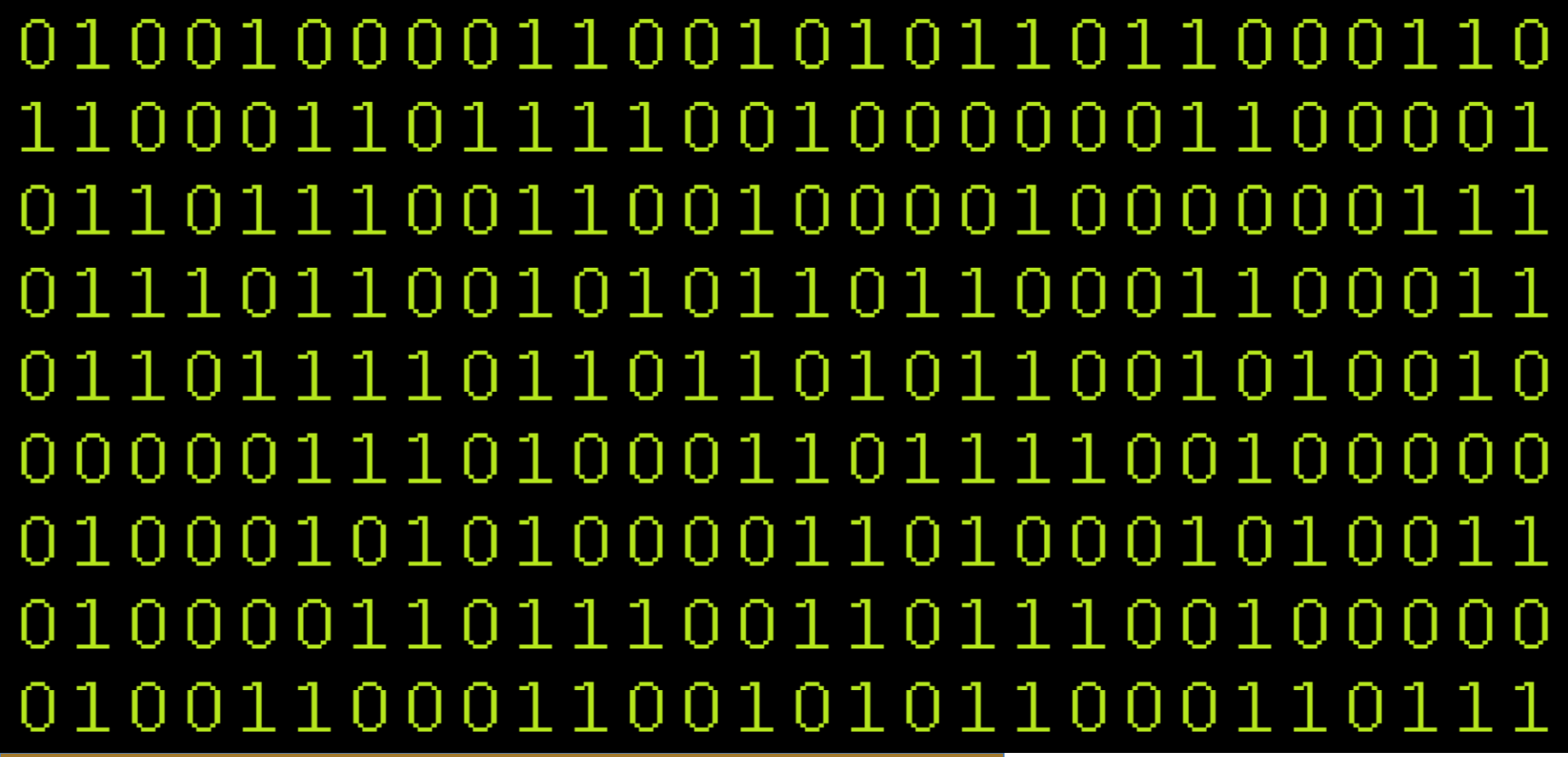



# EMBEDDED SOFTWARE DEVELOPMENT



PURDUE UNIVERSITY

# OUTLINE

- Embedded vs. General Purpose Programming
- Layers of Abstraction (Hardware, Interface, Application)
- Embedded Programming Models
- Real Time Operating Systems
- Definition of “Real Time” Systems
- Definition of “Fail Safe” Systems
- Revision Control Systems
- Firmware Design Techniques

