Writing Portable Device Drivers

INSTRUCTOR: AARON BARANOFF

This is Strictly a Practical Class

Give a man a fish and you feed him for a day. Teach him how to fish and you feed him for a lifetime. Lao Tzu

Give a man a fish and you feed him for a day. Teach him how to fish and you feed him for a lifetime. If teach about the science or theory of fishing prior to learning to fish he will starve before he will learn.

Aaron Baranoff's addition to Lao Tzu's Quote

How to contact me…

E-mail: aaron_baranoff@hotmail.com

I do have another email addresses aebaranoff@gmail.com and aaron@mataitech.com as well as other work ones the hotmail is the one I have been using for teaching and gets forwarded to my cell phone.

I generally answer emails within 24 hours but on a rare occasion I too get swamped and it could take up to 72 hours. If I don't respond to your email within 48 hours please resend.

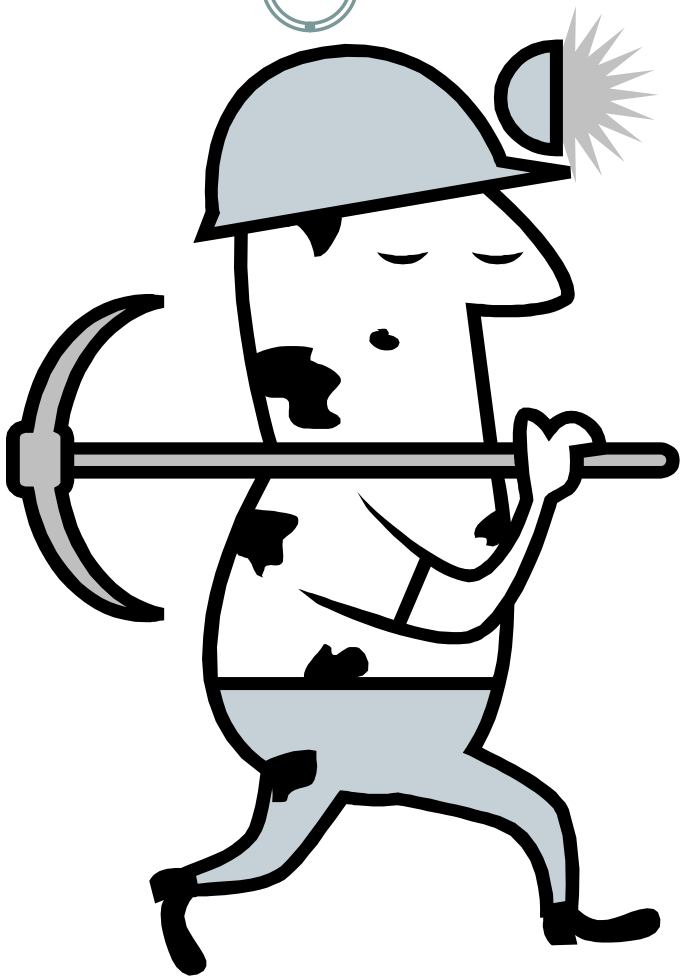
Sent me an e-mail so I get each of your e-mails also so you get each others emails.

E-mail and talk to each other you and the people you meet in this business are each other best source of knowledge and networking.

Week 2

- Introduction
- What is Software?
- What is a device driver
- What is an embedded system
- Discussion - Embedded Systems All Around Us

Welcome to the "Bit Mines”



“bit mines” was coined by an unknown engineers

Limited Number of Weeks not Limited access

- Whether this class is in progress or the class has ended I will always be your instructor.
- Feel free to keep in touch.
- I enjoy hearing and helping my students from past and present.

3 Credits does not mean 30 hrs of me talking and you watching slides.

- I will talk through the slides (the slides themselves contain part but, not all of the information)
- I will post more info, supplemental material and examples during the course of the class.
- In addition I post hints and tips on the labs as we reach that point.
- I will also of course give individual as well as group feedback on the labs.
- Your mission… Learn, do the labs and assignments and ask questions. This will take more then 30 hours of your time. But, has been designed to be doable for a person working fulltime with a family and taking this class.

Introduction (Assignment 1)

- Let's introduce each other. Remember that it is a small world. You may work with these people again and they may help you find opportunities in the future.
- Please include your name, what you do for a living, why you are taking the class and what you want out of the class, and some info about your software experience.
- Lastly when you get a chance send me an email so I can email you with relevant materials or course updates as needed. (also remove your from my spam filter)

(post it to the web – pictures are appreciated but, not required)

Prerequisites

- Comfortable C programmer
- An assembly language is a plus but not a requirement.
- Willingness to learn
- Willingness to ask questions. Don't be shy.

Class Goals

- Learn what is a device driver
- Learn what is portable code
- Learn how to read a data sheet
- Learn how to write a portable device driver
- Learn how to avoid common pitfalls

Be able to apply what was learned and network with the rest of the class.

The Boards and Tools

- AVR STK600 (ATMega2560)
- AVR Studio 4
- GCC for AVR

Course Outline

- Introduction
- What is a device driver
- Interfaces of a device driver
- What is the driver writer responsibility
- What does portability mean
- Portability (Integers, Comments, Floating Point, Pitfalls, Endian, structures...)

- Labs – Review above (plus addition C refresher)
- DMA
- Sequence
- Timing
- Critical Regions
- Verification
- Learning to read data sheets
- More eyes the merrier
- Write you code as if…

Course Outline (part 2)

- Counters
- Registers vs.... the Register keyword
- The Volatile Keyword
- Statics
- Labs – Review above
- Caches
- Optimizers
- Interrupts / Interrupt Service Routines
- Deferred Processing
- Tasks
- Wrapper functions
- Keeping up with hardware
- C++ and drivers
- Labs – Review above
- Wrap-up
- Common Sense Prevails

Detours are Good

- Post questions (lots of questions)
- Anything goes loosely related to embedded systems. As long as everyone respects each other.
- If it at all relates to the class I will answer the questions.
- Feel free to answer fellow students questions (I will make sure we stay on course)

(Have fun and enjoy)

Grading

- Participation (active)
- Labs (Equally weighted)
- Feedback (questions and answers)

What is software?

- Software is simply the sequence of code required to get the proper bits in and out of the proper bytes in the right sequence with the correct timing and present them to the user of the code in a way they can manipulate the process without violating the hardware rules and without violating common sense.
- The user of the software can either be another piece of software or user buy way of some hardware.
- This all needs to be done in a way that is both portable, readable and maintainable.

What is an Embedded System

Here is the dictionary definition...

embedded system *computer*

Hardware and software which forms a component of some larger system and which is expected to function without human intervention.
A typical embedded system consists of a single-board **microcomputer** with software in **ROM**, which starts running some special purpose **application program** as soon as it is turned on and will not stop until it is turned off (if ever).
An embedded system may include some kind of **operating system** but often it will be simple enough to be written as a single program. It will not usually have any of the normal **peripherals** such as a keyboard, monitor, serial connections, mass storage, etc. or any kind of user interface software unless these are required by the overall system of which it is a part. Often it must provide **real-time** response.
Usenet newsgroup: **comp.arch.embedded**.
(1995-04-12)

The Free On-line Dictionary of Computing, © 1993-2006 Denis Howe

An Embedded System (part 2)

Here is my definition…

embedded system

- Hardware and software which is largely stand alone in operation and performs much of its functionality without human interaction. This does not mean that humans don't interact with these systems just that the systems don't require human interaction for much of their functionality. For example a phone can be an embedded system even though a human dials and answers it since it could function via auto answer or by acting as a modem.
- One of the unique destinations is that such systems software is not usually developed on the same system. The code is cross-compiled from another machine and then loaded on that machine/system.
- Since human interaction may not be required many of these systems have no user interface and many only are expected to communicate with other systems.
- Another common item in embedded systems is a real-time aspect and the closer coupling between hardware and software.
- Note: Much of a PC is a collection of embedded systems from sound cards, to video cards, keyboards, monitors, hard disks and more.

Comments on the dictionary definition…

- 1) Single CPU – not a requirement
- 2) Application and OS in many embedded systems are intertwined and custom. Most have no OS.

Things over time become embedded (part 1)

In September 1956, IBM launched the 305 RAMAC, the first computer with a hard disc drive. The HDD alone (pictured being loaded onto a commercial airline) weighed over a ton and stored only 5MB of data.
For a perspective your cell phone has more storage and more reliable storage then that. Think about those USB sticks and your digital camera.
By standards back then a home PC would have been considered portable. (source friend)

Things over time become embedded (part 2)

Thanks to Computerworld for digging up this from the late 1970s. By comparison my 1st HD was 5 megs and was about the size of a standard desktop PC and cost $1500 in 1984 (external of course).

Today (2007) 750Gigabytes external hard drive is about $200.

I spend about $40 on a 2 gig SD card.

Things over time become embedded (part 3)

Now your cell phone has more computing power and more storage then my first hard disks combined and is far more reliable then my first hard drives and yes it uses less power.

This same flash technology is used in routers, DSL/Cable modems, thermostats home appliances, in implanted medical equipment and in your car…

Assignment 2

- Tell me and the class about embedded systems in your life and work.

- Post the information to the web. We will then talk about whether it is an embedded system or not.

(there is no wrong answers on this one)

Some Embedded Systems

What is a Device Driver?

- It is the software glue between the device (hardware) and all other software (application or operating system).

What is the difference between a Device Drivers and application code?

- While remembering software is setting and getting bits in the right sequence and timing. Not all software needs to worry about all these things.
- Given a set of inputs most applications don't care much about sequence or timing. An many applications don't care about specific addresses.
- Device drivers must care since the hardware cares...

The Top-Down Myth

- Design can be either top down or bottom up. Implementation does not necessarily work this way.
- Driver implementation is from the ‘knowns’ to lesser ‘knowns’. Logically so should the design
- Implementation (coding) of a driver should follow common sense.
- Would you build a building from the roof down to the foundation or the foundation up to the roof. An architect can design either way but a building is build from the bottom up all the time.

From the ‘Knowns’ to the Lesser ‘Knowns’

- Knowns
  - The interface to the hardware.
  - The applications programmers interface and external structures.
  - The programming language.
- Lesser Knowns
  - Sequence.
  - Internal structures.
  - Implementation details.

