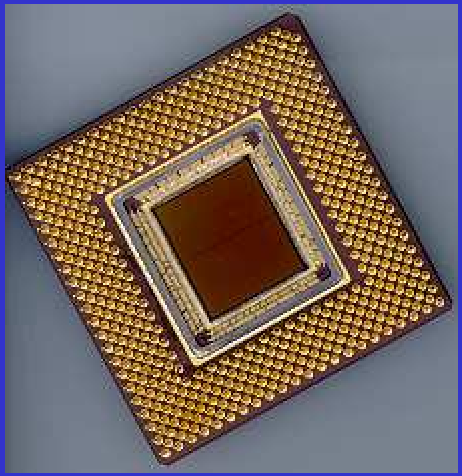


ECE 477 Digital Systems Senior Design Project

Module 8
Embedded Software Development

Outline

- Memory Models
- Memory Sections
- Discussion
- Application Code Organization

Memory Models - 1

- What are the primary differences between general-purpose processor memory models and embedded processor memory models?
  - “Flat” memory model (typically no memory hierarchy or virtual memory)
  - Limited (fixed, “non-infinite”) SRAM data space and Flash program space
  - “Non-homogeneous” memory types
    - SRAM – “read/write” (volatile unless battery backup used)
    - Flash – “read only” (non-volatile in-circuit, sector-erasable and reprogrammable)
    - EEPROM – “read mostly” (non-volatile in-circuit, byte-erasable and reprogrammable)

Memory Models - 2

- How do these differences in memory models influence way in which high-level language code is written?
  - Don’t use too high a level of abstraction
    - Avoid use of big library routines (e.g., printf)
    - Avoid dynamic memory allocation
    - Avoid complex data structures
    - Avoid recursive constructs
    - Watch declarations (char, int, long)
  - Treat “C” like a “macro-assembly” language
  - Remember that floating point support is emulated by lengthy software routines
  - Remember that using table lookup might be a better approach for transcendental functions (sin, cos, tan, log) than calculation via software emulation

Memory Models - 3

- Where do I/O devices appear in the memory model?
  - Depends on processor architecture
    - Most are memory-mapped (devices appear in processor's memory address space)
    - Some (e.g., Rabbit, x86) have separate I/O and memory spaces
  - High-level language instruction syntax may be different to address memory-mapped vs. I/O-mapped devices
  - May also be a different syntax for accessing input vs. output ports

Memory Sections - 1

- Text section
  - Executable instructions (code)
- Data section
  - Initialized global or static data (variables)
- BSS (block start section)
  - Uninitialized global or static data (variables)

Memory Sections - 2

- Run time sections
  - Stack –
    for local variables and parameter passing
    (also, context switch save/restore)
  - Heap –
    dynamic memory allocation (BSS)

Discussion - 1

- What does "real time" mean (or, what are the key characteristics of a "real time" system)?
  - there are "mission critical" timing constraints (usually tied to input/output data sampling rates and/or data processing overhead)
  - service latencies are known and fairly tightly bounded
  - are typically "event-driven"
  - require low overhead context switching

Discussion - 2

- What is the difference between a “time sharing” OS and a “real time” OS? (cite examples)
- Time sharing OS (Unix/Linux/Windows)
  - runs as many users/tasks (quasi-simultaneously) as possible
  - utilizes “time-slice” scheduling – relative priority can be assigned by adjusting size of time slice
  - has little/no concern for service latencies – task scheduling is a “big (round robin) loop”
  - utilizes fixed context switch rate (for Unix/Linux, about 200 Hz)
  - has little concern for amount of context switching overhead (e.g., page thrashing)

Discussion - 3

- What is the difference between a “time sharing” OS and a “real time” OS? (cite examples)
  - Real time OS (QNX,VRTX, Embedded Linux)
    - tries to minimize, bound service latencies
    - utilizes preemptive, multi-tasking scheduling
    - requires process threads that can be prioritized
    - requires multiple interrupt levels

Discussion - 4

- What does “fail safe” mean in the context of embedded software (firmware) development?
  - *A **fail-safe** or **fail-secure** device is one that, in the event of failure, responds in a way that will cause no harm, or at least a minimum of harm, to other devices or danger to personnel.*
  - cite examples of “fail-safe” device behavior
  - cite examples of “non fail-safe” device behavior

Application Code - 1

- What are some possibilities for organizing embedded application code?
  - polled program-driven

    “round robin” polling loop

    advantage: simple!

    disadvantage: large number of devices $\Rightarrow$ big loop $\Rightarrow$ large latency

