address	Contents
0000	Word 0 : USB
0001	Word 0 : MSB
0002	Word 0 : LSB
0003	Page Length
0004	Word 1 : USB
0005	Word 1 : MSB
0006	Word 1 : LSB
0007	unused
0008	Word 2 : USB
0009	Word 2 : MSB

The 16 bits of the boot page memory address are defined as:



The "word pointer" is an internal variable that points to the base address of the word that the DSP is loading

What happens when the processor loads a boot page from external memory?

On the processor cycle following the assertion of the BFORCE bit of SYSCON, *or* when the processor is reset with the MMAP pin asserted (logical zero), the processor begins loading the boot page byte-by-byte from the external memory device as follows:

1. The processor first reads from boot-memory location 0x0003 which holds the page length. The page length = (number of 24-bit PM words in current page/8)-1.
2. The processor uses the page length to compute the address of the last PM word in the current boot-page. It then initializes the internal word-counter with the starting boot-memory address of this 4-byte word.
3. The processor loads upper byte of the word pointed to by the word-counter.
4. The processor loads the lower byte of the word pointed to by the word-counter.
5. The processor loads the middle byte of the word pointed to by the word-counter.
6. The processor decrements the internal word counter to the next block of 4 bytes.
7. The processor repeats steps 3 through 6 until all words are loaded.

How is boot paging implemented in 21xx assembly language?

Boot page implementation in assembly is quite simple. The first step is to break the code into logical pages. Ideally, a paged program is optimized in such a way that it reloads PM as few times as possible. This is because it takes several instruction cycles to load a new program from the external memory device into the processor's PM.

Once the program has been segmented, each page is treated as a standalone program. Because of this, each page requires the following assembly code:

- A `.module/ram/boot=`*n*`/abs=0`

This as the first statement where n is the designated boot page number for the specific page (0-7).

- All `#include` statements required for the code in the given boot page.
- All system constant definitions

This is done by including a `.h` file that contains these declarations in each page (see the listing of misc.h below and mem-x.h on the following page).

- An interrupt vector table with the reset vector pointing to the first line of your code.
- An .endmod statement at the end of the code.

The next step is to add statements within the pages that load other pages. Loading a new page into program memory requires specifying the page number in the BPAGE field of SYSCON and setting the BFORCE bit of SYSCON. The page load begins immediately. To facilitate this process, you can define constants that are loaded into the SYSCON which load any given page. This is done by creating a `.h` file to be #included in each page. The following is an example of this:

```
{misc.h  : Boot page constants}
.const Now_Boot_Page_0 = B#0000001000011001;
.const Now_Boot_Page_1 = B#0000001001011001;
.const Now_Boot_Page_2 = B#0000001010011001;
.const Now_Boot_Page_3 = B#0000001011011001;
.const Now_Boot_Page_4 = B#0000001100011001;
```

bit 9 (BFORCE)

bits 6-8 (BPAGE)

Notice the BFORCE bit (bit 9) is set to logical one for each page and that bits 6 through 8 define the page to jump to)

If every page includes this file (`#include <misc.h>`), a boot page load assembly program can be as follows:

```
AX0 = dm(Now_Boot_Page_4);
DM(SYS_CTRL_REG) = AX0;
```

You can take this a step further by creating a routine within the page to load another page:

```
jump_to_page_3:
      AX0 = dm(Now_Boot_Page_3);
      DM(SYS_CTRL_REG) = AX0;
```

For example, if page 3 is loaded into program memory and you want to load page 4, the Z bit of the ALU should be set after a numerical operation using the following statement:

```
      if EQ jump jump_to_page_3;
```

If the z bit is set, the processor jumps to the above routine and loads page 3 into PM. After page 3 loads, the processor begins execution where the reset vector in the interrupt table of page 3 points.

If the pages of code share variables and/or arrays (i.e. an array of filter coefficients in data memory), they must be defined as static in page 0 and referenced as external variables in every other page where they are used. For example, suppose every page requires access to a 32 element array located in data memory as well as three variables that are also located in data memory. You need to create two `.h` files: one that defines the array and variables and is included in page 0 and the second that defines the array and variables as externals and is included in pages 1 through 7. The first one looks like this:

```
{Mem-g.h   -   global   memory/constant
definitions}
.var/dm/static          array[32];
.var/dm/static          variable_1;
.var/dm/static          variable_2;
.var/dm/static          variable_3;
.GLOBAL           array;
.GLOBAL           variable_1;
.GLOBAL           variable_2;
```

```
.GLOBAL          variable_3;
```

Note: variables may also be stored in PM. It is important that variables are allocated in such a way that they are not overwritten during page loads as in the following code example:

```
var/pm/abs=0x3000/static   array[32];
```

As the longest page is less than 0x3000 words, this array is safe.

Page 0 includes the following statement:

```
#include    <Mem-g.h>
```

The .h file (included in pages 1 through 7) follows:

```
{Mem-x.h - external definitions}
.EXTERNAL   array;
.EXTERNAL   variable_1;
.EXTERNAL   variable_2;
.EXTERNAL   variable_3;
```

Pages 1 through 7 include the statement:

```
#include    <Mem-x.h>
```

It is useful in some applications to define global variables that hold the previous page information (corresponding SYSCON values) so that pages are implemented as subroutines. For example, you can call page 5 from any page; when page 5 is finished executing, it knows which page to reload based on the value of the global variable holding the previous page information. Again, you can define the variable in the Mem-g.h file as:

```
.var/dm/static previous_page;
```

and in the Mem-x.h file as:

```
.EXTERNAL  previous_page;
```

Conclusion

Boot-paging provides a simple and efficient way to execute blocks of code larger than the internal program memory of the DSP. When implemented, boot pages are virtually transparent to the programmer and to the DSP system.