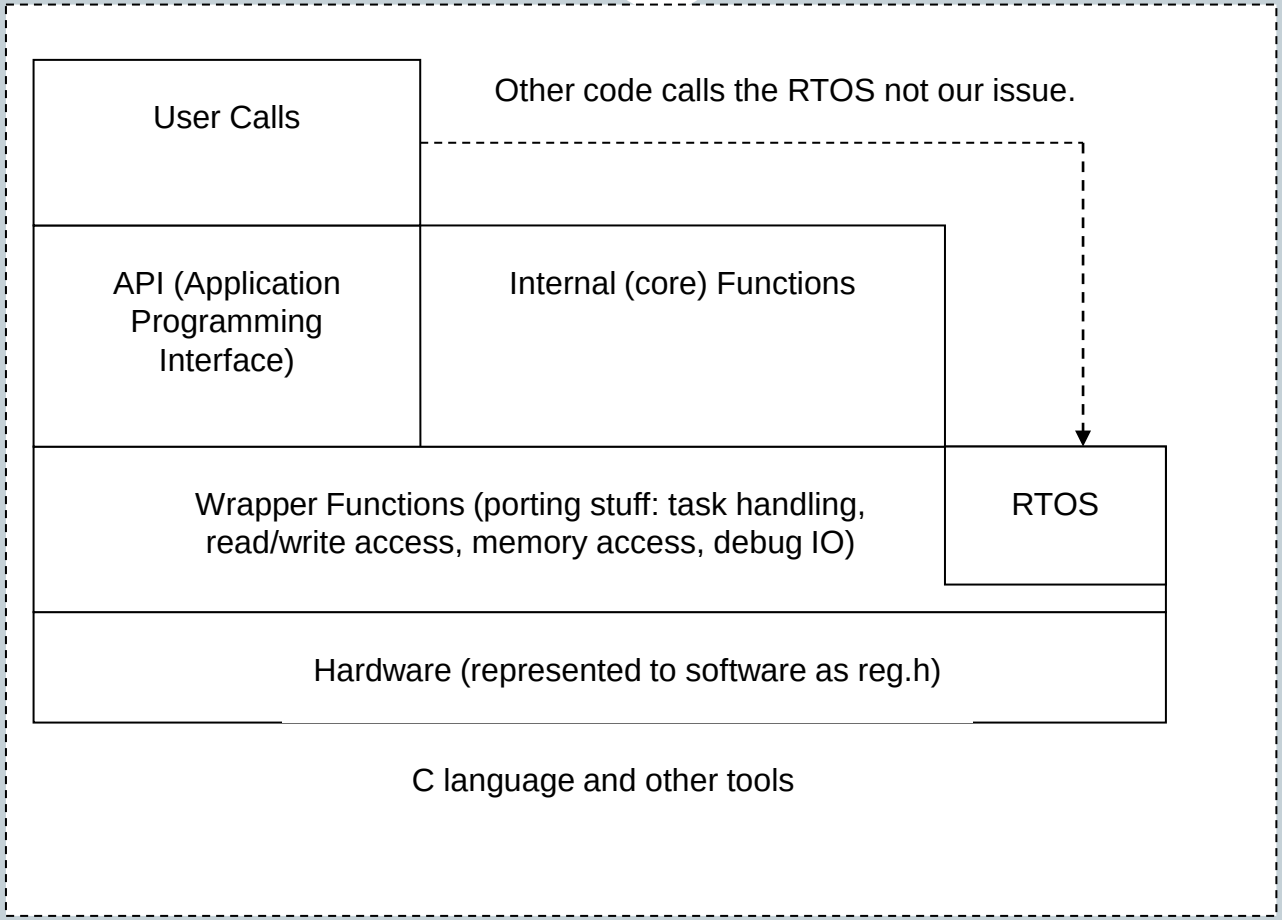
From the ‘Knowns’ to the Lesser ‘Knowns’ (con’t)

- Implementation is definitely a case of designing from the ‘knowns’ and meeting in the middle.
- Design is largely the same process as in most drivers implementation the only difference is instead of generating code the goal is to fill in the unknowns.

Interfaces of a device driver

- API - Application Programmer Interface
- Hardware interface
- Operating System (OS) Interface

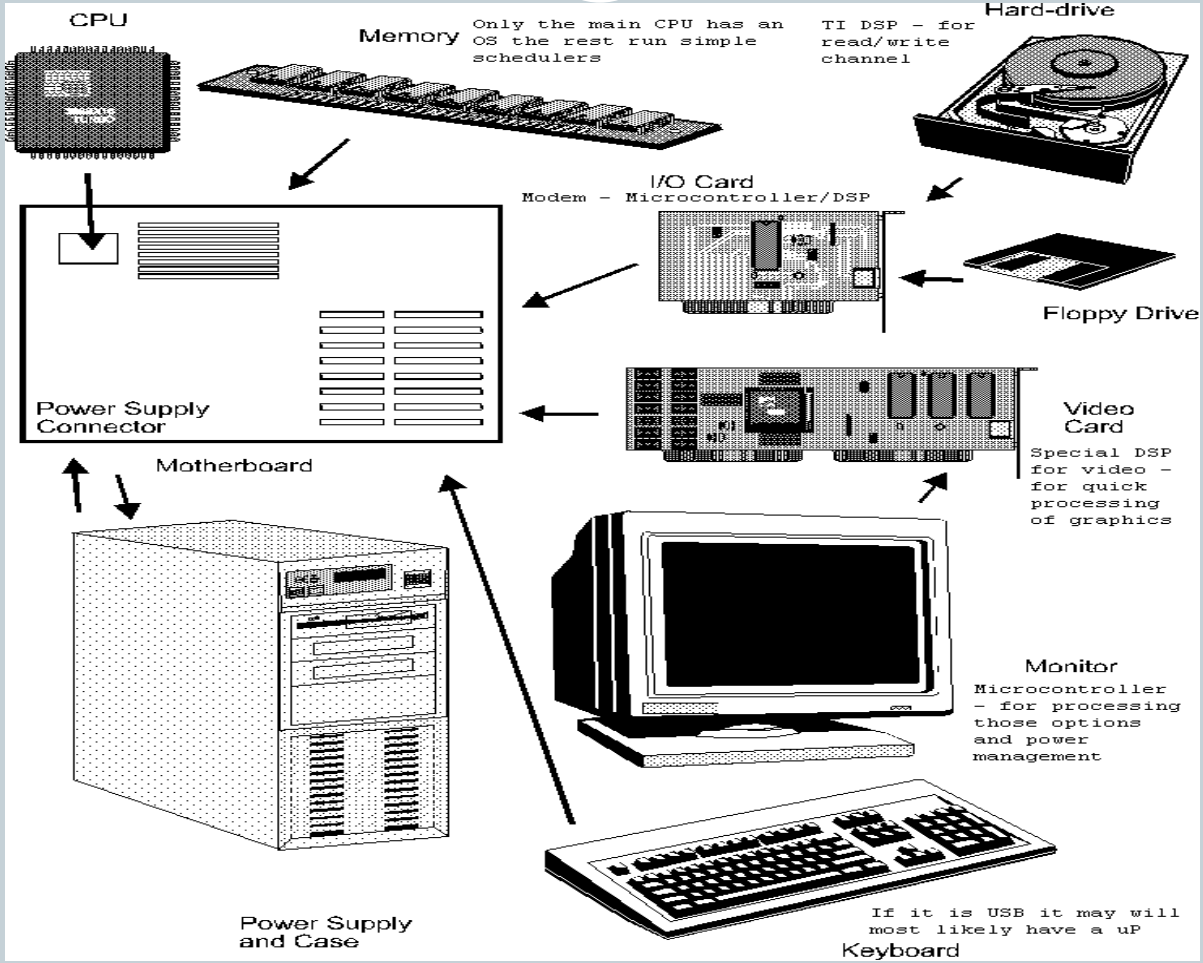
Interfaces of a device driver (con't)



RTOSes are the least important thing to a well written driver

- Remember 70 to 90% of all embedded systems do not have an OS.
- Most that do have an OS have a proprietary or non-standard one.
- Only expect resources you truly need.

Is a PC an Embedded Platform



What are the responsibilities of the driver writer

- Writer of the driver
- Guinea pig for new hardware
- Tester of your software
- Tester of the hardware
- First customer
- Tester of application which calls your driver.
- Fixer of all non-easily fixable hardware problems.
- Plus much more...

Is a writing a device driver an Art or Science

- **art** *n.*
- Human effort to imitate, supplement, alter, or counteract the work of nature.
  - The conscious production or arrangement of sounds, colors, forms, movements, or other elements in a manner that affects the sense of beauty, specifically the production of the beautiful in a graphic or plastic medium.
  - The study of these activities.
  - The product of these activities; human works of beauty considered as a group.
- High quality of conception or execution, as found in works of beauty; aesthetic value.
- A field or category of art, such as music, ballet, or literature.
- A nonscientific branch of learning; one of the liberal arts.
  - A system of principles and methods employed in the performance of a set of activities: *the art of building.*
  - A trade or craft that applies such a system of principles and methods: *the art of the lexicographer.*
  - Skill that is attained by study, practice, or observation: *the art of the baker; the blacksmith's art.*
  - Skill arising from the exercise of intuitive faculties: “Self-criticism is an art not many are qualified to practice” (Joyce Carol Oates).
  - **arts** Artful devices, stratagems, and tricks.
  - Artful contrivance; cunning.

Is a writing a device driver an Art or Science (cont.)

- **sci·ence** *n.*
  - The observation, identification, description, experimental investigation, and theoretical explanation of phenomena.
  - Such activities restricted to a class of natural phenomena.
  - Such activities applied to an object of inquiry or study.
- Methodological activity, discipline, or study: *I've got packing a suitcase down to a science.*
- An activity that appears to require study and method: *the science of purchasing.*
- Knowledge, especially that gained through experience.

Source: *The American Heritage® Dictionary of the English Language, Fourth Edition*

Portable Device Driver is the Foundation of Your Customer's System

- The more complex the driver the harder it is to make it portable.
- The more complex the driver the more valuable it is to make it portable.
- The more portable it is the more systems will be able to make use of the device and the driver.

Definition of Portability for a Device Driver

- Writing a driver such that the majority of the code has no dependence on the OS, CPU and compiler, etc. The code must be segmented in such a way that the sections that are portable are clearly documented and separate from those that me need to get modified for a given OS, CPU and compiler, etc.
- This does not guarantee 100% compatibility across all systems this minimizes the effort required to change this driver.
- This also does not necessarily impact performance but may depending on how you doing your porting. (i.e. Performance vs. Portability) is not mutually exclusive).

What does Portability mean?

- Portability across CPUs
- Portability across OSes
- Portably across Compilers
- Portability across the unknown

Portability across CPUs

- Big vs... Little Endian
- Bus Width
- Performance

Portability across OSes

- PC Operating Systems
- Real-Time Operating Systems
- Micro Kernels
- Timer Based Schedulers
- Interrupt Driven Systems
- Systems with only a main program flow

Portably across Compilers

- ANSI C does not guarantee compatibility.
- What are all those types?
- What can I rely on?
- Is RAM initialized and to what and how?

Portability across the unknown

- For convenience the hardware folks decide the lower bits of address are tied low and the least significant bit of your address is not 0 but something else.