Application Code - 2

- What are some possibilities for organizing embedded application code?
  - interrupt-driven (vectored or polled)

    sometimes called “event driven” $\Rightarrow$ all processing (after initialization) is in response to interrupts

    may want CPU to “sleep” between interrupts to reduce power consumption

Application Code - 3

- What are some possibilities for organizing embedded application code?
  - command-driven or "flag"-driven (also referred to as "state machine")

    "hybrid" of program-driven and interrupt-driven

    ⇒service routines "activated" based on (ASCII string) commands received

    ⇒alternately, activities of polling loop can be controlled by state of various "flags" set by interrupt service routines (e.g., in response to button presses, time slice expiration, etc.)

Application Code - 4

- What are some possibilities for organizing embedded application code?
  - real-time OS kernel (timer-interrupt driven)

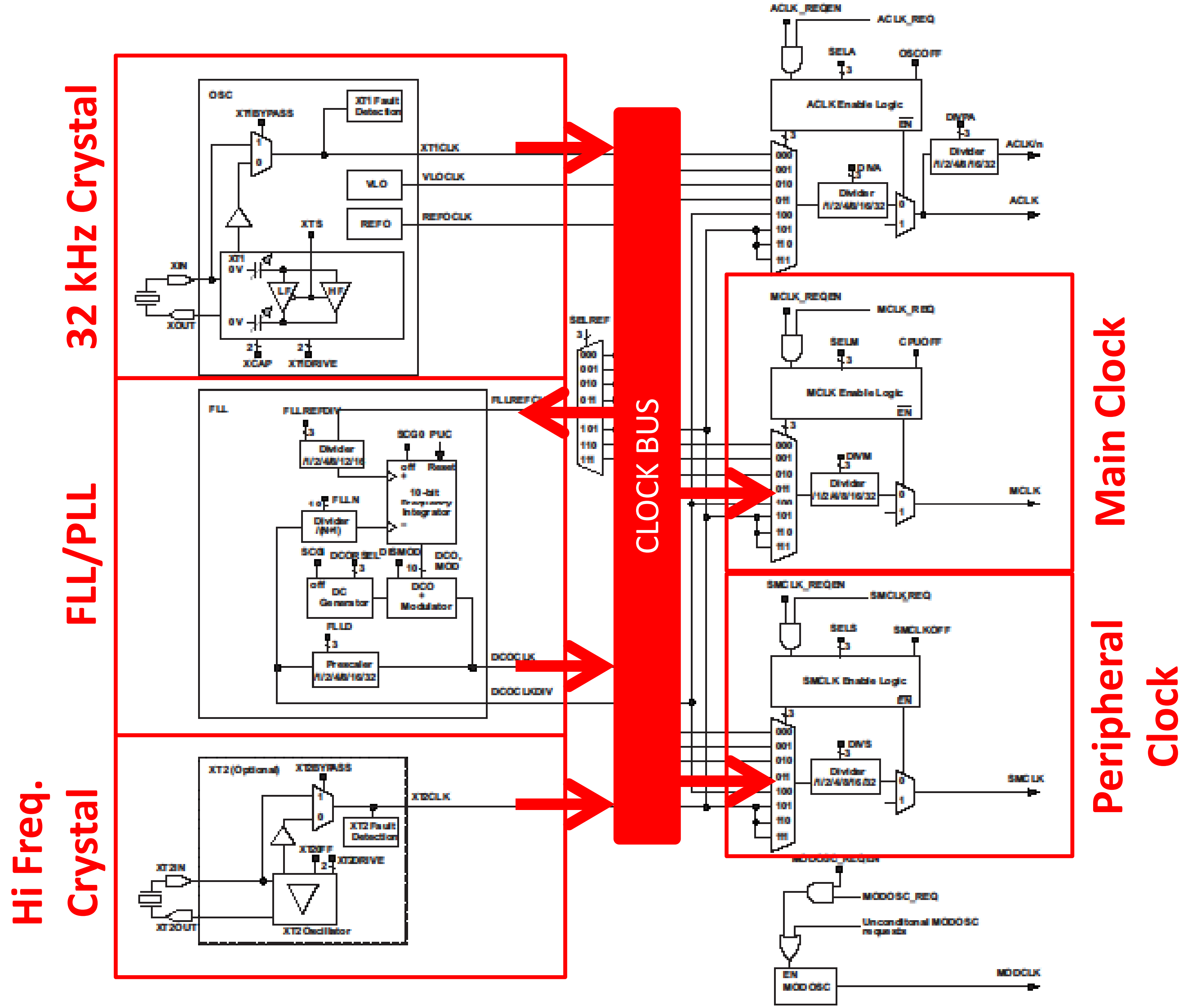
    data structure provides list of currently enabled tasks (can be dynamically inserted/deleted)