# EMBEDDED VS. GENERAL PURPOSE

- What separates embedded and general purpose processor models?
  - “Flat” memory model (no virtual memory, hierarchy, or cache typically)
  - Limited SRAM data space and Flash program space
  - “Non-homogenous” memory types (SRAM, Flash, EEPROM, etc.)
  - Hardware interrupts – far more common in embedded programming than general purpose

# EMBEDDED VS. GENERAL PURPOSE

- How do these differences influence the way in which code is written?
  - Avoid use of large library routines (i.e. printf)
  - Avoid dynamic memory allocation
  - Avoid complex data structures
  - Avoid recursive constructs
  - Increased awareness of declarations (char, int, long)
  - C code generally written in “macro assembly” style
  - Remember: floating point support emulated via lengthy software routines
  - Pre-compiled values (table lookup) sometimes better than software-based calculation methods (trig)

# LAYERS OF ABSTRACTION

- Objective: well-written software which is portable, easy to understand, maintainable, and has good performance
- Solution: Separate software into various abstracted layers
  - Hardware: Direct drivers for various hardware devices (example: “send/receive data from SD card using SPI)
  - Physical: Layer between base hardware and “software”; hardware-specific details are abstracted (example: store/retrieve data from memory)
  - Application: User-specified device functionality (example: play a selected MP3 file)
- Tradeoff: Increased abstraction increases portability, maintainability and clarity while reducing performance (memory requirements and speed)

PURDUE UNIVERSITY

# EMBEDDED PROGRAMMING MODELS

## Program-Driven Approach

- Polled (Program-Driven) Approach:
  - All code and checks are performed in a single loop
  - Advantage: code is simple to write and understand
  - Disadvantage: Latency of code difficult to determine; as code becomes more complex latency may grow very large

```
while(1) {
  timer--; //decrement timer
  if(timer == 0) {
    //execute timer-dependent code
  }
  //check if button was pushed
  debounceButton();
  ...
}
```

# EMBEDDED PROGRAMMING MODELS

## Flag-Driven Approach

- Flag-Driven (State Machine) Approach:
  - Interrupt subroutines used to determine if certain conditions have been met (e.g. button presses, pending SPI/UART/I2C data, timer expiration) and set “flags”
  - Main loop then checks if flags have been set and runs corresponding code
  - Improved performance over polled designs; program flow slightly less straightforward

```
while(1) {
  if(timer0Flag) {
    //execute timer0 code
  }
  if(button1Pressed) {
    //execute button1 code
  }
  if(uartDataRdy) {
    //execute UART code
  }
  ...
}
```

PURDUE UNIVERSITY

# EMBEDDED PROGRAMMING MODELS

## Event-Driven Approach

- Interrupt-Driven (Event-Driven) Approach:
  - Main loop performs device initializations then idles
  - All processing (post initialization) is performed in response to prioritized interrupts
  - Processor may sleep between interrupts to reduce power consumption
  - Advantages: High performance code, low power operation
  - Disadvantages: Program flow may be difficult to follow and understand

# EMBEDDED PROGRAMMING MODELS

## RTOS Approach

- Real Time Operating System (RTOS) Approach:
  - All currently enabled system tasks are maintained within a data structure (tasks can be dynamically inserted or deleted)
  - Timing interrupts used to determine when tasks must be rolled in or out
  - Priority of enabled tasks can be changed by altering a task's time-slice allocation

# REAL TIME OPERATING SYSTEMS

- A few popular RTOS kernels:
  - ChibiOS/RT: Compact, high performance RTOS for 8/16/32 bit microcontrollers
  - FreeRTOS: Small, compact RTOS supported on many different microcontroller families (34 at time of writing)
  - SafeRTOS: High-security variant of FreeRTOS
  - Integrity: High-security proprietary RTOS, used in military jets. Guaranteed computation times.

