What is an embedded system?

- Embedded system is a computer based system that is designed for a single purpose or limited set of purposes
- The key difference between an embedded system and a general purpose computer is in the available programs. An embedded system runs software that is provided by the system manufacturer. In a general purpose computer the user can decide the which programs and from which vendor to run
- The application of the embedded system plays a key role in programming
  - Embedded system is (typically) run on its own without an operator
    - Reliability, fault tolerance and fault recovery are important
      - It is not always possible to shutdown or reboot the system at any time
      - Safety (e.g. patient, operator, service personnel, etc.)
  - Interacts with its environment
    - Sensors, actuators, communication interfaces, etc.
    - Can be a part of a larger system

Embedded systems programming

- Software is not (typically) developed on the target processor – target processor is the one that is going to run the application
  - Separate development environment adds more challenge to testing
    - It isn’t always possible to simulate all subsystems but the tests must be run on the target hardware
    - Must consider repeatability and easy addition of test cases
- The resources of an embedded system are often sized to meet the application requirements
  - Aim for low unit cost → constrained resources (RAM, computing power, power consumption, ...)
  - Constraints must be taken into account in development

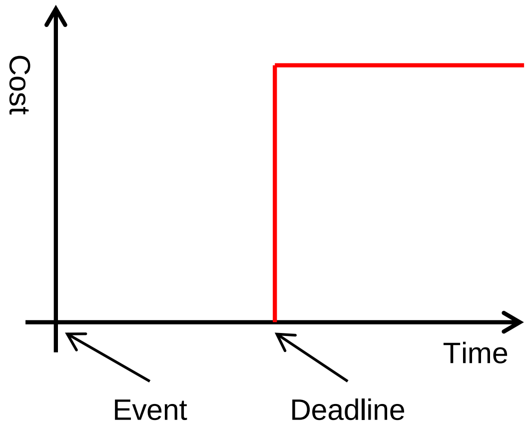
Embedded systems programming

- Embedded systems are ofter real time systems as well
  - Timing constraints can be hard or soft
  - Response time is a very typical constraint
    - Response to an event or message processing time
- Hard realtime
  - Failure to meet the requirements is always a critcal error
- Soft realtime
  - Failure to meet the deadline is undesirable but not necessarily an error condition
  - Typically statistical figures
    - For example average processing time or processing time variance

Real time systems

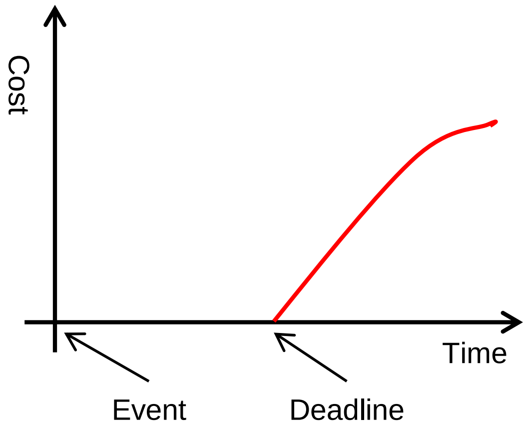
Hard real time

- Failure to meet the deadline causes danger or major financial damage



Soft real time

- Failure to meet the deadline increases cost but is acceptable
- Often related to Quality of Service (QoS)



Timing requirements

- Embedded systems react to events
  - Messages
  - Signals (external signals)
  - Timers
  - Integrated peripherals
    - Communication with other systems (e.g. UART, network card, etc.)
- Events have a hierarchy
  - Layers of protocol stack
- Arrival models (for event)
  - Periodical
    - Some variation may be present in the period
  - Non-periodical
    - For example burst transmission

Timing requirements

- Execution/processing times of events play a key role in evaluating the system response time
    - Hard realtime systems are evaluated using the worst-case execution times
    - Soft real time systems are evaluated using average execution times
        - Variance is typically an important QoS parameter
- Event and actions can occur sequentially or simultaneously
    - Sequential and independent actions and events are ”easy”
    - Codependent simultaneus actions and events require synchronization

Special characteristics of embedded programming

- Pointers to control registers of peripherals and processor hardware
  - Actions are performed or events handled by reading and/or writing processor and peripherals control registers
    - Ordering and timing of read and write is critical
  - Control registers are usually mapped in to the address space and they are read/written the same way as RAM
  - Programmer needs an understanding of the hardware
    - The addresses of control registers can be found in manufacturers documentation
- Volatile modifier
  - Tells the compiler that the value of a variable can change at any time and in a way that compiler can not detect – even when the variable is not being accessed in the program
  - Pointers to memory mapped control registers are declared with volatile modifier

Special characteristics of embedded programming

- Bit manipulation
  - Logical operations and shifts are needed in embedded programming to allow access to individual bits or groups of bits in the control registers
- Lower hardware abstraction level
  - Programmer needs to know how the hardware works
  - Operating systems are becoming more and more common in mid- and high end embedded systems and hide some of the details behind OS API
    - Device driver writer/system programmer still needs to understand the hardware
    - Embedded/real time operating systems tend to be ”lighter” than general purpose computer OSes
  - Interrupts
  - Low level code is often written in C
    - C++ programs can reuse C code or encapsulate it in an object
    - Possibility of using C++ must be taken into account in C code

