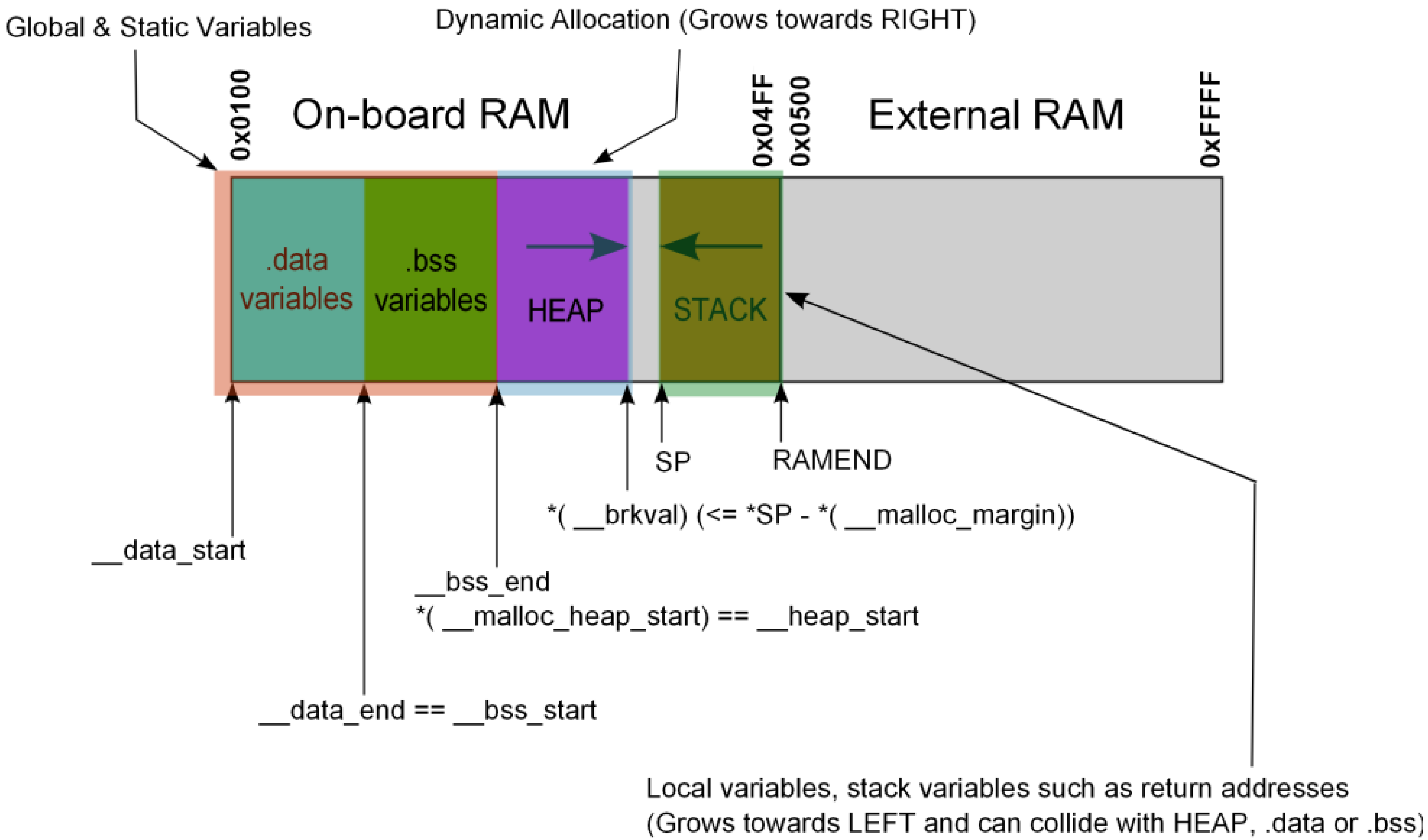
Embedded System Memory Usage

- On embedded systems, memory and storage are extremely limited
- AVR ATMega169P
  - 16 Kbytes of In-System Self-programmable Flash program memory
  - 512 Bytes EEPROM
  - 1 Kbytes Internal SRAM
- The biggest risk in memory management on embedded systems is dynamic memory allocation