# REAL-TIME SYSTEMS

## Definition of "Real Time" Systems

- "Our system will operate in real-time." (But what does "real-time" mean?)
- Real-time systems possess "mission critical" timing constraints (sampling rates, processing latency, etc.)
- System latencies are known and tightly bounded
- Typically event-driven
- Low overhead context switching

# FAIL-SAFE SYSTEMS

## Definition of "Fail Safe" Systems

- Fail-safe devices are designed in such a way that a failure will cause minimum or no harm to other devices or personnel
- Examples of fail-safe behavior:
  - A computer controlled lock that is capable of being opened, without power, from the secure side of the lock
  - A milling device or tool that features a hardware-based emergency stop (device can be overridden without computer intervention)
  - A UAV that initiates an emergency landing when power drops below critical levels

# REVISION CONTROL SYSTEMS

## Revision Control Systems

- Software is an increasingly complex and involved activity, and managing changes is a vital skill for any developer
- Revision control systems provide the user with the ability to track and manage changes, restore source code from previous changes, and share changes with others
- A few revision control systems:
  - git: Popular compact distributed version control system
  - mercurial: Simple, distributed version control system
  - Subversion (svn): Open source centralized version control system

# FIRMWARE DESIGN TECHNIQUES

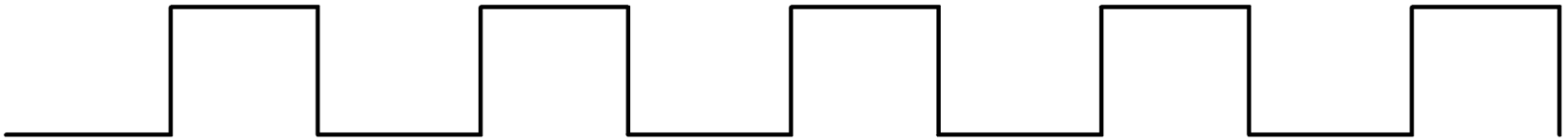
## Device Configuration

- Microcontrollers feature sets of configuration registers or bits which control their operation. Examples of device configuration options include:
  - Which clock source the microcontroller uses at startup
  - Which I/O lines will be used for programming functions
  - Device behavior in the event of a “brown-out” (brown-out reset, or BOR)
  - Behavior concerning watchdog timers (WDT)
  - Device behavior in the event of a stack under/overflow
  - Write protection
- Configuration bits must be set within the code or IDE for a device to function properly

