Doing More with Less

Chapter 8

Optimization

- For general computers
  - Speed
- For embedded systems
  - Code space
  - Data RAM
  - Speed
  - Power

Code Space

- Linker script
  - ld links code and data according to the specifics in the linker script
  - “Memory” : describe the location and size of blocks of memory in the target.
  - “Sections” : control exactly where input sections are placed into output sections, their order in the output file, and to which output sections they are allocated.
- Map file
  - -Wl,-Map=xxx.map

Linker Scripts

- BSS segment
  - Contains uninitialized global variables that will go in RAM.
- Data segment
  - Contains global variables that are initialized and will go in RAM.
  - May include BSS as a subsegment.
  - May include the heap and stack.
- Text segment
  - Contains code and constant data that may be put in read-only memory or in RAM.
- Vector segment
  - Contains the exception vector table to handle interrupts.

Linker Scripts

- To determine the memory layout.

```
SECTIONS
{
   /* Memory location is in Flash; place next commands at this location */
  . = 0x000000;
  Code : {
    *(.vectors)} /* Put interrupt table at very first location in memory */
    *(.text)
  }
   /* Now put everything in off-board RAM */
  . = 0x110000;
  Data : { *(.data) }
  UninitGlobals : { *(.bss) }
}
MEMORY
{
  /* Define each memory region */
  Flash (rx)  : ORIGIN = 0x000000, LENGTH = 0x10000  /* 64k */
  InRam (rwx) : ORIGIN = 0x010000, LENGTH = 0x08000  /* 32k */
  ExRam (rw)  : ORIGIN = 0x110000, LENGTH = 0x20000  /* 128k */
}
SECTIONS
{
  Code : {
    *(.vectors)}
    *(.text)
  } > Flash
  Data : { *(.
```

