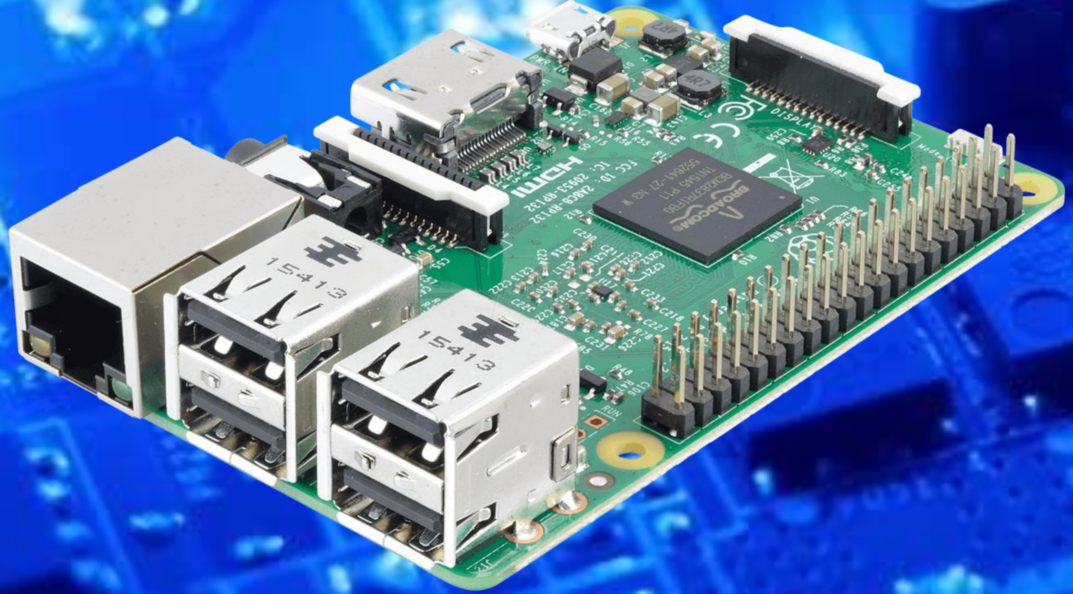
EEE3096S



Serious I/O Programming

Embedded Systems II

L17

Dr Simon Winberg

Electrical Engineering
University of Cape Town

Outline of Lecture

- Programming Memory Ports
- Memory Mapped & Port Mapped
- Use of the volatile keyword in C
- The Importance of Linkers
- Approaches to Memory Mapped I/O
- Class Activity:

  Assessing understanding of peripheral registers

Outline

1. Peripheral Access
2. Inline Assembly
3. Jump tables
4. Bitwise Operations

Part 1

This lecture

5. Shadow Registers (recap)
6. Speed and Code Density
7. Polling and Interrupts
8. Measuring execution time & Watchdog timers

Part 2

Embedded Systems Software Techniques

1. PERIPHERAL ACCESS & I/O

Embedded Systems II

Accessing Hardware in C

- Two approaches for external device IO
  - Port Mapped IO
  - Memory Mapped IO

Port Mapped IO

reminder

- A special IO address space is used to access devices
- Special CPU instructions are used to access this address space (e.g. **IN, OUT**)

Advantage:

- Peripherals can contain simpler memory address decoding logic.

Disadvantage:

- Special instructions and programming techniques required to access peripherals

Example mappings for a 16-bit processor with separate memory and port addresses

Memory Address Map

Start	End	Description
0x0000	0x000F	Exception Vector
0x1000	0x3FFF	Program memory
0x4000	0xFFFF	Data memory

Port Address Map

Start	End	Description
0x0000	0x0000	LEDs
0x0001	0x0001	Pushbuttons
0x0002	0x000F	Unused
0x0010	0x0011	UART control
0x0012	0xFFFF	Unused

Memory Mapped IO

reminder

Peripherals accessed through a dedicated area of memory address

Memory Mapped IO is the more common approach nowadays.