Simple (maybe not so simple) example of what can go wrong


```
int i = 0;
int *address;

// read an address
int readAddr(int *addr)
{
        return *addr;
}

// dump first 10 registers of the device
example()
{
        address = 0; // assuming the base address is 0
        while (i < 10)
        {
                printf("%d = readAddr(0x%x)\n", readAddr(address[i]), i);
                i++;
        }
}
```

Simple (maybe not so simple) example of what can go wrong (con't)


```
int i = 0;
int *address;

// read an address
int readAddr(int *addr)
{
        return *addr;
}

// dump first 10 registers of the device
example()
{
        address = 0; // assuming the base address is 0
        while (i < 10)
        {
                printf("%d = readAddr(0x%x)\n", readAddr(address[i]), i);
                i++;
        }
}
```

1. Will I even be initialized? Not in some embedded environments without external help.
2. What address will this print? What if this is on a 16 bit system, how about 32 or 64 bit system.
3. printf does not always exist on some embedded environments and it is also a stack hog.
4. // can't be used on a C compiler.

Outline of a Portable Driver

- xxxport.c – porting functions
  1. read/write functions
  2. OS call wrappers
  3. NO globals initialized outside a function
- xxxport.h – porting macros
  1. typedefs
  2. additional OS call macros
  3. declarations for functions defined in the OS
  4. OS required headers.
  5. NO globals or statics initialized outside a function
- xxxapi.c – your API

  NO OS or CPU specific stuff. NO globals initialized outside a function
- xxxapi.h – your API declarations (possible register defines for your device)

  NO OS or CPU specific stuff. NO globals initialized outside a function
- xxxother.h – your other structures

  NO OS or CPU specific stuff. NO globals initialized outside a function
- xxxint.c – your internal functions

  NO OS or CPU specific stuff. NO globals initialized outside a function
- xxxint.h – your internal declarations

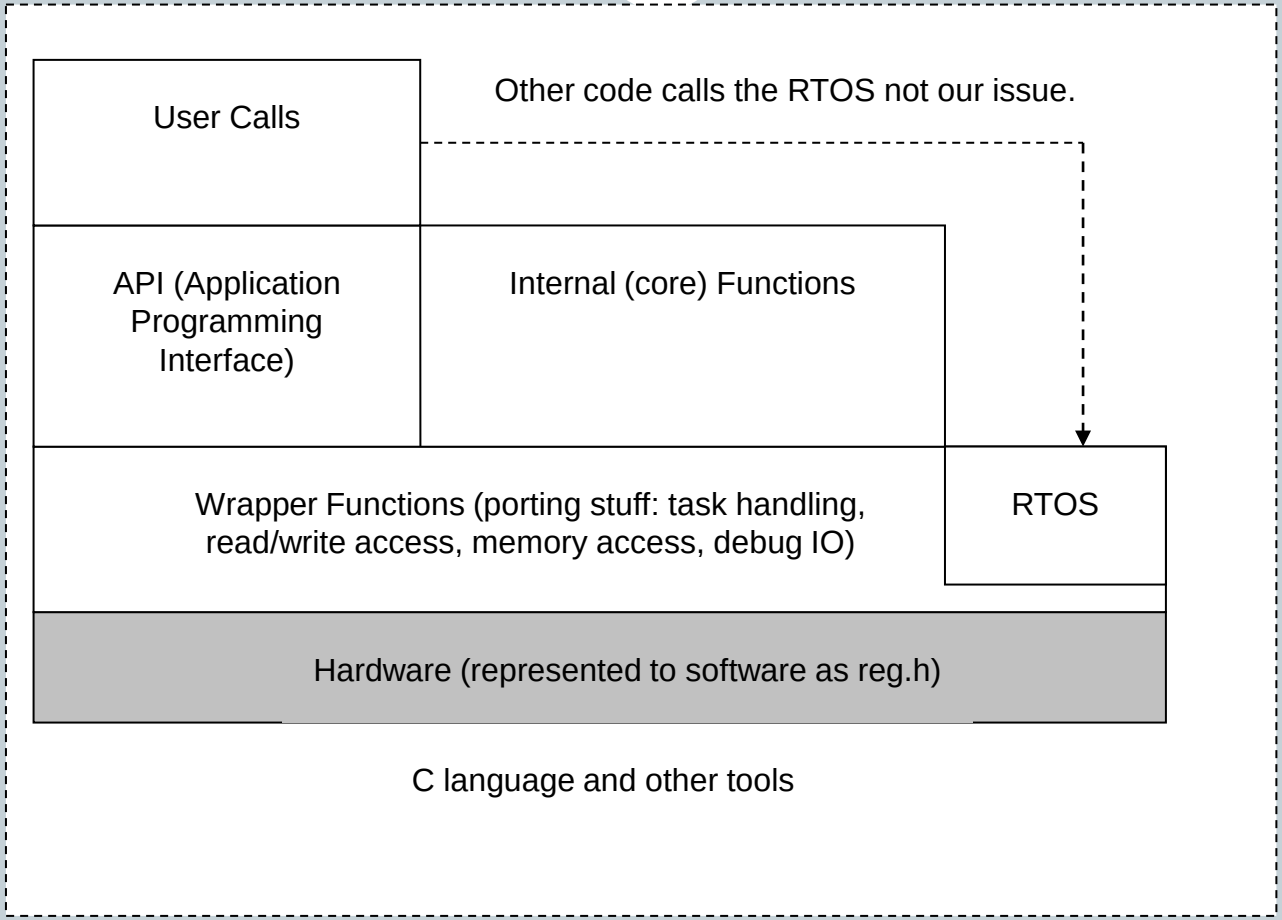
  NO OS or CPU specific stuff. NO globals initialized outside a function

Steps in Writing a Portable Device Driver

1. Review the Data Sheet and/or other documentation. Try to understand basically what this device is supposed to do.
2. Create you device specific header file (#define(s) for address offsets and bits and bit masks. Be lazy if possible cut and paste them from the hardware documentation. While you are cutting and pasting and reformatting, study the registers and learn their meaning.
3. Based upon the review of the registers, reread the datasheet to get a better understanding of the part now. Try to understand how what bits gets set or cleared in the registers impacts the device. Is this device part of the data path or is it initialized and forgotten about. When will I need to talk to it. Learn the basic sequence requirements
4. Based upon what this device is consider what kind of APIs will it need. Are there any standards (or spec) based APIs that you must support. Outline them or even code empty functions ie.

   ```
   initialize_device()
   {
   }
   ```
5. As you are thinking about how the APIs are going to interface with the registers. Start coding your porting functions and outlining your high-level data structures and tasks etc.

The First and Most Important Step Create a Good Register Header File



Read the Register Info and Convert it to a Header File

12.4.5 PORTB – Port B Data Register

Bit	7	6	5	4	3	2	1	0	
0x05 (0x25)	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

12.4.6 DDRB – Port B Data Direction Register

Bit	7	6	5	4	3	2	1	0	
0x04 (0x24)	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	DDRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

12.4.7 PINB – Port B Input Pins Address

Bit	7	6	5	4	3	2	1	0	
0x03 (0x23)	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	PINB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

Read the Register Info and Convert it to a Header File (con't)

EEARH and EEARL – The EEPROM Address Register

Bit	15	14	13	12	11	10	9	8	
0x22 (0x42)	–	–	–	–	EEAR11	EEAR10	EEAR9	EEAR8	EEARH
0x21 (0x41)	EEAR7	EEAR6	EEAR5	EEAR4	EEAR3	EEAR2	EEAR1	EEAR0	EEARL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	X	X	X	X	
	X	X	X	X	X	X	X	X	

- **Bits 15:12 – Res: Reserved Bits**

These bits are reserved bits and will always read as zero.

- **Bits 11:0 – EEAR8:0: EEPROM Address**

The EEPROM Address Registers – EEARH and EEARL specify the EEPROM address in the 4 Kbytes EEPROM space. The EEPROM data bytes are addressed linearly between 0 and 4096. The initial value of EEAR is undefined. A proper value must be written before the EEPROM may be accessed.

EEDR – The EEPROM Data Register

Bit	7	6	5	4	3	2	1	0	
0x20 (0x40)	MSB							LSB	EEDR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bits 7:0 – EEDR7:0: EEPROM Data**

For the EEPROM write operation, the EEDR Register contains the data to be written to the EEPROM in the address given by the EEAR Register. For the EEPROM read operation, the EEDR contains the data read out from the EEPROM at the address given by EEAR.

Read the Register Info and Convert it to a Header File (con't)

EECR – The EEPROM Control Register

Bit	7	6	5	4	3	2	1	0	
0x1F (0x3F)	–	–	EEPM1	EEPM0	EERIE	EEMPE	EEPE	EERE	EECR
Read/Write	R	R	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	X	X	0	0	X	0	

- **Bits 7:6 – Res: Reserved Bits**

These bits are reserved bits and will always read as zero.

- **Bits 5, 4 – EEPM1 and EEPM0: EEPROM Programming Mode Bits**

The EEPROM Programming mode bit setting defines which programming action that will be triggered when writing EEPE. It is possible to program data in one atomic operation (erase the old value and program the new value) or to split the Erase and Write operations in two different operations. The Programming times for the different modes are shown in Table 8-1 on page 36. While EEPE is set, any write to EEPMn will be ignored. During reset, the EEPMn bits will be reset to 0b00 unless the EEPROM is busy programming.