Code Space

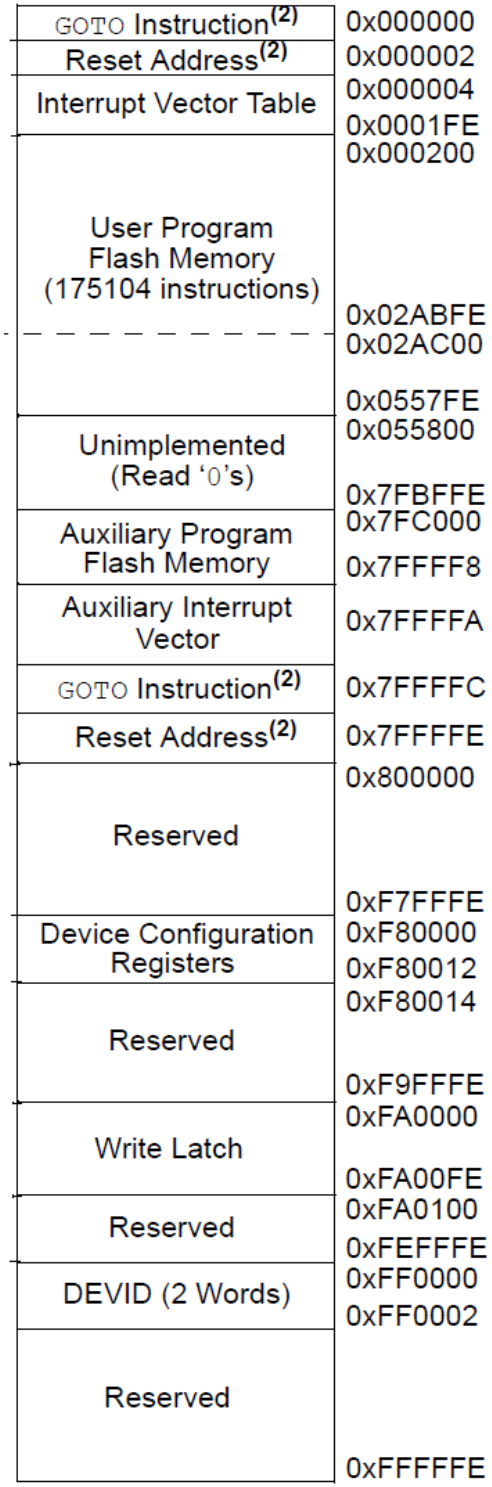
- Case 21
- Memory

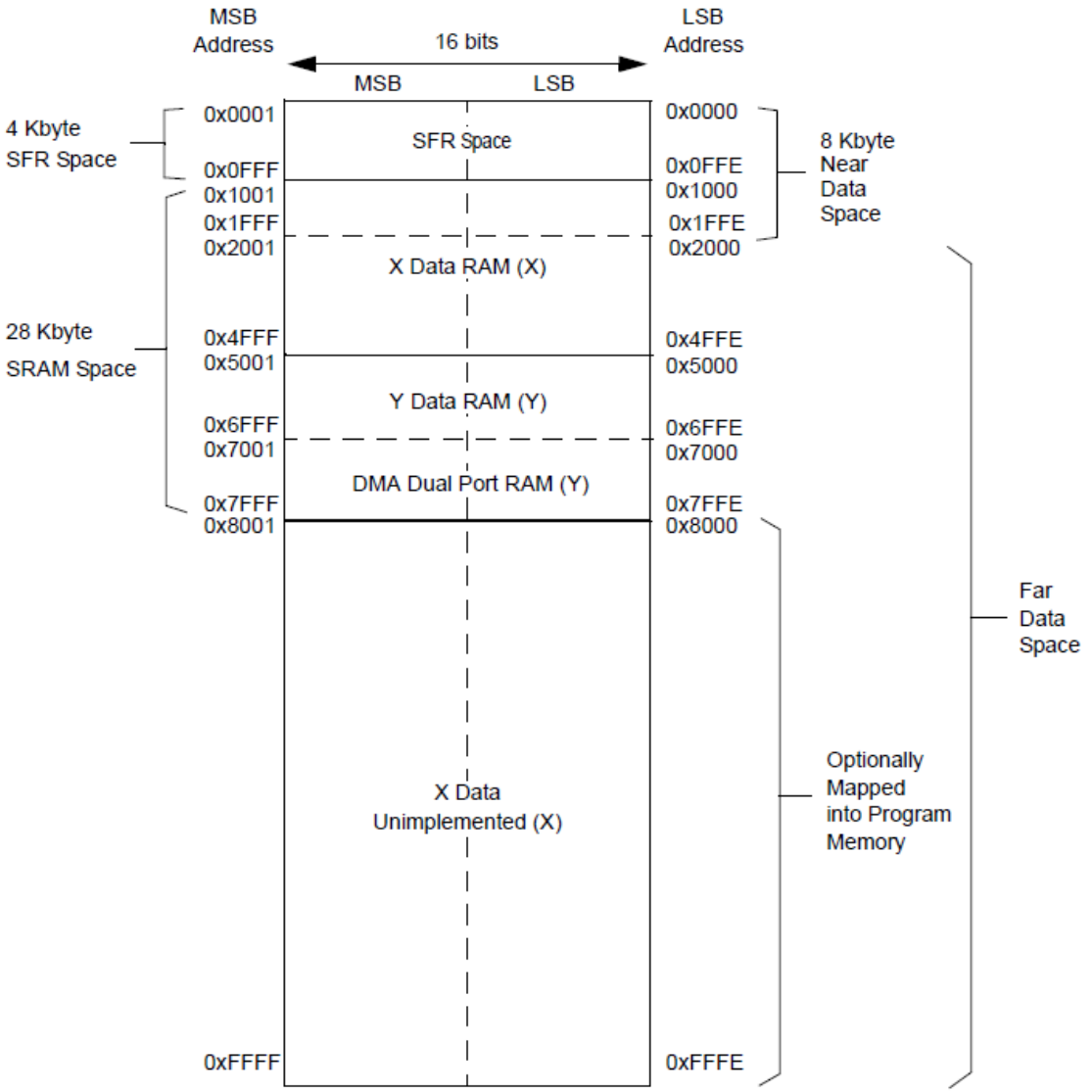
```
MEMORY {
  name(attr) : ORIGIN=xxx, LENGTH=yyy
  ...
}
```

- Sections

```
SECTIONS {
  secname addr : {
    filename( section section ... )
    ...
  } >mem
  ...
}
```