Advantages:

- No special instructions or techniques required
- Peripherals can be accessed with a standard C pointer

Disadvantages:

- Some address space used for peripherals
- Peripherals need complex address decoding hardware

Example mappings for a 16-bit processor with memory and I/O on same addresses bus

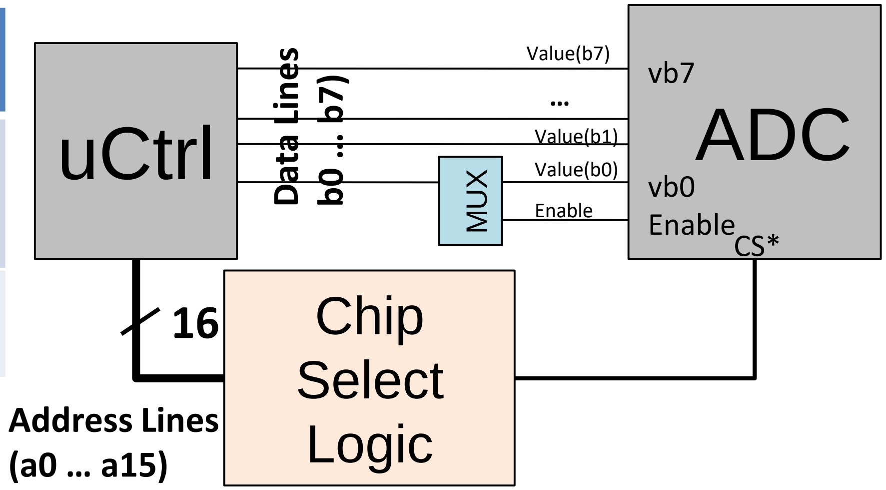
Address Map

Start	End	Description
0x0000	0x000F	Exception Vector
0x1000	0x3FFF	Program memory
0x4000	0xEFFF	Data memory
0xF000	0xFFFF	4Kb for peripherals

Memory Mapped I/O Examples

- Example assume you want to control a 8-bit ADC from a 16-bit microcontroller
- The ADC has following hardware peripheral mappings on the platform (see below)
- Multiple ways to implement this…

Address	Register Name	Description
0xFF00	Enable	Enable/disable register. Write 1 to enable device. Write 0 to disable device.
0xFF01	Value	Input value converted from the ADC



The volatile keyword:
essential ingredient for memory mapped I/O

- Generally, a compiler makes use of registers were possible instead of incurring the latency of accessing memory
- When reading or writing hardware registers, you want to force the compiler to generate code that explicitly accesses the memory locations concerned
- This can be done using the volatile keyword

E.g.: The first line below may optimize the generated machine code so that it writes once to address 0xFF00 and does not bother with the read. In the second line the volatile keyword in the declaration ensures 0xFF00 is written and then explicitly read in the printf.

```
1.  int *x = 0xFF00;  x[0] = 1; printf("%d",x[0]);
2.  volatile int *x = 0xFF00;  x[0] = 1; printf("%d",x[0]);
```

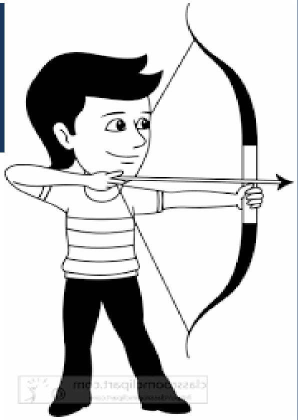
Various examples of the above code examples was given in Dr Gaffar lectures, but I think we didn't emphasise the relevance of volatile.

IMPORTANCE OF LINKERS

"The thingamajig that packages your program's stuff" (unofficial definition)

We'll see a more official definition soon…

Why should you know your linker?



