Chapter 6 - Memory

RAM, Flash Memory and Others
Strings.c Example

/*
** Strings
*/
#include <p32xxxx.h>
#include <string.h>    // provides string functions

// 1. variable declarations
//    a[] is a constant array
//    b[] is a variable array
//    Both a[] and b[] are initialized with values
const char a[] = "Exploring the PIC32";
char b[100] = "Initialized";

// 2. main
main()
{
    strcpy(b, "MPLAB C32");    // assign a new value to b[]
}
Memory Space Allocation

- The linker decides where the variables are to be placed in memory.
- The linker also finds a physical address for every "symbol" to be placed in memory.
- Given `char s[ ] = “Exploring the PIC32”;`
  - It reserves a contiguous block of 20 bytes in the RAM for `s[ ]`, a space called `.data`
  - It stores the value “Exploring the PIC32\0” in a 20-byte-long table in the Flash Memory, in a space called the `.rodata`, or read-only data.
Memory Space Allocation

- The X/C32 compiler creates and inserts the `crt0` code (prologue), to be called before `main()`, to copy the initial value from the Flash Memory into RAM. This is known as the “initialization of s[ ]” step.
- Thus, value stored in flash program memory is copied into `s[ ]` in the RAM data memory, i.e. memory twice the size of `s[ ]` is used.
- But, if we use the keyword `const`, then the linker will NOT reserve any RAM space.
- In `strcpy(s, “HELLO”);`, “HELLO” is const, i.e. it will be in `.rodata` of the flash memory.
Memory Space Allocation

- Now look at the code in Strings.c:
- a[] is “already initialized” (right after the Project was built). But b[] remains empty and not “initialized” until after the reset.
- Watch Properties show that a[] lives in the Program memory, while b[] in File Register, i.e. the RAM.
- Since b[] is in RAM, the crt0() code must be executed first before b[] is initialized!
- Now look at the Disassembly Listing.
Disassembly of Strings.c

- Before reset, where is the green PC pointer?
- After reset, where is the green PC pointer?
- What happened to b[] before and after reset?
- Do you see the strcpy.S code?
- Do you see the crt0.S code?
- What other .S code do you see?
- Actually, all this lengthy code has already been optimized for the 32-bit bus and caches. (Bonus Q: What is PIC32’s cache structure?)
Using the Watch Window

[Diagram of Watch Window with symbols a and b]

[Table showing symbol names and values]
Setting the Watch Properties

![Watch Properties Window]

- **Symbol:**
- **Size:** 8 bits
- **Format:** ASCII
- **Byte Order:** High:Low
- **Memory:** Program
String “b” after crt0.S initialization

![Watch window displaying the string 'Init b'

- Symbol Name: 'a', 'b'
- Value: 0x49, 0x6E, 0x69, 0x74, 0x61
- Char: 'I', 'n', 'i', 't', 'a'

Di Jasio - Programming 32-bit Microcontrollers in C
Disassembly Listing

14:      // 2. main program
15:      main()
16:      {
9D000018 27BDFEE8  addiu     sp,sp,-24
9D00001C AFBF0014  sw        ra,20(sp)
9D000020 AFBE0010  sw        s8,16(sp)
9D000024 03A0F021  addu      s8,sp,zero
17:      strcpy( b, "MPLAB C32");  // assign new content to b
9D000028 3C02A000  lui        v0,0xa000
9D00002C 24440000  addiu      a0,v0,0
9D000030 3C029D00  lui        v0,0x9d00
9D000034 2445074C  addiu      a1,v0,1868
9D000038 0F400016  jal        0x9d000058
9D00003C 00000000  nop
18:      } // main
9D000040 03C0E821  addu      sp,s8,zero    // asm code for strcpy()
9D000044 8FBF0014  lw         ra,20(sp)
9D000048 8FBE0010  lw         s8,16(sp)
9D00004C 27BD0018  addiu      sp,sp,24
9D000050 03E00008  jr         ra
9D000054 00000000  nop
Memory Map, e.g. Strings.map

- The memory map consists of 3 major parts:
  - List of Included Archive Members (.o and .a files)
  - Memory Configuration Table (locations and sizes of all memory areas, hardware-dependent)
  - Linker Script and Memory Map (longest of the 3, every section gets placed in mem areas by linker)
  - .reset section: for reset code
  - .vector_x section(s): for ISR code, up to 64 sections
  - .startup section: for crt0 code
  - .text section(s): for user code, including main()
  - .rodata section: for (global) constants
  - .data section: for global variables
  - .data1 section: for values used to initialize variables
  - *(.data1): a pointer (address) to .data1 section
Looking at the “Map”

```
C:/Program Files/Microchip/..../lib\libc.a(strcpy.o)
Strings.o (strcpy)
```