Space: Pic24ep





Optimization

- Case 22, Atmega128
- O0, O1, O2, O3, Os
- Exclude/reduce libraries that are not needed.
  - .text.libgcc.mul, .text.libgcc.div, .text.libgcc.prologue, .text.libgcc.builtins, .text.libgcc.fmul, .text.libgcc.fixed

	text	data	bss	dec	hex
O0	1586	0	0	1586	632
O1	356	0	0	356	164
O2	340	0	0	340	154
O3	340	0	0	340	154
Os	340	0	0	340	154

Function vs Macro

```
if a < b,
  if a < c, return a
  else, return c
else,
  if b < c, return b
  else, return c
```

```
#define min3(x, y, z) (((x)<(y))?(((x)<(z))?(x):(z)):(((y)<(z))?(y):(z)))
```

Implementation	1 call (diff from baseline)	2 calls	3 calls
Macro	0	76	152
Function (local or external)	20	60	96
Macro with space optimization	–40	8	56
Function with space optimization	–40	–20	0

Constants and Strings

- All constants and strings are stored in code space.
- Do not duplicate strings in code.
  - Declare char arrays for strings and use pointers for referencing the strings.
- Remove auxiliary code
  - assert, log, debug
  - Use macro to enable/disable auxiliary code

```
#define MOTOR_LOG 1 // set this to zero to turn off debugging
#if MOTOR_LOG
#define Log(level, str)            Log(eMotorSubSystem, (level), (str))
#define LogWithNum(level, str, num) LogWithNum(eMotorSubSystem, (level), (str), (num))
#else
#define Log(level, str)
#define LogWithNum(level, str)
#endif
```

Example: string

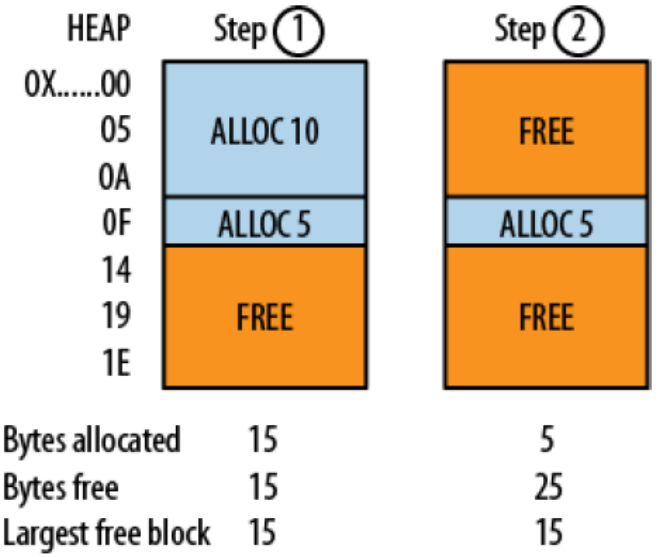
- Case33.strings

```
int main() {
  char str[]="hello world!\n";
  printf("hello world!\n");
}
```

```
int main() {
  char str[]="hello world!\n";
  printf(str);
}
```

Data RAM

- Malloc
  - Do NOT use
    - Wasted ram, lost processor cycles, fragmentation
  - Non-deterministic operation
  - Non-deterministic ram size
    - Depend on the state of heap
- Recommended alternatives
  - Declare global variables
  - Recycle buffers
  - Circular buffer
  - Chunk allocation



Example: malloc

- Return type: type pointer cast
- Params: number * sizeof(type)
- Must free after use

```
#include <stdlib.h>
int block[10];
int main() {
  int array[10];
  int* buf = (int*)malloc(10*sizeof(int));
  free(buf);
  return 0;
}
```

Data RAM

- From map to determine the size of
  - .data
  - .bss
- Stack
  - The size of stack is determined by
    - Local variables
    - Chain of function calls
  - Do not use recursion.
  - Marcos do not use stack and are faster than small functions.