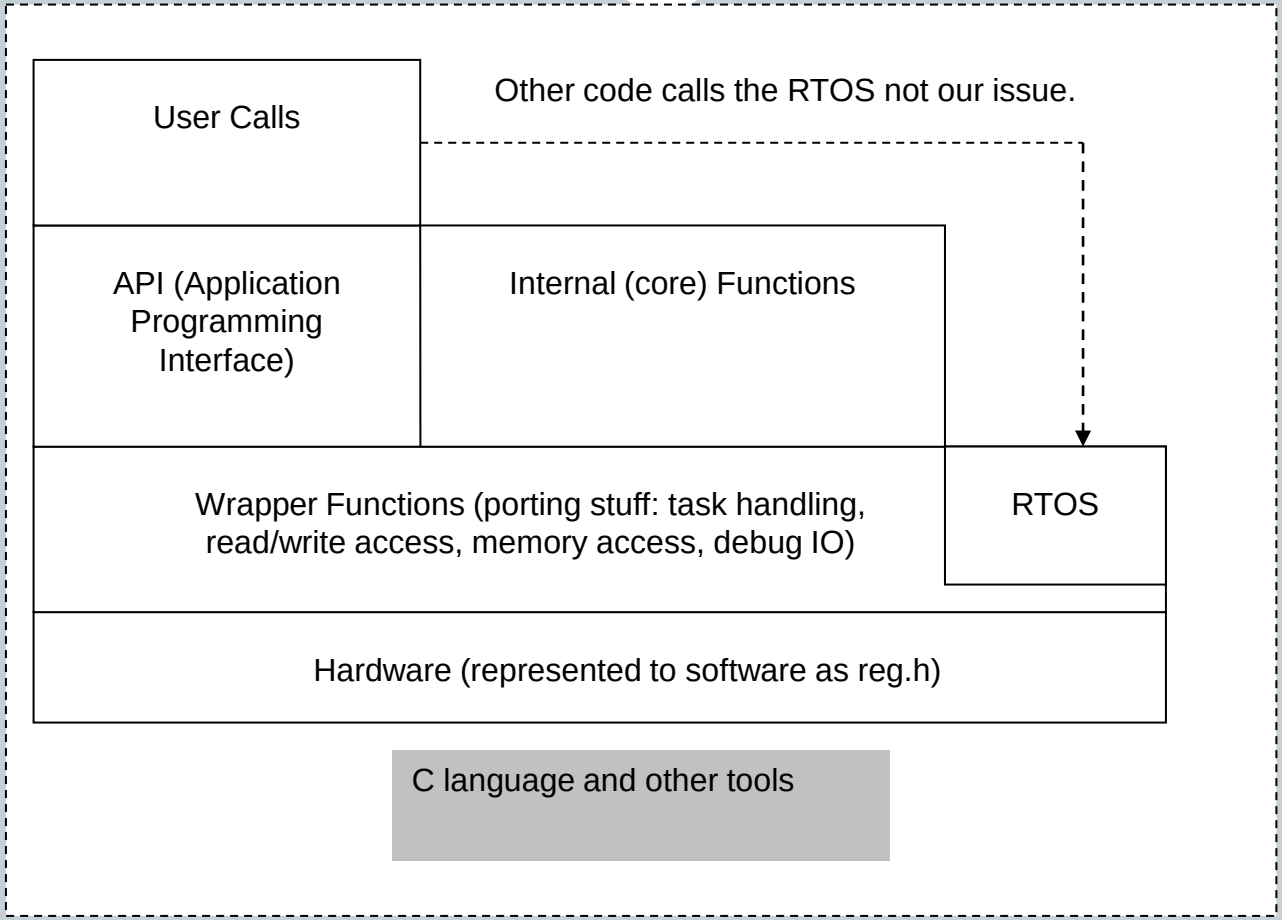
Read the Register Info and Convert it to a Header File (con't)


```
#define PORTB        0x25 /* Port B Data Register */
#define PORTB7    (1<<7)    /* R/W */
#define PORTB6    (1<<6)    /* R/W */
#define PORTB5    (1<<5)    /* R/W */
#define PORTB3    (1<<3)    /* R/W */
#define PORTB4    (1<<4)    /* R/W */
#define PORTB2    (1<<2)    /* R/W */
#define PORTB1    (1<<1)    /* R/W */
#define PORTB0    (1<<0)    /* R/W */

#define PINB        0x24 /* Port B Input Pins Address */
#define PINB7    (1<<7)    /* R/W */
#define PINB6    (1<<6)    /* R/W */
#define PINB5    (1<<5)    /* R/W */
#define PINB3    (1<<3)    /* R/W */
#define PINB4    (1<<4)    /* R/W */
#define PINB2    (1<<2)    /* R/W */
#define PINB1    (1<<1)    /* R/W */
#define PINB0    (1<<0)    /* R/W */

#define DDRB        0x24 /* Port B Data Direction Register */
#define DDRB7    (1<<7)    /* R/W */
#define DDRB6    (1<<6)    /* R/W */
#define DDRB5    (1<<5)    /* R/W */
#define DDRB3    (1<<3)    /* R/W */
#define DDRB4    (1<<4)    /* R/W */
#define DDRB2    (1<<2)    /* R/W */
#define DDRB1    (1<<1)    /* R/W */
#define DDRB0    (1<<0)    /* R/W */
```

Learning to read data sheets

- A TDK Ethernet PHY
- Our Atmel board and the ATMega2560 chip

Another Part of the interface is to the C language and the C compiler



Lab 1

- To familiarize yourself with the ATMEL board and tools compile, build and run the test program I will supply to you and show me it compiling and running.

```
/* LAB 1 - Writing Portable Device Drivers */
/* poor example of portability              */
/* fine for tool demonstration              */
/* Aaron Baranoff                          */


void delay(unsigned int val)
{
    unsigned int temp;
    unsigned int temp2;

    for(temp = 0; temp < val; temp++)
            for (temp2 = 0; temp2 < val; temp2 ++) ;
}

int main(void)
{
    *((volatile unsigned char *)0x24) = 0xff;
    while(1) {
            *((volatile unsigned char *)0x25) = 0;
            delay(500);
            *((volatile unsigned char *)0x25) = 0xff;
            delay(500);
    }

    return(0);
}
```

Sizeof is Your Friend

- sizeof() – lets you at compile time determine the size of and structure or variable that is passed to it in bytes. You can use sizeof() to verify porting correctly. You need to use it when allocating memory.

```
void sizeofdemo(void)
{
   struct
   {
         unsigned char a;
         unsigned long int b;
   } mystuff;
   /* prints 1 */
   printf("Size of a char %d\n", sizeof(char));
   /* prints 2 */
   printf("Size of a short %d\n", sizeof(short));
   /* prints 4 (on system – but is system dependent) */
   printf("Size of a int %d\n", sizeof(int));
   /* prints 12 (on one system – but is system dependent) */
   printf("Size of a mystuff %d\n", sizeof(mystuff));
}
```

What is an integer?

- Don't Assume a Size
- It may be any size ANSI does not say for most types.
- Always define the types you what.

```
#define Uint32 unsigned int
#define Int32 signed int
typedef unsigned int Uint64
typedef signed int Int16
```

What is a char?

- This is probably safe to assume a size but just don't.
- Some compilers default this as signed others as unsigned
- So, always define the types you what.

```
#define Uint8 unsigned char
#define Int8 signed char
typedef Uint8 unsigned char
typedef Int8 signed char
```

Floating Point Numbers

- Floating is not available in all systems.
- When hardware does not support it is very CPU intensive.
- Use it if necessary but with care. Always warn in the comments and in any documentation that floating point is used.
- Sizes can differ.

Floating point and embedded systems

- Using true floating point only as necessary. It is very CPU intensive. On some systems that have no floating point assistance in hardware, it greatly increases your code size and may require additional work such as saving extra registers.
- Some floating point emulation is done in software when the hardware does not support it.
- When possible and with in reason, work around it.
- Used scaled integers when possible.

Floating point and embedded systems (cont)

- How do a round up without floating point

```
c = a / b; /* rounded down by default */
c = (a + (b-1)) / b; /* is a / b rounded up */
c = (a + ((b-1) / 2)) / b; /* round if greater
                                    than  half */
```

Order does matter for math

- Use parentheses to control order.
- There is a precedence order but, control what you want to happen.
- When you have space, multiply first then divide. This reduces rounding error.
- Be careful of rolling a signed value from positive to negative by adding a positive value.

Comments & Portability

- C++ style comments

```
// Comment
```

- C style comments

```
/* Comment */
```

Remember C comments can be used in C++ but, C++ can not be used in C. Therefore for portable sake use the C style comments.

Big Endian vs... Little Endian

- Byte order can be reverse between different processors.
- Little Endian is used on most Intel Platforms
- Big Endian is used on most non-intel platforms.
- What if you have a little endian device on a big endian system?

Not all pointers are the same

- How large is a pointer?

  On some systems it is 32 bits (4 bytes)

  On other systems it is 64 bits (8 bytes)

  And on some microcontrollers it is 16 bits (2 bytes or less)

- Are the pointers physical addresses or virtual address?

  Remember software applications can use virtual address but hardware want physical addresses. So conversion may be required.

- Remember the importance of sizeof()

Pointers

- &data – is the pointer (address of data)
- *pData – is the value that is pointed to by the address stored in pData;

```
void examplePointer(void)
{
   INT4 data=0;         /* data is a 4 byte integer */
   INT4 *pData;         /* pData is a pointer to 4 byte integer */

   pData = &data;      /* pData is now set to the address of data */

   data++;              /* data is now 1 */
    (*pData)++;         /* data is now 2 since we incremented what
                           pData pointed to */
}
```

Pointers (continued)

- Never write to an uninitialized pointer.

```
void examplePointer2(void)
{
   INT4 data=0;          /* data is an 4 byte integer */
   INT4 *pData;          /* pData is a pointer to an 4 byte integer and in not
                            initialized */
   INT4 *pData2;         /* pData2 is a pointer to an 4 byte integer and in not
                            initialized */

   pData = &data;       /* pData is now set to the address of data and is
                            initialized */

   pData2 = (INT4 *)malloc(sizeof(INT4 )); /* pData2 is now set to the address of data
                            and is initialized if the value is not NULL (0) */
   if (pData2 = NULL)
          return;        /* Error can't initialize pData2 */

}
```

Incrementing Pointers

- Pointers are always manipulated in units of values they are pointing to.

```
void examplePointer3(void)
{
    INT4 *pData;             /* pData is a pointer to an integer and in not
                                initialized*/
    INT1 *pByte;

    pData = (INT4 *)malloc(sizeof(INT4)*10); /* pData is now set to the address of data
                                             and is initialized if the value is not NULL (0) */
    if (pData = NULL)
            return;     /* Error can't initialize pData */

    *pData = 0x1234; /* sets the value pointed to by pData */

    pData++;          /* increments the address of pData by the sizeof an int */
    pData = pData + 5;/* increments the address of pData by the sizeof an int times 5 */

    /* want to do something odd I want to decrement the pointer by 5 (not by 5 time size
       of an INT4 */

    pByte = (INT1 *)pData;

    pByte += 5;
}
```

Pointers and Arrays

- An array is a pointer which is initialized and also has space allocated for it.

A simplistic way of representing it.

```
int array[10];
```

is similar to...

```
int *array;
array = malloc(sizeof(int)*10);
```

Except where memory is created is different.

Pointers and Arrays (Con't)


```
void examplePointer4(void)
{
   INT4 data[10];    /* data is an integer */
   INT4 *pData;       /* pData is a pointer to an integer and in not
                         initialized */
   pData = &data[0]; /* pData is now set to the address of data[0] and is
                         initialized and is now the same as data */

   pData[1] = 0;
   /* is the same as */
   data[1] = 0;
   /* another way of representing the same thing is */
   *(data+1) = 0;
   /* or */
   *(pData+1) = 0;
}
```

Function pointers

- Pointers can point to functions as well as data. Function pointers are commonly used in embedded systems.

```
INT4  test_func(INT4 a)
{
  printf("a = %x\n",a);

  return(a);
}

void testFptr(void)
{
  INT4 (* funcp)(INT4);     /* this is a variable declaration,
                               really! */

  funcp = test_func;
  (*funcp)(0x55aa55); /* execute function indirectly */
}
```

Those crazy structures?