# FIRMWARE DESIGN TECHNIQUES

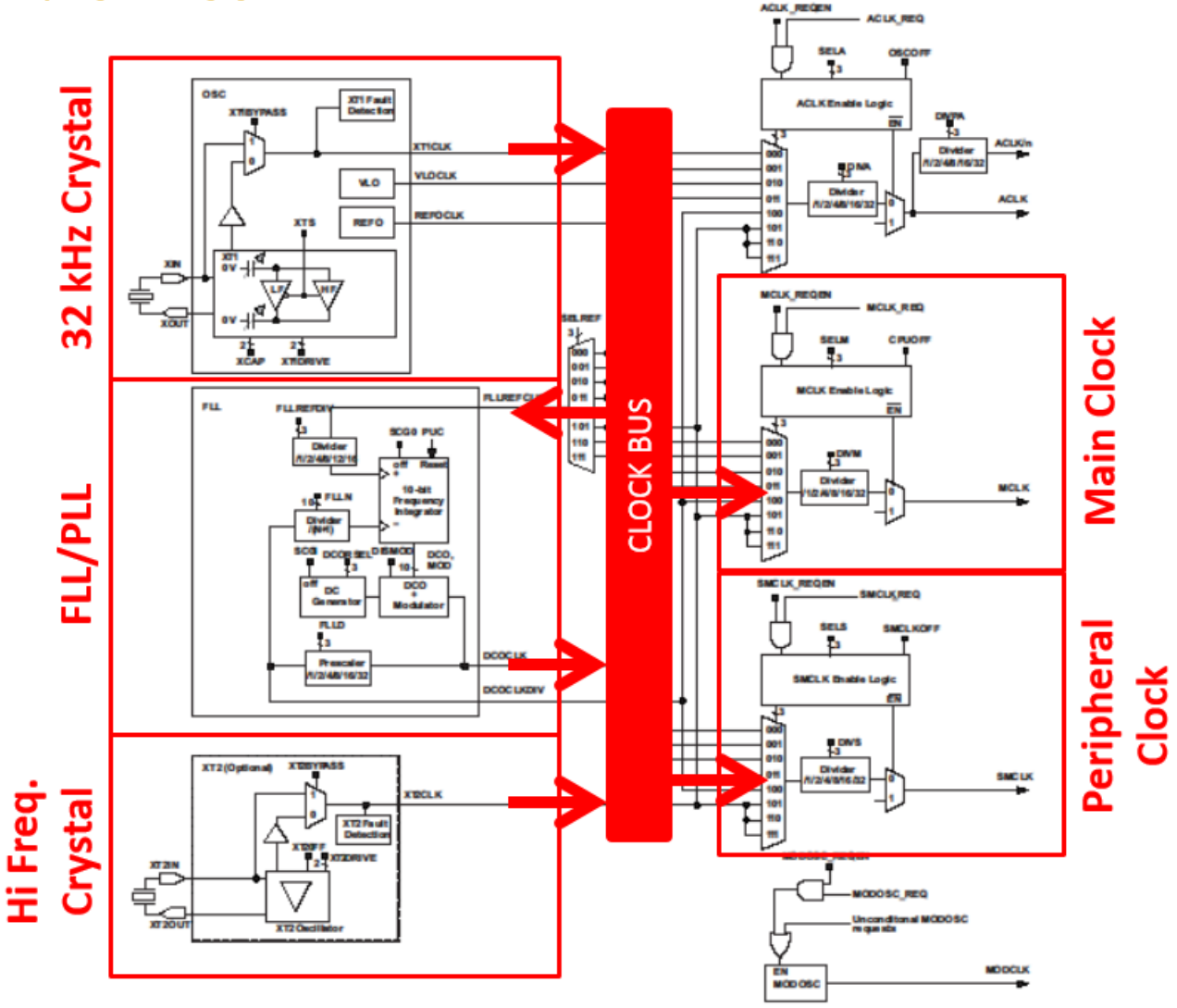
## Oscillator Configuration

- Oscillators are the basis of all logic in a programmable device, and thus one of its most important components
- All programmable devices support clocking from a variety of internal and external sources
- Nearly all programmable devices support the ability to switch clock sources and scalers during device operation
- Higher clock speed improves computational performance but burns more power



# FIRMWARE DESIGN TECHNIQUES

## Example Oscillator Block




PURDUE UNIVERSITY

# FIRMWARE DESIGN TECHNIQUES

## Oscillator Configuration

Some example pseudocode (note the use of #define):

```
// 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
```

# FIRMWARE DESIGN TECHNIQUES

## Power Configuration

- To conserve power, microcontrollers can be placed into a sleep state and then later awoken (via external interrupt, watchdog timer time-out, etc.)
- Power management detailed in device datasheet