Example: stack

- Function scope
  - Stack is allocated one time
- Block scope
  - Stack is adjusted for each block

```
foo() {
  int i, j;
  for (i=0; i<10; i++) ;
  return;
}
```

```
foo() {
  int j;
  for (int i=0; i<10; i++);
  return;
}
```

Registers vs Variables

- Register is faster
  - `register int i = 10;`
  - Only suggest to compiler, but not a guarantee
- Function parameters
  - Fewer than 4 parameters
  - N-bit parameters
  - Try to pass values (not pointers) to functions.
  - But, pass pointers of structures to functions.

```
int bar = 10;
foo(&bar);  /* this takes more RAM than passing by value
               because bar has to be in RAM to have an address */
```

Scope of Local Variables

- Scope: where a variable is used by the code
- Help compiler to optimize
  - Limit the scope of each variable
  - Within each scope, limit the number of variables
  - Better chances for compiler to free up registers

```
for (i=0;  i < MAX_ARRAY_LENGTH; i++) { array[i] = i; }
… /* do stuff to array, need to set it up again */
/* i still equals max array so subtract one and run through the loop again*/
for (i--; i >= 0; i--) { array[i] = i; }
```

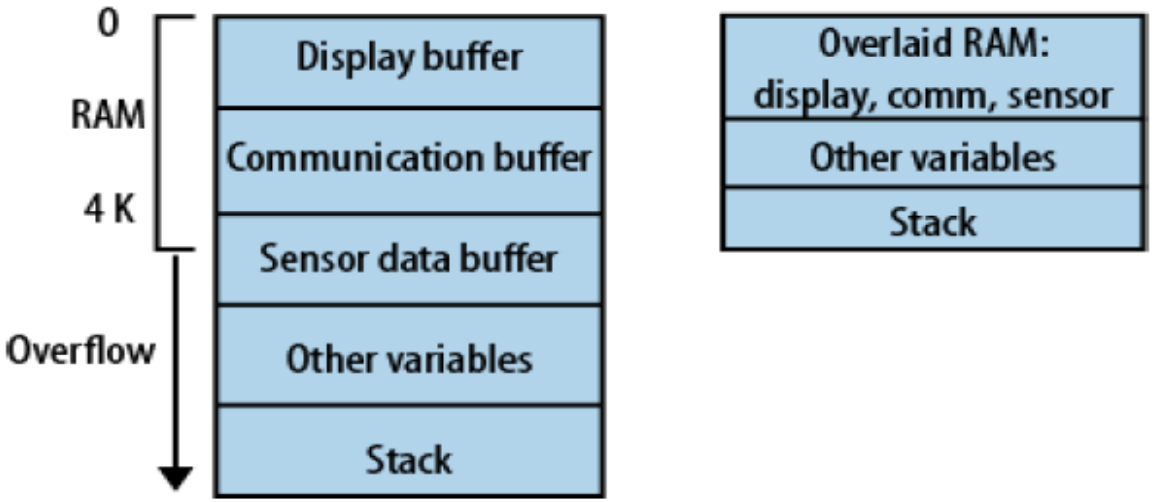
```
for (i=0;  i < MAX_ARRAY_LENGTH; i++) { array[i] = i; }
… /* do stuff to array, need to set it up again */
for (i=0;  i < MAX_ARRAY_LENGTH; i++) { array[i] = i; }
```

Global Variables

- Cons
  - Outside the flow of code
  - Reentrant
  - Cannot be stored in registers
- Pros
  - Share variables in function calls
    - Do not pass as parameters
    - Use globals

