Network Embedded Systems
Sensor Networks

Tips and Tricks

Marcus Chang, mchang@cs.jhu.edu

Project Part 3

- PC --UART--> Telosb --radio--> Telosb
- Reliable communication and storage
  - Data corruption
  - Missing data
- Optimize
  - Minimize code size and memory usage
  - Minimize power
  - Maximize bandwidth

Project Part 3

- Python script
  - Writer thread
    - Take user input
    - Encode
    - Transmit over UART
  - Reader thread
    - Receive over UART
    - Decode

- Warning
  - Remember to use thread safe data structures
    - E.g. Queue, threading.Event

Project Part 3

- Hints
  - Human readable vs. machine readable
  - Move or aggregate functionality
- Profile
  - romsize.pl build/telosb/main.exe
- Disable inlining for profiling
  - CFLAGS += -fno-inline
- Set radio packet length to 100
  - CFLAGS += -DTOSH_DATA_LENGTH=100

Watchdog Timer

- Exercise 8
  - Setup the watchdog to reset if not fed every 250 ms

```
// WDTPW:    WDT password
// WDTCNTCL: clear counter
// WDTSSEL:  select ACLK
// WDTIS0:   divide by 8192
WDTCTL = WDTPW | WDTCNTCL | WDTSSEL | WDTIS0;

// check WDT flag
if (IFG1 & WDTIFG)
  led2On();
else
  led0On();
```

Watchdog Timer

- Exercise 8
  - Use infinite loop in main-function to test feeding

```
while(1)
 {
   // TimerA overflows every 64 ms
   // reset WDT count down / feed the watch dog
   WDTCTL = WDTPW | WDTCNTCL | WDTSSEL | WDTIS0;

   // infinite loop for testing
   while(infinite)
   {
     …
   }

   _BIS_SR(LPM0_bits | GIE);          // Enter LPM0 w/ interrupt
 }
```

Watchdog Timer

- Pitfalls
  - The watchdog starts on and is either on or off.
  - The IE1 |= WDTIE; is used when the WDT is used as a regular timer (like TimerA/B)
  - Feeding the watchdog directly from a periodic timer will not work as intended

```
void feedWatchDog() {
  WDTCTL = WDTPW | WDTSSEL | WDTIS0 | WDTCNTCL;
}

startTimer(0, 100, 1, feedWatchDog);
```

Watchdog Timer

- Better use of periodic timer or idle task

```
void feedWatchDog() {

  if ( checkMain()          // is the main loop progressing as intended
  && checkSamples()         // is the node collecting the min/max number of samples
                            // and are they within the expected bounds
  && checkRadio()           // is the radio duty-cycle and send/receive within bounds
  …                         // other checks
  ) {
    // MCU running as intended, feed the watchdog
    WDTCTL = WDTPW | WDTSSEL | WDTIS0 | WDTCNTCL;
  }
}

startTimer(0, 100, 1, feedWatchDog);
```

Watchdog Timer

- Controlled reboot
  - How to save state across reboots?
    - E.g. last function called, reason for reboot, etc.
  - Some registers are left unchanged across reboots

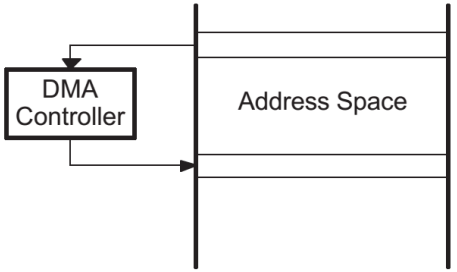
Register	Short Form	Register Type	Address	Initial State
ADC12 control register 0	ADC12CTL0	Read/write	01A0h	Reset with POR
ADC12 control register 1	ADC12CTL1	Read/write	01A2h	Reset with POR
ADC12 interrupt flag register	ADC12IFG	Read/write	01A4h	Reset with POR
ADC12 interrupt enable register	ADC12IE	Read/write	01A6h	Reset with POR
ADC12 interrupt vector word	ADC12IV	Read	01A8h	Reset with POR
ADC12 memory 0	ADC12MEM0	Read/write	0140h	Unchanged
ADC12 memory 1	ADC12MEM1	Read/write	0142h	Unchanged
ADC12 memory 2	ADC12MEM2	Read/write	0144h	Unchanged
ADC12 memory 3	ADC12MEM3	Read/write	0146h	Unchanged
ADC12 memory 4	ADC12MEM4	Read/write	0148h	Unchanged
ADC12 memory 5	ADC12MEM5	Read/write	014Ah	Unchanged