    $\Rightarrow$ periodic interrupt (RTI) used to determine when tasks rolled in/out

    $\Rightarrow$ can vary relative priority of enabled tasks by changing time slice allocation

Oscillators and Clocks

- Arguably the most important configuration for an embedded system
- All systems depend on clock configuration
- Should be the second block initialized
  - The first is the watchdog timer
- Higher clock speed means more computations per second and more power use
- What does a typical oscillator block look like? (next slide)



32 kHz Crystal
FLL/PLL
Hi Freq. Crystal
CLOCK BUS
Main Clock
Peripheral Clock
OSC
XT1BYPASS
XT1 Fault Detection
XT1CLK
VLO
VLOCLK
REFO
REFOCLK
XTS
XIN
XOUT
XCAP
XT1DRIVE
FLL
FLLREFDIV
FLLREFCLK
SELREF
SCG0
PUC
FLLN
SCG1
DCORSEL
DISMOD
DCO, MOD
DC Generator
DCO + Modulator
FLLD
Prescaler /1/2/4/8/16/32
DCOCLK
DCOCLKDIV
XT2 (Optional)
XT2BYPASS
XT2CLK
XT2 Fault Detection
XT2OFF
XT2DRIVE
XT2IN
XT2OUT
XT2 Oscillator
ACLK_REQEN
ACLK_REQ
SELA
OSCOFF
ACLK Enable Logic
DIVPA
DIVA
ACLK/n
ACLK
MCLK_REQEN
MCLK_REQ
SELM
CPUOFF
MCLK Enable Logic
DIVM
MCLK
SMCLK_REQEN
SMCLK_REQ
SELS
SMCLKOFF
SMCLK Enable Logic
DIVS
SMCLK
MODOSC_REQEN
MODOSC_REQ
Unconditional MODOSC requests
MODOSC
MODCLK

Simple Pseudocode

```
// 32 kHz watch crystal input on XT1
OSC_1_CONFIG_REG = CHOOSE_XT1;

// 32 kHz watch crystal = 32,768 Hz
// 32,768 Hz * 64 = 2,097,152 Hz (~2 MHz)
PLL_CONFIG_REG = CHOOSE_OSC_1 + MULTIPLY_BY_64;

// Main clock ~2 MHz
MAIN_CLK_CONFIG_REG = CHOOSE_PLL + DIVIDE_BY_0;
#define MAIN_CLOCK_FREQ 2097152 // in Hz

// Peripheral clock ~500 kHz ( 2097152 / 4 = 524288 )
PERIPH_CLK_CONFIG_REG = CHOOSE_PLL + DIVIDE_BY_4;
#define PERIPH_CLOCK_FREQ 524288 // in Hz
```

What’s with the #define???

```
// Set up the system timer interrupt at 1048 Hz
unsigned short divider = PERIPH_CLOCK_FREQ / 1048;
// 500.27 in this example -> 500 after integer truncation
TIMER_1_CONFIG_REG = CHOOSE_PERIPH_CLOCK + divider;
```

- If peripheral clock changes later, this code will NOT need to be modified
- Creates a robust software design
- These ideas aren’t just limited to clocks and timers (external component values, etc.)

More #define Tricks

- Get input from GPIO

```
#define BUTTON_1_MASK     0x04
#define BUTTON_1_PRESSED  ( PORT1_IN & BUTTON_1_MASK )
```

- Drive output pins

```
#define LED_1_MASK   0x20
#define LED_1_ON     ( PORT1_OUT |= LED_1_MASK )
#define LED_1_OFF    ( PORT1_OUT &= ~(LED_1_MASK) )

if( BUTTON_1_PRESSED ) {
  LED_1_ON;
} else {
  LED_1_OFF;
}
```

More #define Tricks

- Use macros to inline simple functions

```
// Utilize truncation to round a number
#define MAX( x , y )  ( (x) > (y) ? X : y )
if( MAX( adc_val_1 , adc_val_2 ) > 255 ) { ... }
```

- Create settings

```
#define __DEBUG_MODE__
#ifdef __DEBUG_MODE__
  printf( “Debug mode ON\n” );
#endif
```

- Error messages during compile time

```
#error Useful to use with different settings options.
```