Know your linker, it's like your marksman for your program resources

- Understanding how the linker works is important for embedded systems developers
- Reasons for this
  - Know where compiler tools are placing your program's instructions and data
    - Your architecture may need instructions and code placed in specific areas to run, this may be particularly important if running stand-alone (without an OS)
  - Know what footprint your program will take
    - This can be an important step in planning the deployment of applications and to guide optimization, e.g. if you have a system with slow and fast memory you may want certain code (e.g. ISRs) in fast memory
  - Improve the safety/reliability of the program
    - Depending on your architecture you may to put sensitive operations or data at specific locations
    - Ensure RO memory going into physical RO memory locations
    - May have size-limited encrypted/locked memory sections that you want sensitive code placed so that this cannot be reverse engineered

The GCC linker is quite a sophisticated tool, but we will only briefly discuss it focusing on the essential aspects that you may well encounter in ES product development.

Linkers – what they are

- From CS lectures you presumably are aware of what a linker is…
- <u>Defn. Linker:</u>

  In computing, a linker or link editor is a computer utility program that takes one or more object files generated by a compiler and combines them into a single executable file, library file, or 'object' file.

  (source: https://en.wikipedia.org/wiki/Linker_(computing))

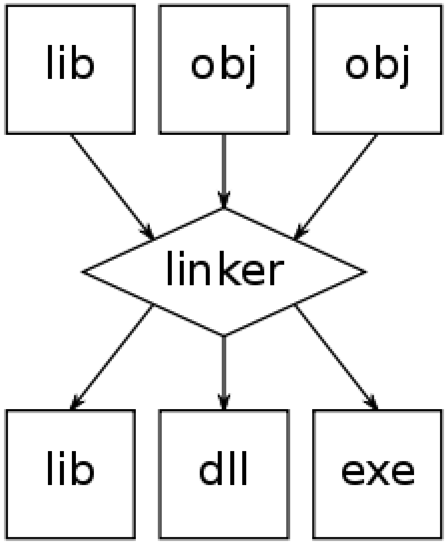


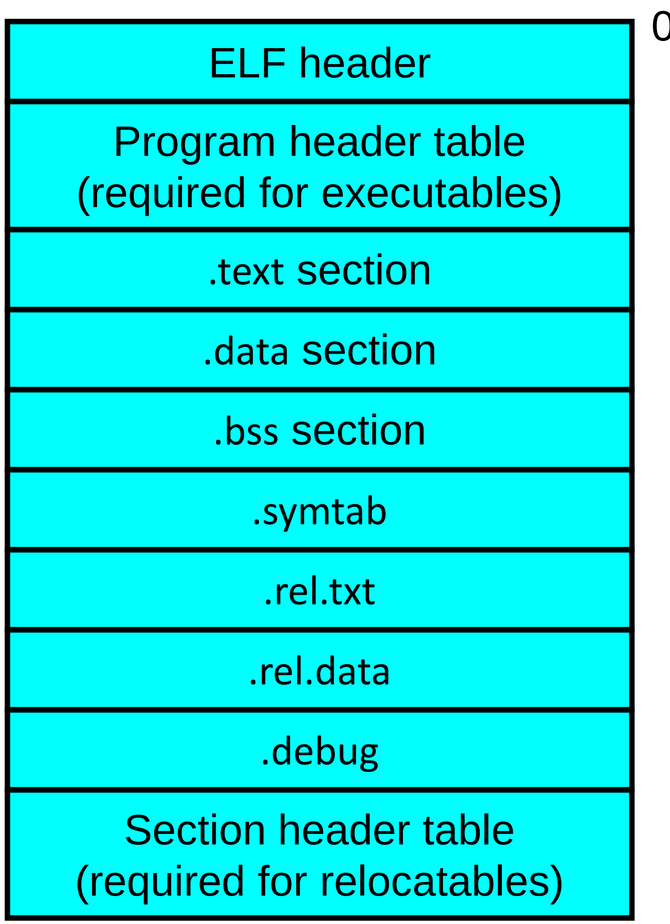
Illustration of the linking process. Object files and static libraries are assembled into a new library or executable

What Does a Linker Do?