- Bit fields
- Byte alignment
- Packing
- Unions

Bit fields


```
union
{
    struct
    {
        int a:1;
        int b:1;
        int c:6;
        int d:8;
        int e:16;
    } thirtytwobits;
    unsigned int allbitsmaybe;
} bitfielduninon;
```

- Bitfields are strictly a software convenience. They may not save memory and their order and representation in memory is not guaranteed.
- If you can to use them safely if you don’t expect or need a predictable representation in memory, you are far better off to map to memory in a portable fasion to use bit masking and arrays. (We will cover this a bit later as a refresher)

Byte alignment

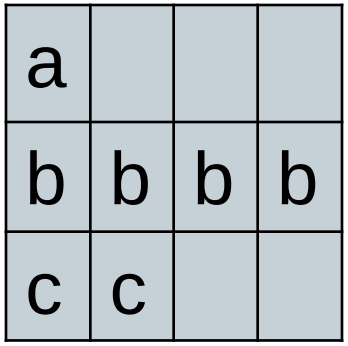

```
struct bytealignment
{
  char a;
  char b:
  short c;
  int d;
}
```

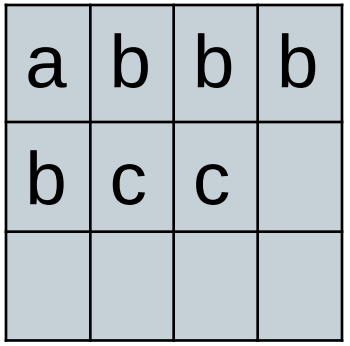
- What size will this be in memory?
- Guess what. You can't tell. It will differ by processor, compiler or even setting within the compiler.
- If you care how it is aligned in memory, allocate it or use an array and bit manipulate the values you want into it.

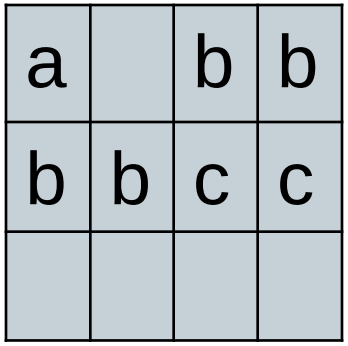
Structures - Byte alignment


```
struct
{
  char a; /* this field
  is 1 byte wide */
  int b; /* this field is
  4 bytes wide */
  short c; /* this field
  is 2 byte wide */

} myStruct2;
```

- The fields may be in memory like any of the following.







Packing

- Sorry there is NO standardized way to pack structures.
- Don’t count on any consistency in the way structures are packed.
- If you need something packed in memory, allocate it or use an array and bit manipulate the values you want into it.

Structures - packing

- When packing is supported packing forces. Usually via a keyword packed.
- However packing is NOT guaranteed to exist in all compilers and when is does exist it not always consistent.
- Some compilers do packing down to the bit others do not.
- Some processors don't handle unaligned data very well other do fine.
- Good practice when defining structures is to group variable types. (at least group by variable sizes if you know them) ALL EMBEDDED PROGRAMMERS SHOULD HAVE KNOWLEDGE OR THE VARIABLE SIZES FOR THEIR PROCESSOR/COMPILER but do you know your customers system

Do not use packing if you expect you code to work across multiple platforms. Pack it yourself by using arrays and other tools.

a	b	b	b
b	c	c	

Unions – They are there but be warned. They are NOT portable


```
union
{
  struct
  {
      int a:1; /* this field is 1 bit wide */
      int b:6; /* this field is 6 bits wide */
      int c:1; /* this field is 1 bit wide */
  } b;
  unsigned char reg;
} myunionstruct;
```

Unions Part 2


```
union
{
   struct
   {
          int a;
          int b;
   } part1;
   struct
   {
          char a;
          char b;
          char c;
   } part2;
} myunion;
```

- What size will this be in memory?
- Unions like bitfields are strictly a software convenience the they may not save memory and their order and representation in memory is not guaranteed.
- If you wish to use them for convenience that is safe if you expect a representation in memory “Don’t”.
- Guess what you can’t tell it will differ by processor, compiler or even setting within the compiler.
- Don’t count on any consistency in the way structures are represented in memory.
- Stay away from unions if you count on any representation in memory.

The ‘volatile’ keyword

- Examples without volatile

```
void testNoVolatile(void)
{
    unsigned int *hw_addr =
            (unsigned int *)0x12345678;

     *hw_addr = 1; /* this may be ignored */
     *hw_addr = 2; /* this may be ignored */
     *hw_addr = 3;
}

void delay(unsigned int times)
{
    unsigned int i;
    for (i=0; i< times; i++);
}


void testing(void)
{
    unsigned int *hw_addr2 =
            (unsigned int *)0x87654321;

     *hw_addr2 = 0x1000; /* this may be ignored */
     delay(1000000); /* hopeful – delay that may get optimized out */
     *hw_addr2 = 3;
}
```

The ‘volatile’ keyword (cont)

- Example with volatile

```
void testVolatile(void)
{
   volatile unsigned int *hw_addr =
        (unsigned int *)0x12345678;

    *hw_addr = 1; /* this will NOT be ignored */
    *hw_addr = 2; /* this will NOT be ignored */
    *hw_addr = 3;
}

void fixeddelay(unsigned int times)
{
   volatile unsigned int i;
   for (i=0; i< times; i++);

}
```

Static Keyword

- Two purposes for static: limiting the scope, specifying persistence
- Functions or variables can be static. A function or variable (declared outside of a function) is static then it is only visible inside that c file.
- If a static variable is declared inside a function, it acts like a global variable for that function. Its value is remembered from one call to the next time that function is called.

```
static int localfunction(char a)
{
   if (a == 'b')
       return 1;
   else
       return 0;
}

static int var;

int  globalfunction(int temp)
{
   static int staticvar=0;

   temp = staticvar++;

   return temp;
}
```

Registers vs... the Register keyword

- The keyword ***register*** was created to allow the programmer to force a variable into a CPU register. With better optimizers it is available a legacy feature that you should not use.

```
void example(void)
{
  register unsigned char c;
  /* more code here */
}
```

- CPU registers should not be confused with device registers.
- Also ***register*** variables can not be addressed.
- Also ***register*** is just a request not a guarantee.

Natural Size and Bit operations

- What happens when you do a bitwise invert on a numeric constant.

```
int a = 0xff7f;
if (~0x80 == a)
{
/* you may get here you may not –
   depends on how many bits ~0x80 gets converted to */
}
UINT16 a = 0xff7f;
if ((UINT16)(~0x80) == a)
{
/* you now control your destiny */
}
```

Lab 2

- Create a header file for the ATMEGA2560. Include register definitions for the Timers, the UART, the EEPROM, and all the PORTS.
- Add your new header file to LAB1 and cleanup the code to use your new header file.
- Make all you reads and writes go through a function or a macro.

- Review Part 2
- Critical Regions
- Verification
- Learning to read data sheets
- Lab 3

Binary Refresher

- Convert the following to binary and hex

1. 10
2. 20
3. 255
4. 128 Above this is unsigned bytes
5. -1Below this is signed
6. -10

Binary Refresher (cont)

- Convert the following to binary and hex (in 8 bits 1-4 are unsigned 5-6 are signed)

1. 10 – 1010 (0xa)
2. 20 – 10100 (0x14)
3. 255 – 11111111 (0xff)
4. 128 – 10000000 (0x80)
5. -1 – 11111111 (0xff)
6. -10 – 11110110 (0xf6)

- Note 255 and -1 are the same value in binary.
- The MSB (Most Significant bit is the sign bit)

Binary Refresher (cont)

- –1 is equivalent to specifying binary value of all 1's no matter what the variable type

  That is the importance of signed vs unsigned.

Signed Variables

- To create calculate the binary for a given binary value. (example for 8 bits)

1. Take the value minus the sign and invert each bit. (for example 3 become 00000011 then gets inverted to 11111100)
2. Then add one to the binary value – (11111101)
3. So -3 is 11111101 (0xfd)

Watch out for unexpected results


```
void func(void)
{
  signed char c;
  c = 124;
  c++; /* 125 */
  c++; /* 126 */
  c++; /* 127 */
  c++; /* 128 - nope -128 */
  c++; /* 129 - nope -127 */
}
```

Bitwise operations

- And – test bit(s) operation
- Or – set bit(s) operation
- Xor – toggle bit(s) operation
- Not – invert bits operation (whole variable)
- Shifting – shifts bits of variable left or right (many uses…)

Bitwise ‘and’

- a & b – a is bitwise anded with b

```
/* bitwise and */
unsigned char testBAND(unsigned char a, unsigned char b)
{
  return a & b;
}
```

- For clarity a && b – a is logically anded with b is not the same as a & b.

```
/* logical and */
unsigned char testLAND(unsigned char a, unsigned char b)
{
  return a && b;
}
```

Bitwise ‘and’ (cont)

What is returned for each of the following?

- testBAND(0,1)
- testLAND(0,1)
- testBAND(2,1)
- testLAND(2,1)
- testBAND(5,4)
- testLAND(5,4)

Bitwise ‘and’ (cont)

What is returned for each of the following?

- testBAND(0,1) - 0
- testLAND(0,1) - 0
- testBAND(2,1) - 0
- testLAND(2,1) - 1
- testBAND(5,4) - 4
- testLAND(5,4) - 1

Bitwise ‘or’

- a | b – a is bitwise ored with b

```
unsigned char testBOR(unsigned char a,
  unsigned char b) /* bitwise or */
{
  return a | b;
}

/* logical or */
unsigned char testLOR(unsigned char a, unsigned char
  b)
{
  return a || b;
}
```

Bitwise ‘or’ (cont)

What is returned for each of the following?

- testBOR(0,1)
- testLOR(0,1)
- testBOR(2,1)
- testLOR(2,1)
- testBOR(5,4)
- testLOR(5,4)

Bitwise ‘or’ (cont)

What is returned for each of the following?

- testBOR(0,1) - 1
- testLOR(0,1) - 1
- testBOR(2,1) - 3
- testLOR(2,1) - 1
- testBOR(5,4) - 5
- testLOR(5,4) - 1

Bitwise ‘not’

- ~a – a is bitwise not-ed

```
/* bitwise not */
unsigned char testBNOT(unsigned char a)
{
  return ~a;
}

/* logical not */
unsigned char testLNOT(unsigned char a)
{
  return !a;
}
```

Bitwise ‘not’ (cont)

What is returned for each of the following?

(assuming unsigned char)

- testBNOT(0)
- testLNOT(0)
- testBNOT(2)
- testLNOT(2)
- testBNOT(5)
- testLNOT(5)

Bitwise ‘not’ (cont)

What is returned for each of the following?

(assuming unsigned char)

- testBNOT(0) - 255
- testLNOT(0) - 1
- testBNOT(2) - 253
- testLNOT(2) - 0
- testBNOT(5) - 250
- testLNOT(5) - 0

Shifting

- a << b – a is bitwise shift-ed to the left

```
/* bitwise shift left */
unsigned char testLeft( unsigned char a, unsigned char b)
{
  return a << b;
}
```

- a >> b – a is bitwise shift-ed to the right

```
/* bitwise shift right */
unsigned char testRight( unsigned char a, unsigned char b)
{
  return a >> b;
}
```

- Watch out for sign extension when using shifting.
- This can be used for quick math and masking (we will see in the labs).

Bitwise shifting (cont)

What is returned for each of the following?

(assuming unsigned char)

- testLeft(1,0)
- testRight(1,0)
- testLeft(2,1)
- testRight(2,1)
- testLeft(5,3)
- testRight(5,3)

Bitwise shifting (cont)

What is returned for each of the following?