### Memory Configuration

<table>
<thead>
<tr>
<th>Name</th>
<th>Origin</th>
<th>Length</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>kseg0_program_mem</td>
<td>0x9d000000</td>
<td>0x00080000</td>
<td>xr</td>
</tr>
<tr>
<td>kseg0_boot_mem</td>
<td>0x9fc00490</td>
<td>0x00000970</td>
<td></td>
</tr>
<tr>
<td>exception_mem</td>
<td>0x9fc01000</td>
<td>0x00001000</td>
<td></td>
</tr>
<tr>
<td>kseg1_boot_mem</td>
<td>0xbf000000</td>
<td>0x00000490</td>
<td></td>
</tr>
<tr>
<td>debug_exec_mem</td>
<td>0xbf02000</td>
<td>0x00000ff0</td>
<td></td>
</tr>
<tr>
<td>config3</td>
<td>0xbf02ff0</td>
<td>0x00000004</td>
<td></td>
</tr>
<tr>
<td>config2</td>
<td>0xbf02ff4</td>
<td>0x00000004</td>
<td></td>
</tr>
<tr>
<td>config1</td>
<td>0xbf02ff8</td>
<td>0x00000004</td>
<td></td>
</tr>
<tr>
<td>config0</td>
<td>0xbf02ffc</td>
<td>0x00000004</td>
<td></td>
</tr>
<tr>
<td>kseg1_data_mem</td>
<td>0xa0000000</td>
<td>0x00008000</td>
<td>w !x</td>
</tr>
<tr>
<td>sfrs</td>
<td>0xbf800000</td>
<td>0x00100000</td>
<td></td>
</tr>
<tr>
<td><em>default</em></td>
<td>0x00000000</td>
<td>0xffffffff</td>
<td></td>
</tr>
</tbody>
</table>

Di Jasio - Programming 32-bit Microcontrollers in C
Memory Map Sections

- **.reset section**, containing the code that will be placed by the linker at the reset vector. This is normally filled with a default handler `_reset()`.
  
  ```
  reset  0xbfc00000  0x10  C:/.../pic32mx/lib/crt0.o
  0xbfc00000  _reset
  ```

- **.vector_x sections**, there are 64 of them each associated to the corresponding interrupt handler. They will be empty unless your program is using the specific interrupt handler.

  ```
  .vector_0  0x9fc01200  0x0
  ```

- **.startup section**, where the crt0 initialization code is placed.

  ```
  .startup  0x9fc00490  0x1e0  C:/.../lib/crt0.o
  ```

- **.text sections**, you will find many of them, where the code generated by the MPLAB X/C32 compiler from your source files is placed. Here is the code produced for the `main()`:

  ```
  .text  0x9d000018  0x40  Strings.o
  0x9d000018  main
  ```

- **.rodata section**, where read-only (constant) data is placed in flash program memory. Here we can find space for our constant string “a” for example:

  ```
  .rodata  0x9d0000738  0x20  Strings.o
  0x9d0000738  a
  ```

- **.data section**, where RAM is allocated for global variables.

  ```
  .data  0xa0000000  0x64  Strings.o
  0xa0000000  b
  ```

- **.data1 section**, where the initialization value, ready for the crt0 code to load into the “b” variable is placed, once more, in program memory space.

  ```
  *(.data1)  0x9d000076c  _data_image_begin=LOADADDR(data)
  ```
## The Memory Window

<table>
<thead>
<tr>
<th>Address</th>
<th>00</th>
<th>04</th>
<th>08</th>
<th>0C</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D00_0760</td>
<td>9D0003AC</td>
<td>9D0004F4</td>
<td>9D000578</td>
<td>74696E49</td>
<td>........ x...Init</td>
</tr>
<tr>
<td>1D00_0770</td>
<td>696C6169</td>
<td>0064657A</td>
<td>00000000</td>
<td>00000000</td>
<td>ialized. ........</td>
</tr>
</tbody>
</table>
Pointers

int *pi;          // a pointer to an integer
int i;           // index or counter
int a[10];       // array of integers