- Merges object files
  - Merges multiple relocatable (.o) object files into a single executable object file that can loaded and executed by the loader.
- Resolves external references
  - As part of the merging process, resolves external references.
    - External reference = reference to a symbol defined in another object file
- Relocates symbols
  - Relocates symbols from their relative locations in the .o files to new absolute positions in the executable.
  - Updates all references to these symbols to reflect their new positions

ELF Object File Format

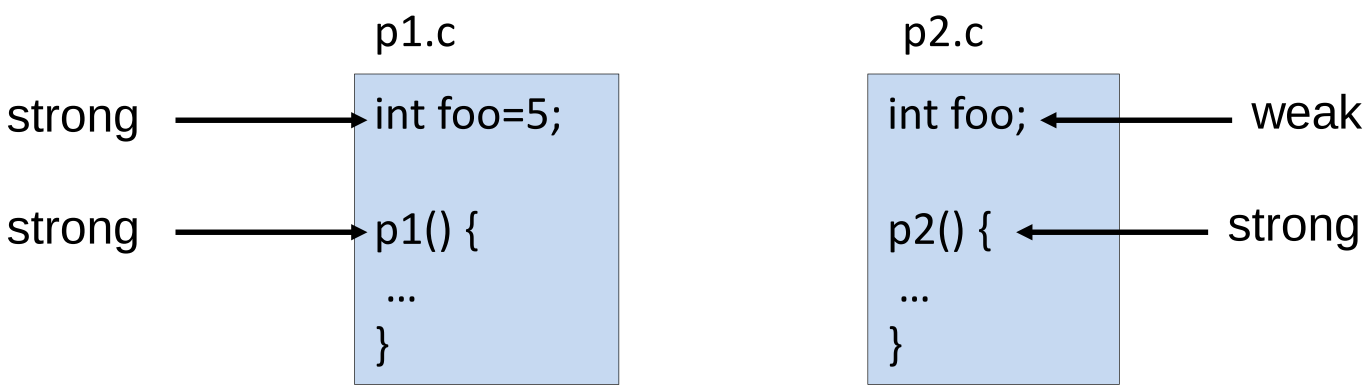
- ELF =
  Executable and Linkable Format
  (formerly “Extensible Linking Format”)
- Is a common standard file format for executable files, object code, shared libraries used by Linux
- This is where your object files and program resources eventually end up.



ELF file structure

Strong and Weak Symbols

- Program symbols are either **strong** or **weak**
  - *Strong* = procedures and initialized globals
  - *weak* = uninitialized globals

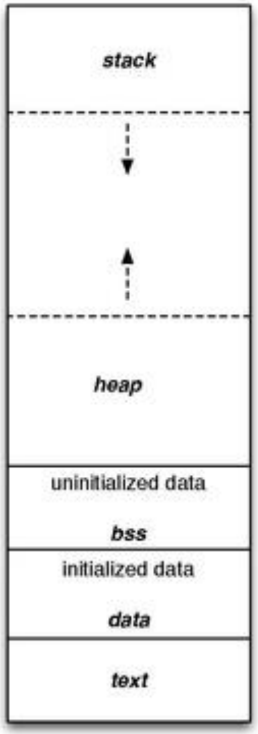


Strong symbols are more certain to be allocate addresses and spaces in the linker file and given explicit initial values, whereas weak symbols may not be optimized out

Example adapted from slides developed by Randal E. Bryant and David R. O'Hallaron for CS 15-213

Linker Sections

- .text section
  - Executable instructions and read-only fixed size constants. (i.e. not just instructions)
    - e.g. void dostuff () { printf(“hello\n”); }
      would include printf and the constant string in the .text section.
- .data section
  - Initialized global or static variables and memory blocks / arrays
    - e.g. char string[] = "Hello World";
      This 12 character string and its initial characters would be slotted into the .data section
- .bss section
  - Uninitialized data section, usually adjacent to data segment. Contains all global variables and static variables that do not have explicit initialization.
    - e.g. static int i; would go into the BSS segment.