Memory Overlay

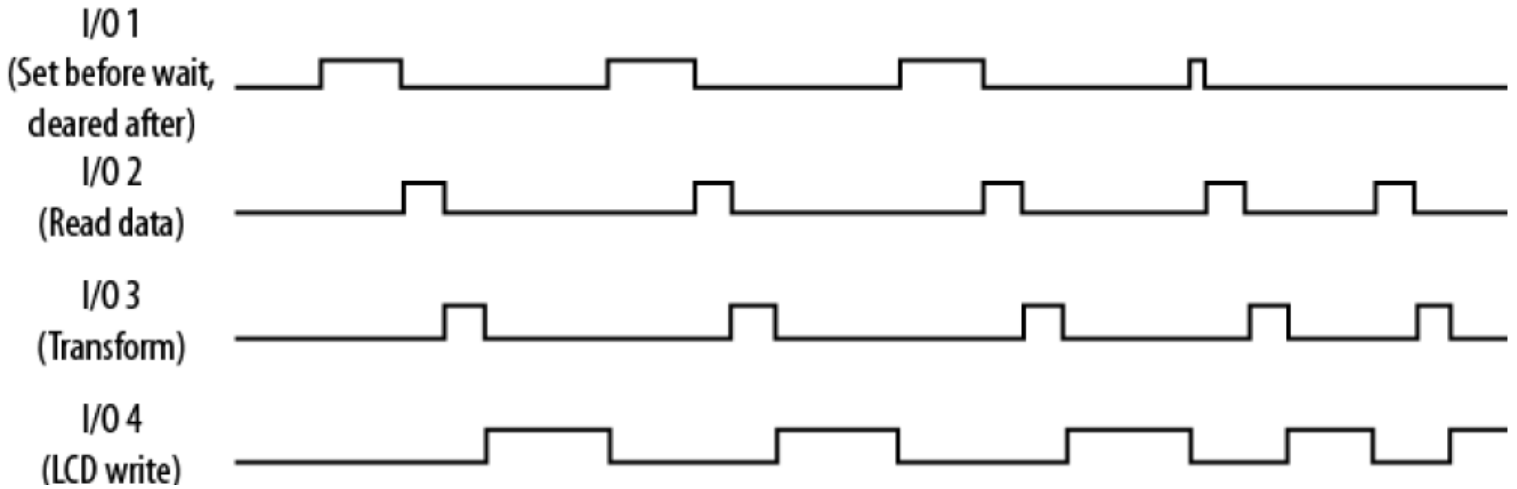
- Multiple buffers needed, but **not used at the same time**.
- Methods
  - Union of buffers
  - Linker script : memory overlay



Speed

- Profiling: to know where the cycles are going
- Profiler changes the behavior and timing of the code.
- IO lines

```
Main loop:
  Loop, waiting for data ready to be set
  Read the data in the buffer
  Transform data into information
  Write the information to the LCD
```



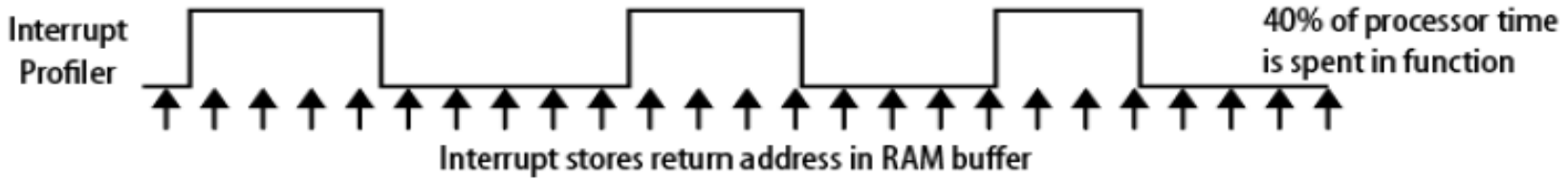
Timer

- In case, no stop watch to use
- Get time before and after the function of importance.

```
profile.count = 0;
profile.sum = 0;
while (1) {
   profile.start = TimeNow();
   ImportantFunction();
   profile.end = TimeNow();
   profile.sum += profile.end - profile.start;
   profile.count++;
   if (profile.count == PROFILE_COUNT_PRINT) {
     LogWithNum(eProfiler, eDebug, "Important Function profile: ", profile.sum);
     profile.count = 0;
     profile.sum = 0;
    }
  ... // continue with other main loop functions
 }
```

Sampling

- Procedure
  - Set a periodic timer and a block of ram
  - On each timer, save the return address of stack
  - When the ram is full, output the list of addresses
  - Figure out where the addresses are in the image using the map file
- Requirements
  - the sampling timer is the only one allowed to interrupt other interrupts