ATmega169P: Data Memory Layout



Global & Static Variables

Dynamic Allocation (Grows towards RIGHT)

Local variables, stack variables such as return addresses
(Grows towards LEFT and can collide with HEAP, .data or .bss)

Storing and Retrieving Data in the Program Space

- Use the PROGMEM macro found in <**avr/pgmspace.h**> and put it after the declaration of the variable, but before the initializer
  - unsigned char mydata[11][10] PROGMEM
- Use the appropriate pgm_read_* macro
  - byte = pgm_read_byte(&(mydata[i][j]));

Avoiding Dynamic Memory Allocation

- Heap allocation, and therefore how data get laid out in memory, is difficult to predict.
- When you allocate on the heap you run the risk of memory leaks—that is, allocating memory and not freeing it.
- The memory management system adds processing and memory overhead for every allocation and de-allocation
- Bad references are more likely to happen when dealing with pointers to freed dynamic memory, and there’s no OS oversight to report segmentation violations.

Memory Usage Tracking

- When we compile a program the compiler reports how much data memory is allocated **statically**, but not how much memory will be allocated at **run-time**
- But if you tell the compiler to allocate them statically, in the .bss or .data segments, then it can include them in its memory usage calculation
- Example:

```
void do_something()
{
 char str[64];     // 64-byte string is allocated on the stack.
snprintf(str, sizeof(str), "Some text\n\r");
}
```

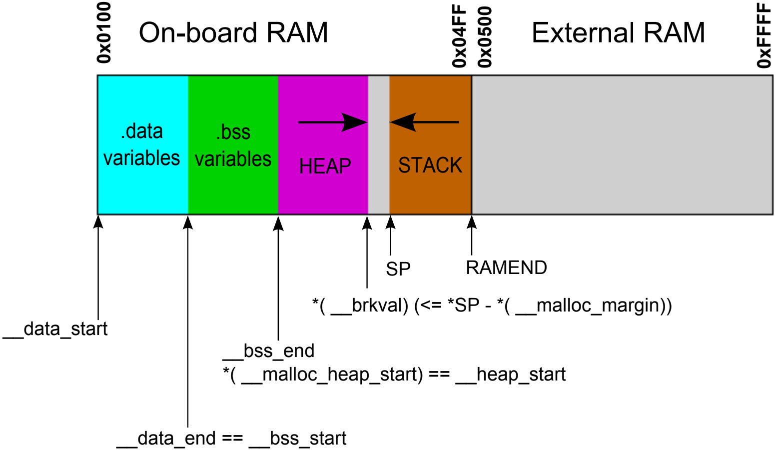
Using Memory Efficiently

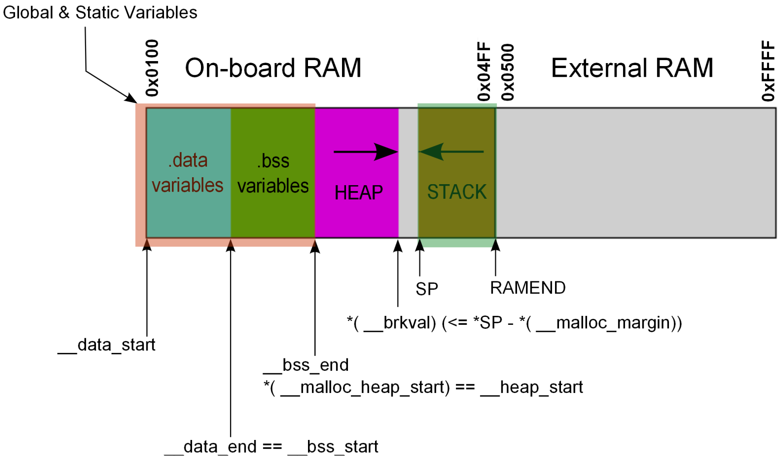
Discussion VI (Version 2.0)