Direct Memory Access

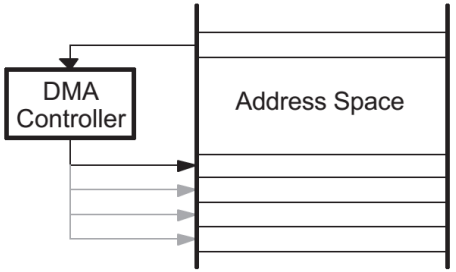
- Exercise 7
  - Copy block of addresses to block of addresses
  - Evaluate performance
    - DMA is faster than regular copy
    - DMA lets the CPU work on other tasks

Direct Memory Access

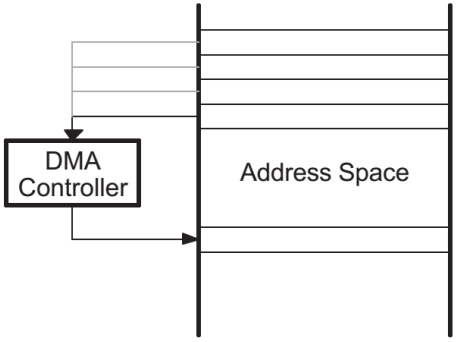
- Other uses?
  - Move data to/from Special Function Registers
    - Move ADC readings, UART characters to buffer
    - Feed UART/radio/etc with data from buffer (or other SFR)



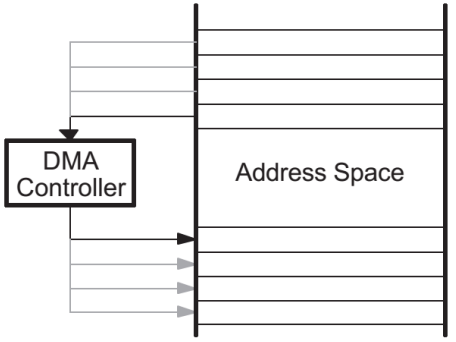
Fixed Address To Fixed Address



Fixed Address To Block Of Addresses



Block Of Addresses To Fixed Address



Block Of Addresses To Block Of Addresses

Goals

- Write code that is easy to maintain
  - Code that is easy to understand
  - Keep bugs out
- Write efficient code
  - Reduce memory and energy consumption
  - Tradeoff between readability
- Easy to port
  - Same code, different platforms
  - Same code, different compilers

Readability

- Use const keyword wherever it is appropriate
  - const means Read-Only
  - Tells other your intent
  - Allows compiler to optimize
  - Constant variables can be kept in ROM
- Don’t over optimize
  - Divide when division, shift when shift
  - The compiler will often detect division by 2

Data Types Revisited

- int and long int are ill-defined
  - Width depends on architecture; leads to bugs when ported
- Fixed width types
  - C99, stdint.h
  - uint8_t, uint16_t, uint32_t, etc.

Data Types Revisited

- Problem with fixed width
  - Does not take advantage of CPU support
    - E.g. 32 bit might be faster than 16 bit if CPU has special hardware
- Portable data types:
  - uint_least8_t
    - Smallest integer guaranteed to be at least 8 bit
  - uint_fast8_t
    - Fastest integer guaranteed to be at least 8 bit

Signed and Unsigned

- Avoid signed integers unless
  - Variable is obviously signed, e.g., human readable temperature
  - Interacting with standard C functions that uses int
- Why?