Memory model for ELF-based systems

Memory Check

1. If you initialize a global array
   e.g. int x[100] = { 0, 1, 2, 3, 4 }; …
   which segment will this go into?

   Answer: .data section

2. If you declare a constant variable
   e.g. const float pi = 3.141592;
   which segment will this go into?

   Answer: .text section

3. If you declare an uninitialized counter value
   e.g. int count;
   which segment will this go into?

   Answer: .bss section

Memory Mapped I/O Approach 1

- Example to:
  - Creating array in C (force as strong symbol using extern)
  - Configuring the linker to place the array at a specific address

File: adc.c

```
unsigned char adc[2];
```

File: main.c

```
extern unsigned char adc[];
void main () {
  // byte to store input value
  unsigned char x;
  // enable the ADC
  adc[0] = 1;
  // read a value
  x = adc[1];
}
```

Mod. linker file to something like:

```
…
SECTIONS {
   …
# Add command lines:
ioout 0xFF00: {
   adc.o } # add all of adc.o
}
```

Memory Mapped I/O Approach 2

- Example data used in assembly that is accessible in C
  - Creating an array in assembler
  - Use extern in C to link to it

File: main.c

```
extern unsigned char adc[];
void main () {
  // byte to store input value
  unsigned char x;
  // enable the ADC
  adc[0] = 1;
  // read a value
  x = adc[1];
}
```

Assembly module:

```
…
.org 0xFF00
.global adc
adc:  # use a label
.byte adc_enable
.byte adc_val
```

Don’t worry about this too much now, we will revisit assembly next term

Memory Mapped I/O Approach 3

- Using a pointer in C
- Example:

```
File: main.c
unsigned char adc* = 0xFF00;
void main () {
  // byte to store input value
  unsigned char x;
  // enable the ADC
  adc[0] = 1;
  // read a value
  x = adc[1];
}
```

Clearly this is the approach is more likely to be used (if there are few ports)… but if you aren't using C or have a lot of assembly code that needs to access port registers then it may well be necessary to know how the other approaches work.

Memory Mapped I/O Approach 4 (ultimately my preferred approach!)

- Using a struct in C – keep it neatly together
- Example:

This is technically called a peripheral register structure

File: main.c

```
typedef struct {
   unsigned char enable;// first register at  0xFF00
   unsigned char value; // second register at 0xFF01
   } ADC;
volatile ADC* adc = (ADC*)0xFF00;
void main () {
   // byte to store input value
   unsigned char x;
   // enable the ADC
   adc->enable = 1;
   // read a value
   x = adc->value;
}
```

That's stuff is in reality mucho importente!

Simon's fond reminiscence of peripheral register structures and other joys of embedded programming…

And voila EEE3096S students are happy experts in peripheral registers! Yaay!

But let's first confirm the hypothesis…

Class Activity & take-home [group] exercise

Consider you are developing a simple digital recording device based on a 32-bit microcontroller. The peripherals and memory used by system are…

<u>Peripherals:</u>

- 10-bit ADC (for recording voice)
  - Has a 1 bit Data_Request line and a 10 bit Data_Out line
- USB (for downloading the recorded data)
  - Send_Data input (when set sends Data In out the USB port)
  - Receive_Data input (when set sets Data Out to last received 8bit sequence)
  - 8-bit DataIn line and 8-bit DataOut line
- 4 x LEDs (“power”, “record”, “full”, “comms”)
- 2 pushbuttons (“record/stop”, “pause/continue”)

<u>Memory:</u>

- 32Kb internal program flash fixed at address 0x0 – 0x7FFF
- 32Kb internal SRAM for data memory fixed at address 0x8000 – 0xFFFF
- 2 Megabytes external RAM on memory bus

TODO: Assume you want to use memory mapped port access. Develop a memory map, showing how the memory devices will be mapped and showing an appropriate mapping of where the above peripherals could be situated in the map. Note that you need to think about the chip select that would be needed so that when you read/write a particular address the correct device will be activated.

The Next Episode…

Lecture P18

Serious I/O Programming (II)

And onwards to ADCs and Interrupts!