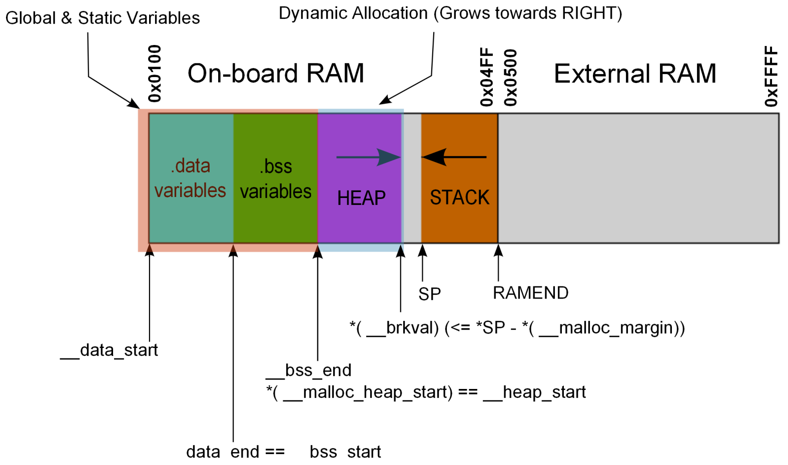
UMBC - CE

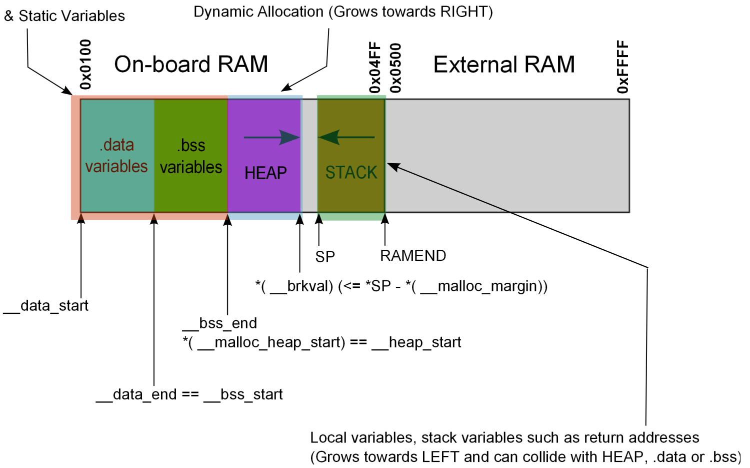
October 5, 2015

Version 1.0 - Initial Document
Version 2.0 - Fixed Typos


0x0100
On-board RAM
0x04FF
0x0500
External RAM
0xFFFF
.data variables
.bss variables
HEAP
STACK
SP
RAMEND
*(__brkval) (<= *SP - *(__malloc_margin))
__data_start
__bss_end
*(__malloc_heap_start) == __heap_start
__data_end == __bss_start


Global & Static Variables
0x0100
On-board RAM
0x04FF
0x0500
External RAM
0xFFFF
.data variables
.bss variables
HEAP
STACK
SP
RAMEND
*(__brkval) (<= *SP - *(__malloc_margin))
__data_start
__bss_end
*(__malloc_heap_start) == __heap_start
__data_end == __bss_start


Global & Static Variables
Dynamic Allocation (Grows towards RIGHT)
0x0100
On-board RAM
0x04FF
0x0500
External RAM
0xFFFF
.data variables
.bss variables
HEAP
STACK
SP
RAMEND
*(__brkval) (<= *SP - *(__malloc_margin))
__data_start
__bss_end
*(__malloc_heap_start) == __heap_start
data end == bss start



Global & Static Variables

Dynamic Allocation (Grows towards RIGHT)

*(__brkval) (<= *SP - *(__malloc_margin))

__data_start

__bss_end
*(__malloc_heap_start) == __heap_start

__data_end == __bss_start

Local variables, stack variables such as return addresses
(Grows towards LEFT and can collide with HEAP, .data or .bss)

Tips for using program space

- What happens when you run out of memory ?