Signed and Unsigned

- Some MCU do not support signed integers in hardware
  - The software library will add overhead
- Division
  - 2^N divisions has to be performed by division instead of shifts
- It often saves an extra comparison in if-statements
  - Unsigned always > 0
- Many built-in standard C functions/macros returns unsigned integers
  - sizeof, offsetof, etc.
- Special Function Registers are defined as unsigned

Signed and Unsigned

```
uint16_t a,b,c, res;

a = 0xFFFF; //Max value for a uint16_t
b = 1;
c = 2;

res = a;
res += b; //Overflow
res -= c;

res = ?
```

```
int16_t a,b,c, res;

a = 32767; //Max value for a int16_t
b = 1;
c = 2;

res = a;
res += b; //Overflow
res -= c;

res = ?
```

Signed and Unsigned

- Overflow on signed integers is undefined
  - The compiler can do whatever it wants
  - Switching compiler might lead to different results

Signed and Unsigned

- Modulo operator
  - What happens if user enters -1?

```
int main(void) {
  int i;
  printf("Enter a number: ");
  scanf("%d", &i);

  if( ( i % 2 ) == 0) printf("Even");
  if( ( i % 2 ) == 1) printf("Odd");

  return 0;
}
```

Signed and Unsigned

- Modulo operations are implementation specific
  - -1 % 2 = -1 on some platforms
- Shifting/masking is implementation specific
  - Shifting number into sign or sign into number

Signed and Unsigned

- Mixing signed and unsigned
  - What does the function print out?

```
void foo(void) {
  unsigned int a = 6;
  int b = -20;
  (a+b > 6) ? puts("> 6") : puts("<= 6");
}
```

Signed and Unsigned

- Mixing signed and unsigned
  - What does the function print out?

```
void foo(void) {
  unsigned int a = 6;
  int b = -20;
  (a+b > 6) ? puts("> 6") : puts("<= 6");
}
```

- Signed integers are promoted to unsigned integers
  - In 2-complement, b becomes a very large positive integer

Compiler Optimization

- Optimize flag is either for speed or size
  - Size
    - Useful for reaching target platform
  - Speed
    - If the code already fits the target platform
    - Speed decreases energy consumption
    - Speed often reduces code size as well

Compiler Optimization

- What can go wrong?
  - Compiler changes order of operations
    - Breaks device specific ordering
      - E.g. flash driver needs specific command order to enable writes
  - Compiler removes “redundant” code
    - Code might have been there to ensure specific timings
    - Volatile variables not declared properly
  - Compiler replaces common code with functions
    - Nested functions can increase the call stack
      - Can lead to stack overflow

Global Variables

- Although the memory space is global, global variables can lead to code that is difficult to read and maintain
  - How many places is the variable used and how?

- Use modules to organize code
  - Static functions and variables are local to the modules
  - Improves code size
    - Compiler can better inline, analyze registers, perform “short” jumps instead of “long” jumps with static functions

Aliasing

- Accessing a variable in more way than one
  - Difficult for the compiler to optimize code
  - Compiler cannot make any assumption about buf

```
char *buf

void clear_buf()
{
  int i;

  for (i = 0; i < 128; ++i)
  {
    buf[i] = 0;
  }
}
```

```
void clear_buf(char *buf)
{
  int i;

  for (i = 0; i < 128; ++i)
  {
    buf[i] = 0;
  }
}
```

Switch-case Statements

- What does the compiler do?
  - if-else if-else chains
  - Jump table
  - A mix of both
- Variable vs. fixed run time
  - The compiler chooses order and method
  - Adding new cases later might change the runtime drastically
    - Use contiguous case values or highly disparate to avoid sudden changes by the compiler

Switch-case Statements

- An array of function pointers will have fixed runtime

```
void test(uint8_t const jump_index)
{
 static void (*pf[ ])(void) = {fna, fnb, fnc, …, fnz};

 if (jump_index < sizeof(pf) / sizeof(*pf))
 {
   /* Call the function specified by jump_index */
   pf[jump_index]();
 }
}
```