(assuming unsigned char)

- testLeft(1,0) - 1
- testRight(1,0) - 1
- testLeft(2,1) - 4
- testRight(2,1) - 1
- testLeft(5,3) - 40
- testRight(5,3) - 0

xor

- a ^ b – a is bitwise xor-ed with b
- Very useful for finding which bits have changed…

```
/* bitwise xor */
unsigned char testXOR( unsigned char a,
  unsigned char b)
{
  return a ^ b;
}
```

Bitwise xor (cont)

What is returned for each of the following?

(assuming unsigned char)

- testXOR(0,0)
- testXOR(1,0)
- testXOR(1,1)
- testXOR(2,1)
- testXOR(5,3)

Bitwise xor (cont)

What is returned for each of the following?

(assuming unsigned char)

- testXOR(0,0) - 0
- testXOR(1,0) - 1
- testXOR(1,1) - 0
- testXOR(2,1) - 3
- testXOR(5,3) - 6

Combinations of Bit Manipulation

- First you can create a macro to find out if one specific bit is set.

```
#define BIT(n)    ((unsigned int) (1 << (n)))
/* this would then get anded with a variable */
```

- Get the max value for a given number of bits.

```
#define MAXVAL(n)   (((unsigned int)(1<<(n)))-1)
/* this could then be used as a mask for a portion of a
  value */
```

Read Only Registers

- Similar to Read and Clear. Registers which read and write mean different things.
- Watch for these. They require special attention. Possible software storage of previous read or write values.

You Have Been Warned

Recursion and Reentrancy

- Recursion on embedded systems is fine but must be controlled and limited. Otherwise you will run out of space on the stack (we will talk about this soon)
- Reentrancy is the ability of a function to be call multiple times at the same time.
  1. This can be done by using only local variables.
  2. Locking critical regions.
  3. Using different control structures
  4. Etc.

Recursion and Reentrancy (Example)

- See recursive.cpp on the website

Write your code as if...

- it would be shipped as source.
- your boss and your next boss will be seeing it.
- write it so you can be proud of it.
- write it so that one of you peers who may support it will not get mad at you.

DMA

- What is DMA?
- What is DMA used for?
- What do I have to watch out for?

What is DMA?

- Direct Memory Access.
- This is where you tell you hardware that there is a block(s) memory that it can access directly via the hardware with minimal intervention by you.

What is DMA used for?

- Networking cards – most packet based devices.
- Video cards

What do I have to watch out for with DMA?

- Absolute control over representation in memory is important.
- Remember the hardware controls, what it expects in memory and how it needs to be presented.
- Watch out for the cache.
- Also watch out for notifications (the driver notifying the hardware and the hardware notifying the driver).

So what is so important about the sequence

- As a driver writer you job is to make the hardware work. Getting the right bits to the hardware is only part of the battle.
- If I tell you to drive to the airport and I say it is 5 miles north of here as the crow fly's you can't exactly drive that way and make it there alive and with a ticket. You need specific directions.
- I have to give you directions. So does the hardware designer. Some are in front of you, others are implied.

Counters

- What order do I read the registers if I have a multi register counter? That is not latch-able.
- Simple...(one possible way)

1. Read the upper bits
2. Then read the lower bits
3. Then the upper bits
4. If the second read of the upper bits is greater then the first read of upper bits then lower bits rolled over.

Timing

- Just like the real world, some things are timing based. If you tell some one that you need their feedback at 5pm, you may have a problem if they show up late.
- The same is true for hardware. If the counter overflows every 10 ms, and you read it every 20 ms. You will have a problem. Even if you read the timer every 10 ms, that may not be good either if your timing is a little off or because something more important happened before you could read it.

Lets go over a simple portable driver example

- On my web site I now have a complete example for you to actually use.

Timing – Keeping up with the clock

- Sleep function in a loop
- Interrupt driven time
- Signal driven

Timing - Sleep function


```
void task(void)
{
  while(1)
  {
     sleep(10);
     dostuff();
  }
}
```

Timing – Interrupt Driven


```
void timerInterrupt(void)
{
      dostuff(); /* this gets call every so often */
}
```

Timing - Sleep function


```
timerInterrupt()
{
     wakeTaskSignal()
}
task()
{
  while(1)
  {
     waitForSignal();
     dostuff();
  }
}
```

How many MIPs does it use?

- This is a setup. Don't fall for it.
- Say it takes X% of Y processor and a given number of megahertz. Based upon that and you expect it to take N% in another processor at another frequency.
- MIPs does not mean anything. Everything from caches to external memory speed to bus width impacts this.
- When moving between CPUs, the variations of instructions impacts things drastically and they are depend on exactly what the code is doing and the CPU and compiler optimization, etc.

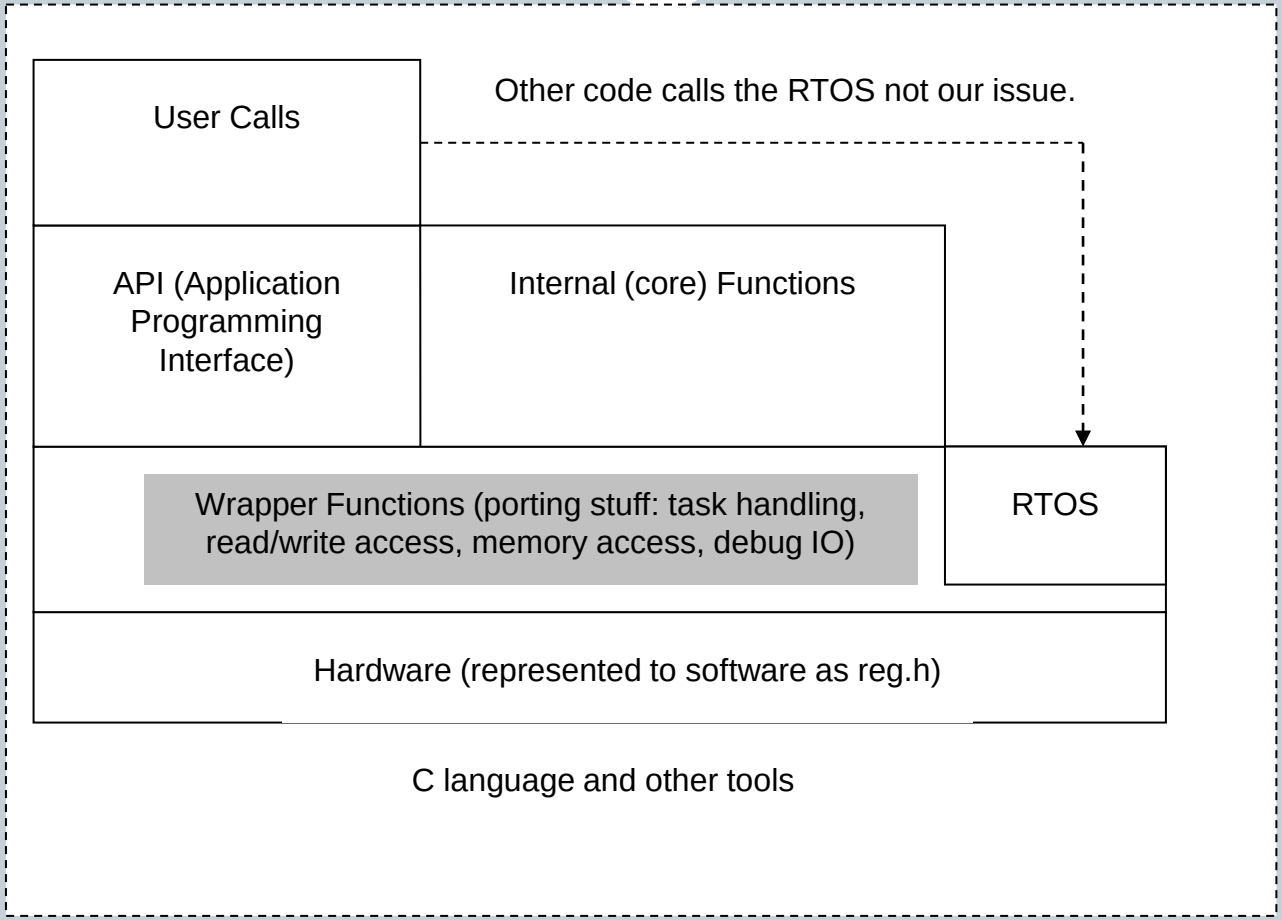
“Trust but verify” Ronald Reagan

- Work with your hardware folks. They need your help as much as you need their help.
- Data sheets are not always the full truth.
- Even compilers make mistakes.

Lab 3

- Write a the driver to read the switches next to the LEDS and cycle the LEDS from 0 to which ever switch is pressed if none are pressed cycle through all of them with a delay that is easily controllable via a complier constant. Use the timer for controlling the delay not the spin loops that were in Lab 1 and 2.

Wrapper Functions



Wrapper functions

- Wrapper functions are functions that may need to have OS or system awareness to work. Put them into a port.h or port.c and keep them separate from the rest of your portable functions.
- readReg
- writeReg
- startTask
- taskDelay
- setISR
- getBaseAddr

Wrapper example readreg()/writereg()


```
#define readreg(addr, data)  data = *(volatile Uint32 *)((addr * ADDR_MULT) + baseaddr)
#define writereg(addr, data)  *(volatile Uint32 *)((addr * ADDR_MULT) + baseaddr) = data
```

or

```
void writereg(CTRL *ctrl,Uint32 addr, Uint16 data)
{
    *(volatile Uint16 *)((addr * ADDR_MULT) + ctrl->baseaddr) = data;
}

Uint16 readreg(CTRL *ctrl,Uint32 addr)
{
    return *(volatile Uint16 *)((addr * ADDR_MULT) + ctrl->baseaddr);
}
```

Interrupt

- An interrupt is the way hardware notifies software that an event occurred without software having to check for it.
- In the simplest view, it is a hardware line from the device attached to a line on the CPU saying “hey I have a message if you care”.
- You can ignore it. You can respond to it. Or you can mask it.

Interrupt Service Routines (ISR)

- On a given interrupt, an ISR is the routine that is called upon receipt of that interrupt or group of interrupts.
- Interrupts should be kept short and sweet and not wait on other things. If you need to wait, defer those things until you are out of the interrupt.
- Some things must be done.

1. See if this is for you device(s)
2. Clear the cause (make it so you will not get beaten up with the same thing over and over again)
3. Take care any critical responses.
4. Signal any deferred processing.
5. If the is no deferred processing make sure you re-enable interrupt.

Interrupt Service Routines (Example)

- On my web site I now have a complete example for you to actually use.

(Lets take a look at it)

Tasks

- A task is function that does not end for a relatively long time. But, it may pause and wait for other things.