Tips for using program space

- What happens when you run out of memory ?
- Data should be below 1K bytes

Tips for using program space

- What happens when you run out of memory ?
- Data should be below 1K bytes
- Shorten your prompts (reuse parts of your string)

Tips for using program space

- What happens when you run out of memory ?
- Data should be below 1K bytes
- Shorten your prompts (reuse parts of your string)
- Don’t have multiple temporary string variables in memory at the same time

Tips for using program space

- What happens when you run out of memory ?
- Data should be below 1K bytes
- Shorten your prompts (reuse parts of your string)
- Don't have multiple temporary string variables in memory at the same time
- Have gcc optimize for space

  (include **-Os** option during compilation)

Tips for using program space

- What happens when you run out of memory ?
- Data should be below 1K bytes
- Shorten your prompts (reuse parts of your string)
- Don’t have multiple temporary string variables in memory at the same time
- Have gcc optimize for space

  (include **-Os** option during compilation)
- Check your stack pointer

```
printf("sp:%d\n",SP);
```

Tips for using program space

- What happens when you run out of memory ?
- Data should be below 1K bytes
- Shorten your prompts (reuse parts of your string)
- Don't have multiple temporary string variables in memory at the same time
- Have gcc optimize for space

  (include **-Os** option during compilation)
- Check your stack pointer

```
printf("sp:%d\n",SP);
```

- Documentation for memory sections available from here

Using program space

- AVR has a library for keeping const data in program memory (flash) and accessing it directly instead of using RAM

Using program space

- AVR has a library for keeping const data in program memory (flash) and accessing it directly instead of using RAM
- Program Memory has 16KB space

Using program space

- AVR has a library for keeping const data in program memory (flash) and accessing it directly instead of using RAM
- Program Memory has 16KB space
- Example functions available in stdio.h

- printf_P
- sprintf_P
- fprintf_P
- fputs_P
- fscanf_P

Using program space

- #include <avr/pgmspace.h>

Using program space

- #include <avr/pgmspace.h>
- Declare const strings using special flag PROGMEM

```
//PROGMEM used to locate a variable in flash ROM
const PROGMEM char myString[] = "Repeated Use";
```

Using program space

- #include <avr/pgmspace.h>
- Declare const strings using special flag PROGMEM

```
//PROGMEM used to locate a variable in flash ROM
const PROGMEM char myString[] = "Repeated Use";
```

- You can create special pointers using PGM_P and access the data using a special macro **pgm_read_byte**

Using program space

- #include <avr/pgmspace.h>
- Declare const strings using special flag PROGMEM

```
//PROGMEM used to locate a variable in flash ROM
const PROGMEM char myString[] = "Repeated Use";
```

- You can create special pointers using PGM_P and access the data using a special macro **pgm_read_byte**
- Documentation is available from here

Examples

```
PGM_P progPtr;
progPtr =myString;

//Somewhere in your code

  while(pgm_read_byte(progPtr)!='\0'){

  printf("%c",pgm_read_byte(progPtr));
  progPtr++;

  }
```

Examples

```
#include <avr/pgmspace.h>
void lcd_puts_P(const char c[]) { //same const char *c
uint8_t ch = pgm_read_byte(c);
while(ch != 0) {
lcd_putc(ch);
ch = pgm_read_byte(++c);
}  }
// Usage:  Note PSTR macro which simplifies placing string
//         literals in flash ROM
//   Code: lcd_puts_P(PSTR("Hello World"));
//     Or: const PROGMEM char SOME_STRING[]  = "Repeated Use";
//         lcd_puts_P(SOME_STRING);
```