// 1. sequential access using array-indexing
for(i=0; i<10; i++)
    a[i] = i;

// 2. sequential access using a pointer
pi = a;
for(i=0; i<10; i++)
{
    *pi = i;
    pi++;
}

// Q. Which way is more efficient, 1. or 2.? (See next slide.)
Pointers

- In 1., the compiler has to multiply the value of \( i \) by `sizeof(int)` which is 4, and then add the resulting offset to the initial address of \( a[] \), which is \( a \). `sizeof()` is a subroutine.
- In 2., \( pi \) was initialized with \( a \). Then \( pi \) was dereferenced with operator `*` to be assigned the value \( i \) and then post-incremented.
- 2. can be further reduced to:
  - `for (i=0, pi=a; i<10; i++)`
  - `*pi++ = i;`
Heap

- Dynamic structures such as pointers use the heap, an area within the data memory.
- The X/C32 Linker allocates the heap in the remaining RAM memory space, above all global variables and reserved stack space.
- You can use the Project | BuildOptions | Project menu command to open the Build Options dialog box. Then select the MPLAB PIC32 Linker tab, and define your heap size.
- `ptr = malloc(sizeof(int)*64); free(ptr).`
Advanced Material

PIC32MX Fixed Translation Map, and User and Kernel Virtual Memory Maps
## The PIC32 Fixed Translation Map

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>Virtual Addresses</th>
<th>Physical Addresses</th>
<th>Size in Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Boot Flash</strong></td>
<td>0xBFC00000 - 0xBFC02FFF</td>
<td>0x1FC00000 - 0x1FC02FFF</td>
<td>12 KB</td>
</tr>
<tr>
<td><strong>Program Flash(^1)</strong></td>
<td>0x0BD000000 + BMXPUPBA</td>
<td>0x01D000000 + BMXPUPBA</td>
<td>BMXPUPBA</td>
</tr>
<tr>
<td><strong>Program Flash(^2)</strong></td>
<td>0x09D000000 + BMXPUPBA</td>
<td>0x01D000000 + BMXPUPBA</td>
<td>BMXPUPBA</td>
</tr>
<tr>
<td><strong>RAM (Data)</strong></td>
<td>0x08000000 - 0x08000000 + BMXDPBAD</td>
<td>0x0x0000000 + BMXDPBAD</td>
<td>BMXDUPBA</td>
</tr>
<tr>
<td><strong>RAM (Prog)</strong></td>
<td>0x08000000 + BMXDPBAD</td>
<td>0x08000000 + BMXDPBAD</td>
<td>BMXDPBAD</td>
</tr>
<tr>
<td><strong>Peripheral</strong></td>
<td>0x0BF800000 - 0x0BF8FFFF</td>
<td>0x01F800000 - 0x01F8FFFF</td>
<td>1 MB</td>
</tr>
<tr>
<td><strong>Program Flash</strong></td>
<td>0x07D000000 + BMXUPBA</td>
<td>0x07D000000 + BMXUPBA</td>
<td>PFM Size - BMXUPBA</td>
</tr>
<tr>
<td><strong>RAM (Data)</strong></td>
<td>0x07F000000 + BMXUPBA</td>
<td>0x07F000000 + BMXUPBA</td>
<td>BMXUPBA</td>
</tr>
<tr>
<td><strong>RAM (Prog)</strong></td>
<td>0x07F000000 + BMXUPBA</td>
<td>0x07F000000 + BMXUPBA</td>
<td>RAM Size(^3) - BMXUPBA</td>
</tr>
</tbody>
</table>

**Note:**
1. Program Flash virtual addresses in the non-cacheable range (KSEG1).
2. Program Flash virtual addresses in the cacheable and prefetchable range (KSEG0).
3. The RAM size varies between PIC32MX device variants.
Physical Addressing Space
RAM Partitioning

- Bus matrix RAM partitioning

- Kernel RAM (Data)  Kernel RAM (Prog)
  - 0x00000000  BMXKDDBA  BMXDDDBA

- User RAM (Data)  User RAM (Prog)
  - 0xB0000000 + BMXDUPBA  0xB0000000 + BMXDUPBA  0xFFFFFFFF
User Mode Virtual Memory Map

- User mode virtual memory map

```
User FLASH

User RAM (Data)

User RAM (Prog)

User SPACE

Kernel SPACE

0x00000000

0x80000000

Generate an immediate exception if access is attempted!
```

Di Jasio - Programming 32-bit Microcontrollers in C