Function Reduction

```
// in Lcd.c
void LcdWriteBuffer(uint16_t* buffer, uint16_t bufLength){

  int i;
  while (bufLength) {
    LcdWriteBus(buffer[i] & 0xFF);       // write lower byte
    LcdWriteBus((buffer[i] >> 8) & 0xFF); // write upper byte
    i++;
    bufLength--;
  }
}
void LcdWriteBus(uint8_t data){

  IoClear(LCD_SELECT_N);          // select the chip
  IoWriteBusByte(LCD_BUS, data);  // write to the I/O lines
  IoSet(LCD_SELECT_N)             // deselect the chip
}

// in Io.c
IoWriteBusByte(uint32_t io, uint8_t data){

  // ioBus was configured during initialization
  ioBus[io] = data;
}
```

Function Reduction

- Reduce repeating small functions or operations

```
//in lcd.c
void LcdWriteBuffer(uint16_t* buffer, uint16_t bufLength){

  int i;
  IoClear(LCD_SELECT_N);                        // select the chip
  while (bufLength) {
    IoWriteBusByte(LCD_BUS, buffer[i]);         // write lower byte
    IoWriteBusByte(LCD_BUS, (buffer[i] >> 8)); // write upper byte
    i++;
    buffLength--;
  }
  IoSet(LCD_SELECT_N)                           // deselect the chip
}
```

Change Iterators

- Bad

```
IoWriteBusByte(LCD_BUS, buffer[i] & 0xFF); // write lower byte
```
(Variable i would already be in a register, already initialized)
Copy the buffer pointer from the stack into a register
Add the buffer pointer to i
Read the contents of memory at that address
Perform bitwise AND with contents
Put i on stack
Call IoWriteBusByte, passing data
Pop i off the stack

```
IoWriteBusByte(LCD_BUS, (buffer[i] >> 8) & 0xFF); // write upper byte
```
Copy the buffer pointer from the stack in a register
Add the buffer pointer to i
Read the contents of memory at that address
Perform shift with contents
Perform bitwise AND with result
Put i on stack
Call IoWriteBusByte, passing data
Pop i off the stack

```
i++;
```
Increment register i

Change Iterators

- Good

```
IoWriteBusByte(LCD_BUS, *buffer & 0xFF); // write lower byte
```
(Variable buffer would already be in a register, already initialized)
~~*Copy the buffer pointer from the stack in a register*~~
~~*Add the buffer pointer to i*~~
Read the contents of memory at that address
Perform bitwise AND with contents
Put buffer on stack
Call IoWriteBusByte, passing data
Pop buffer off the stack

```
IoWriteBusByte(LCD_BUS, (*buffer >> 8) & 0xFF); // write upper byte
```
~~*Copy the buffer pointer from the stack in a register*~~
~~*Add the buffer pointer to i*~~
Read the contents of memory at that address
Perform shift with contents
Perform bitwise AND with result
Put buffer on stack
Call IoWriteBusByte, passing data
Pop buffer off the stack

```
buffer++;
```
Increment register buffer

```
i++;
```
~~*Increment register i*~~

Reduce Math

- Choose a proper type/size of pointer

```
void LcdWriteBuffer(uint16_t* buffer, uint16_t bufLength)
{
  uint8_t *byteBuffer = (uint8_t *) buffer;
  // buffer is now a buffer of bytes.
  // This works only if your endian-ness matches what the hardware expects
  bufLength = bufLength*2;

  IoClear(LCD_SELECT_N);         // select the chip
  while (bufLength) {
    IoWriteBusByte(LCD_BUS, *byteBuffer);
    bufLength--; byteBuffer++;
  }
  IoSet(LCD_SELECT_N)              // deselect the chip
}
```

Unroll Loop

- Loop has overhead.

```
void LcdWriteBuffer(uint16_t* buffer, uint16_t bufLength){

  // Use a buffer of bytes. This works only if your endian-ness is correct.
  uint8_t *byteBuffer = (uint8_t *) buffer;
  bufLength = bufLength*2;


  IoClear(LCD_SELECT_N);          // select the chip
  while (bufLength) {
    IoWriteBusByte(LCD_BUS, *byteBuffer);
    byteBuffer++;
    IoWriteBusByte(LCD_BUS, *byteBuffer);
    byteBuffer++;
    bufLength--;
  }
```

Lookup Tables

- Trade code space for speed
- Case 23: CRC calculation