Critical Regions

- A critical region is a chunk of code that you don't want to be interrupted by another chunk of code. (i.e. modifying a counter when that counter is used my another counter or function)
- Here are some way to protect critical regions.
  1. Task Locks
  2. Interrupt Locks
  3. Semaphores
  4. Architectural Methods – Queuing, Messaging

Critical Regions

- A sequence that should not be paused in the middle by some or all of the rest of the system.
- For example: when updating statistics you may not want the task which is reading it to access it as it is being updated.
- Say you are changing modes you may want to hold off any other operations until it is complete.

What does a lock/unlock function look like


```
/* recursive.c rentrant code */

#include "stdafx.h"
#include "string.h"

#define MAX_DEPTH                256
#define MAX_FILE_NAME_LEN   256

static unsigned int depth;

struct
{
        char filename[MAX_FILE_NAME_LEN];
        int  line;
} log[MAX_DEPTH];

volatile unsigned char lockvar;

/* unitialize variables for logging */

void initLog(void)
{
        lockvar = 0;
        depth = 0;
}

/* lock example */
void lock(void)
{
        /* might add block interrupts as well */
        do
        {
                lockvar++;
                if (lockvar != 1)
                {
                        lockvar--;
                        /* optionally sleep */
                }
        }
        while (lockvar != 1);
}
```

What does a lock/unlock function look like


```
/* unlock example */
void unlock(void)
{
        lockvar--;
        /* might add unblock interrupts as well */
}

/* function to do a circular log */
void logit(char *filename, int line)
{
        lock();
        strncpy(log[depth].filename,filename, MAX_FILE_NAME_LEN);
        log[depth].line = line;
        depth++;
        if (depth >= MAX_DEPTH)
        {
                depth = 0;
        }
        unlock();
}

/* two task that may run at similair times */
void taskA(void)
{
        logit(__FILE__,__LINE__);
}

void taskB(void)
{
        logit(__FILE__,__LINE__);
}

int main(int argc, char* argv[])
{
        recurFunc(2);
        initLog();
        taskA();
        taskB();
        printf("Hello World!\n");
        return 0;
}
```

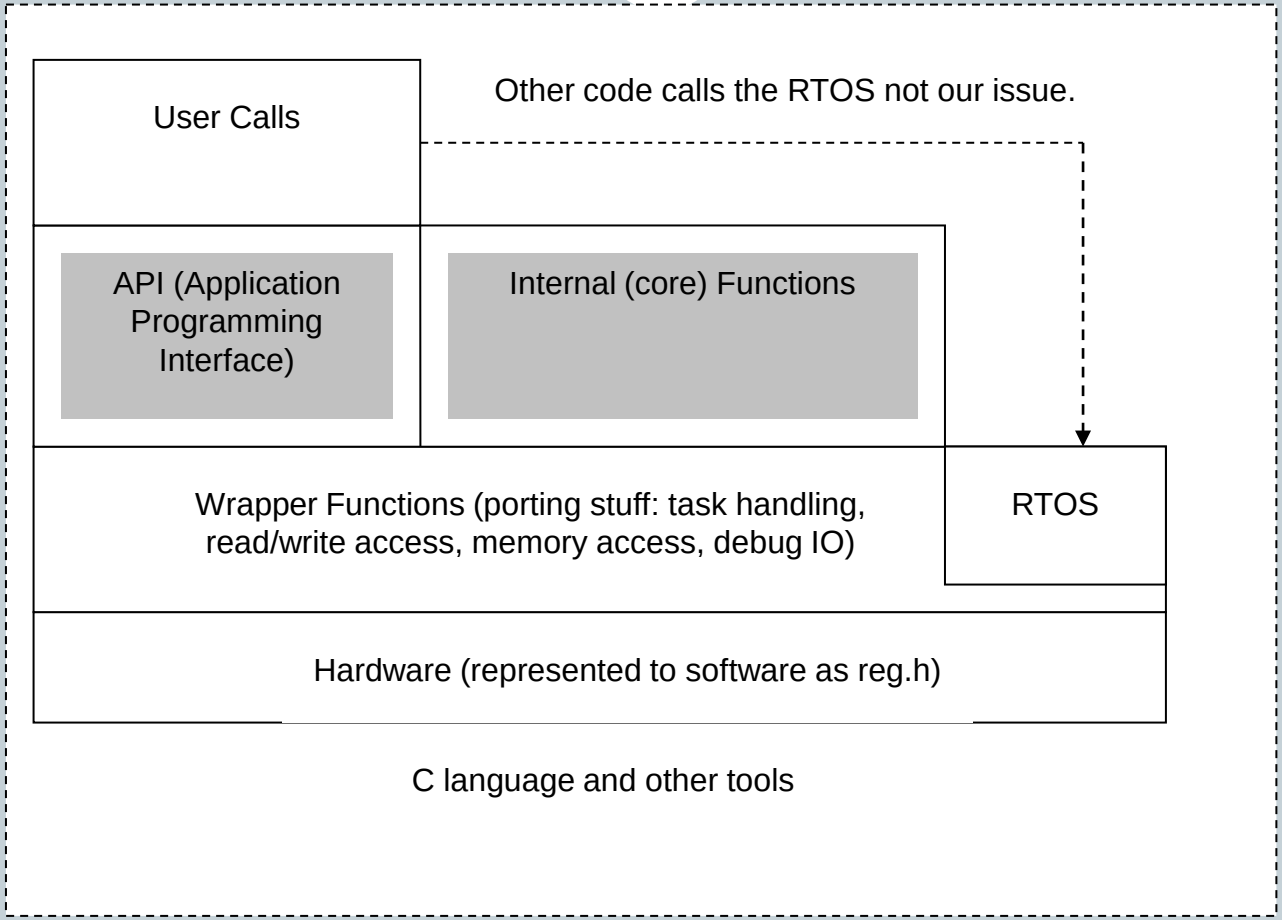
Polled vs... Interrupt

- If you plan on supporting interrupts, why not do both. It is not much more code.
- It is easier to test in polled mode first.
- Interrupts can get you quick response time but can make your system performance less predictable. Why not have the interrupt wake up a task and poll. So then you can free up easier. That what is called deferred processing.

Deferred Processing Routine (DPR)

- A DPR can be either a task or a function that is called after an ISR to handle all the non super time critical stuff.

APIs & Core Functions

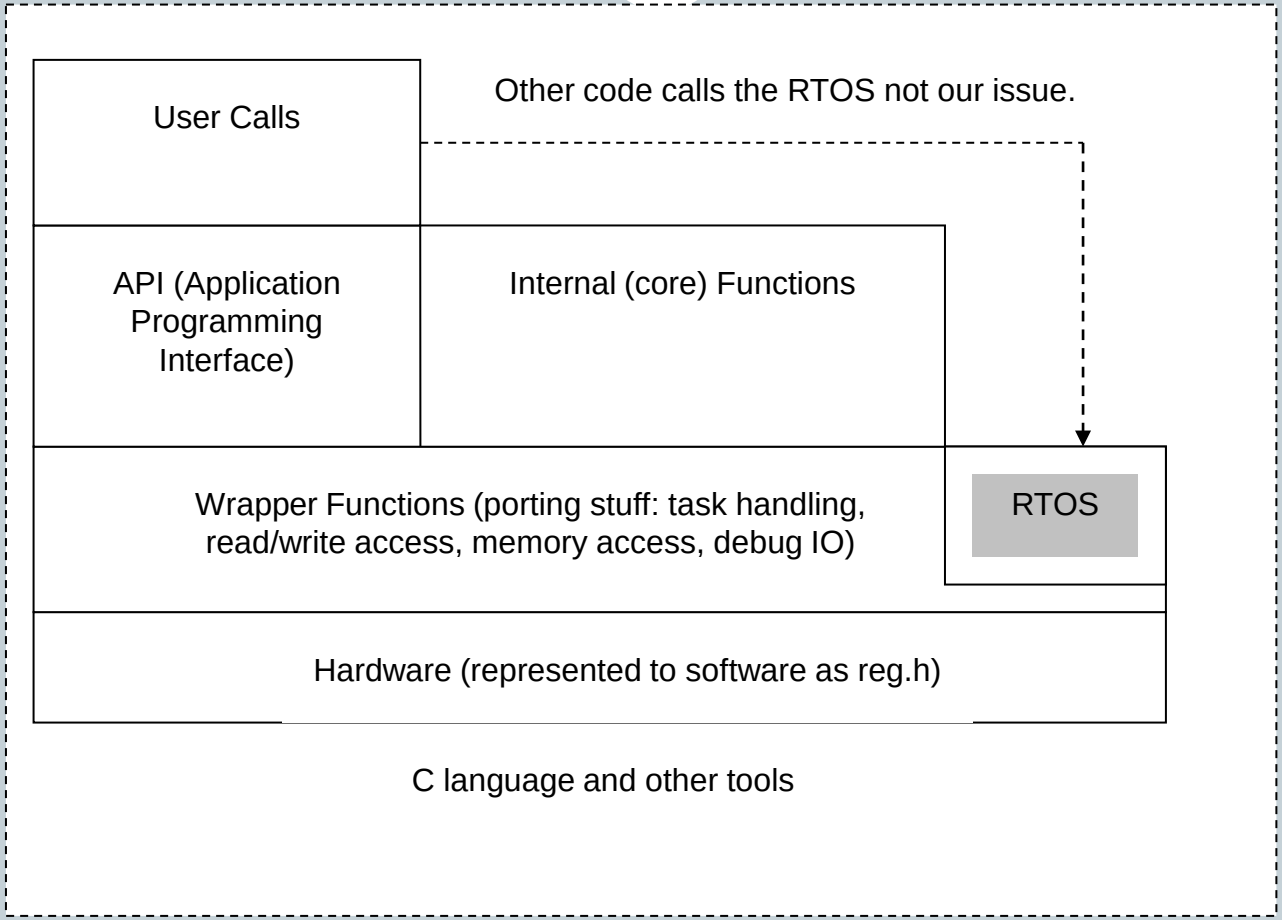


APIs vs.. Core Functions

- APIs are functions you want the user to call. They are the documented interface.
- Porting functions may be APIs also.

- Core functions are internal to the code and are not intended for direct call by any code external to the driver.

The RTOS interface



RTOSes and Embedded Drivers

- Standardized RTOSes are not in 70 to 90% of all embedded systems.
- An OS is not required for most drivers it may me required for the application that needs the driver.
- Most OSes have some of the same basic features but the interfaces to them differ significantly. Refer to the website for VxWorks and POSIX list.
  1. Start/stop a task.
  2. Send a message/event
  3. Delay a task
  4. And many more...

  But, the list and what the function is called can be vastly different between RTOSes.

RTOS or not, Write Your Driver so you don't care

- Just understand what your driver needs and you can resolve much of it yourself with or without an OS.

Optimizers

- Optimizers are you friend and enemy at the same time.
- Their job is to make your code as small as possible and/or as fast as possible.
- Because they follow your instructions you are at fault if they screw up.
- Primary purpose of volatile is to tell the compiler that this variable may be modified by something else, such as hardware. (ah, embedded programming). Therefore every access must re-read the variable from memory before any manipulations. This is also useful to defeat compiler optimizations.

That is why ***volatile*** is so important. Also that was why before optimizers had evolved the keyword ***register*** was created.