Compiler Oddities

- Post vs. Pre-increment/decrement
  - ++i can be faster than i++
  - Splitting up a = *ptr++ in two lines can be faster

```
foo = a[i++];

can be executed as

 foo = a[i];
 i = i + 1;
```

```
 i = 0;
 while (a[i++] != 0)
 {  ...  }

has to be executed as

loop:
   temp = i; /* save the value of the operand */
   i = temp + 1; /* increment the operand */
   if (a[temp] == 0) /* use the saved value */
      goto no_loop;
   ...
   goto loop;
no_loop:
```

Compiler Oddities

- Counting down can be faster than counting up
  - Comparing against zero is often more efficient

```
for (uint8_t lpc = 0; lpc < 10; ++lpc)
{
 foo();
}

can be executed as

INC lpc      ; Increment loop counter
SUB lpc, #10 ; Compare loop counter to 10
BNZ loop   ; Branch if loop counter not equal to 10
```

```
for(uint8_t lpc = 10; lpc != 0; --lpc)
{
 foo();
}

can be executed as

DEC lpc   ; Decrement loop counter
BNZ loop  ; Branch if non zero
```

Lookup Tables

- Operations that are performed often and repeatedly can be put in a lookup table
- Tradeoffs code size with runtime speed

- Example
  - Count the bits in a byte
    - Shift each bit and add them
  - Lookup table
    - Calculate all 256 possible combinations and store in array in ROM
    - Use byte to index array

Modulus operator

- A % B
  - A – B * (A/B)
  - Reverse order

```
void compute_time(uint32_t time)
{
 uint32_t    days, hours, minutes, seconds;

 seconds = time % 60UL;
 time /= 60UL;
 minutes = time % 60UL;
 time /= 60UL;
 hours = time % 24UL;
 time /= 24UL;
 days = time;
}
```

```
void compute_time(uint32_t time)
{
 uint32_t    days, hours, minutes, seconds;

 days = time / (24UL * 3600UL);
 time -= days * 24UL * 3600UL;
 /* time contains the number of seconds in last day */
 hours = time / 3600UL;
 time -= (hours * 3600UL);
 /* time contains the number of seconds in last hour */
 minutes = time / 60U;
 seconds = time - minutes * 60U;
 }
```

Power of 2 Buffers

- Access and index check can be done efficient

```
#define RX_BUF_SIZE (32)
#define RX_BUF_MASK (RX_BUF_SIZE - 1)

static uint8_t Rx_Buf[UART_RX_BUF_SIZE]; /* Receive buffer */
static uint8_t RxHead = 0; /* Offset into Rx_Buf[ ] where next character should be written */

__interrupt void RX_interrupt(void) {
   uint8_t rx_char;
   rx_char = HW_REG;            /* Get the received character */
   RxHead &= RX_BUF_MASK;       /* Mask the offset into the buffer */
   Rx_Buf[RxHead] = rx_char;    /* Store the received char */
   ++RxHead;                    /* Increment offset */
}
```

Unused Interrupt Vector

- Do nothing
  - System may crash
- Put Return From Interrupt command in interrupt vector
  - Bug may prevent system from sleeping
- Explicitly declare all ISR
  - Useful when learning new platform
  - Disable interrupt in unused ISR
  - Put trap function to catch bugs

Examples from

- Michael Barr
  - Barr Code, http://embeddedgurus.com/
- Nigel Jones
  - Stack Overflow, http://embeddedgurus.com/
- Jan-Erik Dahlin, IAR Systems
  - “Writing optimizer-friendly code”

Schedule

- Week 1: Introduction and Hardware
- Week 2: Embedded Programming
- Week 3: Medium Access Control
- Week 4: Link Estimation and Tree Routing
- Week 5: IP Networking
- Week 6: JHU Special feat. Doug Carlson
- Week 7: (seminar, no lecture)
- Week 8: Energy Management and Harvesting
- Week 9: Review and Midterm
- Week 10: Time Synchronization
- Week 11: Localization
- Week 12: Embedded Programming Part 2
- Week 13: (seminar, no lecture)
- Week 14: TBD