FIGURE 3-5: TRANSITION TIMING FOR ENTRY TO SLEEP MODE

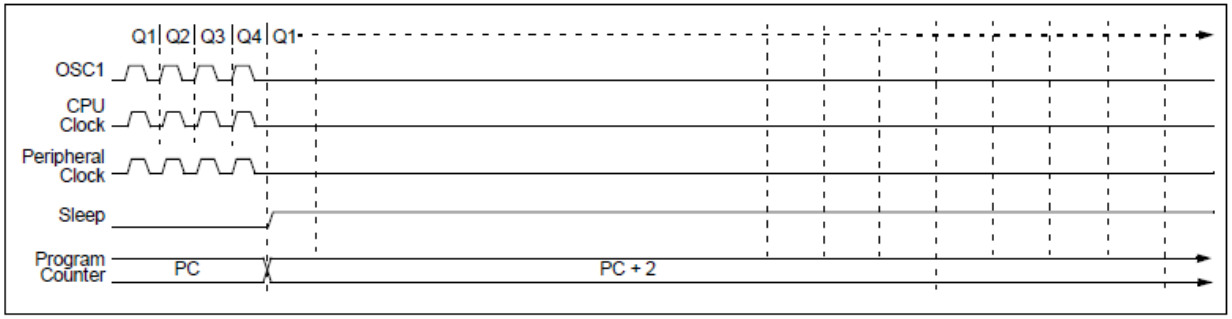
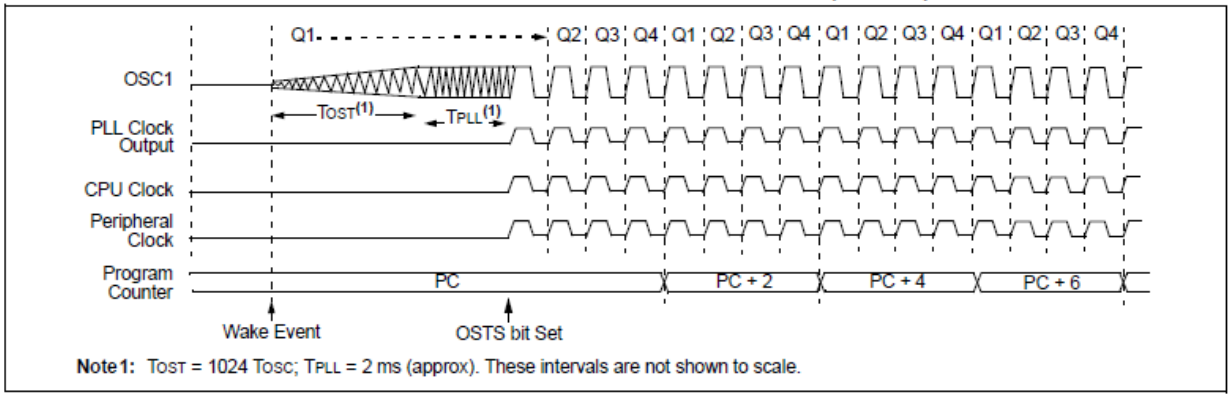



FIGURE 3-6: TRANSITION TIMING FOR WAKE FROM SLEEP (HSPLL)




Note 1: TOST = 1024 TOSC; TPLL = 2 ms (approx). These intervals are not shown to scale.

PURDUE UNIVERSITY

# FIRMWARE DESIGN TECHNIQUES

## Bootloaders

- Depending on your project, it may be desirable to make changes to the code running on your project's programmable devices after the code has shipped (firmware updates, feature unlocks, etc.)
- Bootloader: A piece of code that runs at startup which accepts a program from an external source and writes it to the programmable device's memory
- Allows reprogramming over a number of potential interfaces (USB, Bluetooth, SD Card, Ethernet, etc.)
- Small, open-source bootloaders exist for various microcontroller families, as well as tutorials for creating your own

# Questions?

PURDUE
UNIVERSITY