Optimizers

- It is part of the compiler that is supposed to make your code more efficient then you wrote it.
- Good news is they basically work but they can not read minds.

```
int before_optimizer (int a, int b)
{
	int temp;
	int *ptemp = &temp;

	*ptemp = a;
	*ptemp = *ptemp + b + 3*2 + b;
	return *ptemp;
}
Int after_optimizer (int a, int b)
{
	return a + (b<<1) + 6
}
```

- But if writing to *ptemp was important…

```
Int after_optimizer2(int a, int b)
{
	volatile int temp;
	volatile int *ptemp = &temp;

	*ptemp = a;
	*ptemp = *ptemp + (b<<1) + 6;
	return *ptemp;
}
```

Caches

- Cache's improve performance but can add complexity.
- Make sure your device register's are not in cache-able memory (configure the cache to not cache your register space). If this is not possible or not practical then you will need to add some extra logic to your read and write functions to flush the cache.
- Caches also impact DMA functionality.

More eyes the merrier

- Use all the tools you have around you to verify your code.
- Have peers look at you code
- Ask you hardware folks to look at critical areas.
- Readability is important. Comment it.

"Leave style and formatting out of the code review. Bring the results of all your computer generated checks to the review"

Lab 3.5 - the code review

- Pick a person who you don't work with to review your last two labs and offer suggestions on how to make it more portable. The object here is constructive criticism the grade here is for the reviewer.

Lab 4 – Fix lab 2 and 3 + a new one

- Based upon what we learned with volatiles and our other feedback fix lab 2 and 3.
- Then using the cleaned up lab 3 as a starting point write a driver and application to read keys coming in via the Spare RS232 on the ATMEL board and echo them back. When a control Z is hit, start recording the keys to the ATMEGA2560's build in EEPROM and when the control z is hit again stop recording. If it is turned back on, start at the beginning again. When control y is hit then start playing back the recorded values. However the light and switches much work like they did in lab 3. To make this behave that way you should make the leds cycle in an ISR.
  1. Remember since it is in eeprom it should remain through a power cycle.
  2. Include carriage return linefeed expansion in you code.
  3. 9600, 8, n, 1
  4. Continue to allow the lights to cycle
  5. For some extra credit make the UART interrupt driven.

Question

- If you did not read ahead could you have given me each of the labs without the changes you just did? (without the incorporated stuff from Lab 3.5 or without my feedback)

- Well that is version control. Although not part of this course, it is very important to learn. If you change something someone may want the old one back. Even if is just for comparison.

Those crazy customers

- They pay you paycheck.
- They need your help and you need theirs.
- Ask your customers what interface is best for them when you can.
- Use their feedback to make you driver as simple as you can for them. However don't forget portability. So, taking their feedback does not always mean copying one customer's API.

Working around those pesky unplanned hardware features

- Some things are BUGs (errata)
- Others are just annoying behavior
- Some are documentation issues.

Give your hardware folks feedback. Unless they know it is a problem they will repeat it again and again.

Document in your code the undocumented features

Keeping up with errata

- Hardware documentation is not always right.
- Hardware has problems. (Undocumented features and flaws)

Hardware folks don't call the bugs they call them ERRATA

If you write your code flexible enough you can save rewrites to fix bugs with both software and hardware features.

Use Your Common Sense is the Most Important Thing to Writing a Good Portable Driver

- Minimize the number of global variable.
- Make as many of your internal functions static.
- Use good names for your functions and variables.
- Don't use common names in your global function or variables. They may collide with other functions or variables.
- Keep the majority of your code being 100% portable.
- Keep the portable stuff separate from the rest of the code.
- Make it easy to modify you code that needs to be changed for portability sake.

C++ and drivers

- Most C compilers are really C++ compilers but, remember those customers using C++ can use a C driver, not the reverse.
- When possible, compile you code in C mode with C++ type checking. C++'s type checking is far better and more picky then C's.
- However, stay away from the C++ features. A driver writer must know exactly what his or her driver is doing, not having C++ do things behind you back.

Tools of the trade

- Debuggers
- Simulators
- JTAG emulators (or similar emulators)
- ICE (In-Circuit Emulators)
- Logic Analyzers
- Scopes
- Meter
- Pattern Generators
- Debug IO
- Code coverage tools (software and hardware)
- Co-Simulation
- Co-Debug
- Anything else you can dream of.

Debuggers

- Are extra software running on the platform under test.
- Sometimes all or part are built into the OS.
- May impact performance and flow of your code.
- Limitations, may include not being able to debug ISRs or paths that the debugger is using.

Simulators

- Simulators are running you code on a similar environment to what you are testing on.
- Try building and running you code on a PC particularly if you have a simulation environment.
- Does not emulate all hardware particularly external to what is expected.

Emulators

- Emulators come in two major flavors JTAG emulators (or similar emulators) and ICE (In-Circuit Emulators)
- JTAG emulators rely on the CPU to have hooks to give it access to the internals of the CPU to allow it control over break points and registers.
- ICEs get control by physically replacing the CPU with a compatible device with complete control.
- JTAG emulators are less expensive particularly as CPUs get faster. But generally have more limitations although not always (very CPU dependent)
- ICE can be very expensive. But, their features are limited only but creativity of the vendor and money.

Logic Analyzers, Scopes, Meters and Pattern Generators

- This is usually the realm of the hardware folks. It should not be just for them.
- Not all devices have any human readable IO.
- Learn how to use these tools. Get someone to show you. Instead of asking is the signal coming out, ask can you help me attach the analyzer so I can see if the signal is coming out correctly.

Debug IO

- Debug IO is as simple as the traditional print.
- Where is my IO coming out of. Is it debugged yet. What if that is what I am debugging.

Code coverage tools (software and hardware)

- Have I exercised all of my code. Have I exercised all of the hardware. This is were code coverage tools can help.
- Hardware tools are very expensive in terms of money software tools can be expensive in terms of CPU power.

Use any tools your mind can come up with to test the code and hardware

- Be creative. The last thing you want is a simple bug to be blamed on you. If you catch it before someone else does then you don't end up on the hot seat.

Co-Simulation, Co-Debug and Co-Development

- The tools are finally here.
- Some are affordable
- Some are not
- Work Smarter Rather then Harder
- Develop in SW/HW Parallel truly

Guess what

- As of this point you have all the pieces to start writing portable device drivers.
- More importantly, you have learned how to recognize what is not portable and may give you and your future customers problems down the road.

Now comes real world practice.

Longevity

- Keep you skills up to date. Read, learn and teach and learn from those around you.
- Don’t be afraid to try new things. Failures are a way to learn.
- Adapt to change.
- Be proud of what you do and why you do it.
- Learning the technology of the day is not as important as learning the building blocks of the technology. Expand you roots don’t grow one deep root then may not allow you do adapt.
- Most of all enjoy what you do. Have fun.

Too Many Engineers Are Afraid to Experiment

- Try new things.
- Try new tools. Try to be more efficient.
- Work smarter rather then harder.
- Sometimes is faster to try to several experiments then to design multiple attempts that have not been tried.
- If you try an experiment document it so you can learn and other can learn.
- Make you can verify success or failure of your test.

Do as much as you can before the hardware comes in?

- Code
- Documentation
- Test
- Experimentation
- Simulation

Portability is important but functionality is key

- Design for portability.
- But, coding is filled with choices.
- It must work first otherwise you code has failed.
- Be extra carefully or share memory structures that rely on specific memory layouts.

Make the right choices.

Don’t forget timing

- Looping through code a fixed number of times is often better then a variable number of times.
- Fixed timing is better then variable timing and less time is better then more.
- Don’t forget the optimizer. Clean code compiles best. Volatiles...

  Counting clocks is the way of real time software

There are multiply ways to do the same thing

- There is no wrong way of doing something as long as works. But, you need to be sure it works. Test, Debug and Test.
- Compiling without Errors or Warning
- Turn all warnings to errors. Compiler or Linker.
- Lint
- Automation is and designing for test are key.
- Co-Development and Co-Simulation.
- Use what ever you can and every tool in your tool chest… No exceptions…

Symmetry – Where possible..

- If you can write it make sure you can read it back.
- If you can transmit make sure you can receive...
- Offer loopbacks at the lowest level of your software and in your hardware.

Design your code to allow the applications above it to be tested.

Find a Buddy

- Find a friend or a college and use each other as a sounding board and extra set of eyes.
- Learn from each other. Likes and dislikes. Think lab 3.5.

Testing requirements are the easier they testing all of your code

- Testing requirements is only a portion of your test.
- How much of your code are you testing?
- How much of your hardware are you using?
- Are you random testing for the things that are not planned.
- Test you code with the optimizer turned onto it maximum setting.

Putting all together in one package – delivering a driver

- Code
- Build instructions – make sure the code can be compiled without special compiler settings.
- Porting Instructions
- API documentation
- Scheduling

Scheduling does not me an RTOS

- RTOSes are not required. Timing is required
- Here are some options...
  1. RTOS multiple tasks
  2. Main app with interrupt driven events possibly multiple timers.
  3. Main app tight loop no interrupts.
  4. A list of tasks that run to completion. Function pointers called in a loop when their timer has ended.
- Keep it simple
- If use an RTOS use one based on your needs not because it was used before.

Used before or I don't know is the mostly likely reason for choosing an RTOS

Use Your Common Sense is the Most Important Thing to Writing a Good Portable Driver

- Minimize the number of global variable.
- Make as many of your internal functions static.
- Use good names for your functions and variables.
- Don't use common names in your global function or variables. They may collide with other functions or variables.
- Keep the majority of your code being 100% portable.
- Keep the portable stuff separate from the rest of the code.
- Make it easy to modify you code that needs to be changed for portability sake.

Reminder

- Software is simply the sequence of code required to get the proper bits in and out of the proper bytes in the right sequence with the correct timing and present them to the user of the code in a way they can manipulate the process without violating the hardware rules and without violating common sense.
- The user of the software can either be another piece of software or user buy way of some hardware.
- This all needs to be done in a way that is both portable, readable and maintainable.

Things not to forget

- Volatile – use them
- Its your fault if your can not be optimized
- Who's Initializing you global variables
- Cut and paste from the datasheet better yet generate your headers in parallel with your datasheet.
- Write you code around your hardware features and create and API to match those features. Then map them to the OS/RTOS API.

Go Over My Lab 4

- Lets talk about my Lab 4 and Lab 5 (also).

Open Season

- I want all you questions on embedded software. Anything...
  - I have talked about the technology direction.
  - The job market
  - Any other details...

You Control The Class from This Point On

Enjoy

Reminder I will always be your teacher

When or if you have questions feel free to ask whether it be weeks or months from now.

All That is Left is for you to put this to practice

Books and References

- A C Reference Manual (fifth edition)
  by Samuel P. Harbison III & Guy L. Steele Jr.
  Copyright 2002, Prentice Hall
- C Traps and Pitfalls
  by Andrew Koenig
  Copyright 1989, Addison Wesley