Special characteristics of embedded programming

- Resource constrains
  - Low end embedded systems have quite limited amount of RAM
  - Frequent use of new/delete can cause problems (or use of malloc/free when programming in C)
  - Hardware/peripherals are physical resources that can't be duplicated → locking is needed in multithreaded/interrupt driven software
- 24/7 unsupervised operation requires special attention in memory management
  - Even a small memory leak will crash the system after a while

Bit manipulation

- IO devices are commonly mapped into memory locations
  - Single memory location contains up to 32 bits that control the IO device
- A bit or group of bit within for example 32 bit word can be manipulated with shift and mask operations
  - Shifting allows us to keep the values in smaller and usually more intuitive range
- For example change bits 13-15 within a 32-bit word without changing the other bits. SFR is the 32-bit word to modify, val is an integer value in the range 0 – 7
  - `SFR = (SFR & 0xFFFF1FFF) | (val << 13);`

  or

  - `SFR = (SFR & ~0xE000) | (val << 13);`

  or

  - ```
    SFR &= ~0xE000;
    SFR |= val << 13;
    ```
    ```

Memory mapped devices and typecasting

```
#define LPC_GPIO_PIN_INT_BASE     0x1C000000
#define LPC_GPIO                  ((LPC_GPIO_T            *) LPC_GPIO_PIN_INT_BASE)

typedef struct {                       /*!< GPIO_PORT Structure */
__IO uint8_t B[128][32];  /*!< Offset 0x0000: Byte pin registers ports 0 to n; pins PIOn_0 to PIOn_31 */
__IO uint32_t W[32][32];  /*!< Offset 0x1000: Word pin registers port 0 to n */
__IO uint32_t DIR[32];    /*!< Offset 0x2000: Direction registers port n */
__IO uint32_t MASK[32];   /*!< Offset 0x2080: Mask register port n */
__IO uint32_t PIN[32];    /*!< Offset 0x2100: Portpin register port n */
__IO uint32_t MPIN[32];   /*!< Offset 0x2180: Masked port register port n */
__IO uint32_t SET[32];    /*!< Offset 0x2200: Write: Set register for port n Read: output bits for port n */
__O  uint32_t CLR[32];    /*!< Offset 0x2280: Clear port n */
__O  uint32_t NOT[32];    /*!< Offset 0x2300: Toggle port n */
} LPC_GPIO_T;

void Chip_GPIO_SetPinDIROutput(LPC_GPIO_T *pGPIO, uint8_t port, uint8_t pin)
{
    pGPIO->DIR[port] |= 1UL << pin;
}
```

The mapping between the members of structure and the memory locations is compiler dependent.

From the user manual

Table 137. Register overview: GPIO port (base address 0x1C00 0000)

Name	Access	Address offset	Description	Reset value	Width	Reference
B0 to B31	R/W	0x0000 to 0x001F	Byte pin registers port 0; pins PIO0_0 to PIO0_31	ext	byte (8 bit)	Table 138
B32 to B63	R/W	0x0020 to 0x003F	Byte pin registers port 1	ext	byte (8 bit)	Table 138
B64 to B75	R/W	0x0040 to 0x004B	Byte pin registers port 2	ext	byte (8 bit)	Table 138
W0 to W31	R/W	0x1000 to 0x107C	Word pin registers port 0	ext	word (32 bit)	Table 139
W32 to W63	R/W	0x1080 to 0x10FC	Word pin registers port 1	ext	word (32 bit)	Table 139
W64 to W75	R/W	0x1100 to 0x112C	Word pin registers port 2	ext	word (32 bit)	Table 139
DIR0	R/W	0x2000	Direction registers port 0	0	word (32 bit)	Table 140
DIR1	R/W	0x2004	Direction registers port 1	0	word (32 bit)	Table 140
DIR2	R/W	0x2008	Direction registers port 2	0	word (32 bit)	Table 140
MASK0	R/W	0x2080	Mask register port 0	0	word (32 bit)	Table 141
MASK1	R/W	0x2084	Mask register port 1	0	word (32 bit)	Table 141
MASK2	R/W	0x2088	Mask register port 2	0	word (32 bit)	Table 141
PIN0	R/W	0x2100	Port pin register port 0	ext	word (32 bit)	Table 142
PIN1	R/W	0x2104	Port pin register port 1	ext	word (32 bit)	Table 142

IO-interfaces

- Simple IO-interfaces are passive and always ready for data transfer (for example buttons, switches, leds, etc.)
    - Read or write can be performed at any time
- Sophisticated IO-interfaces are not always ready for data transfer since they may be busy with ongoing activity (for example still transmitting previous data)
    - Processor needs to be notified when interface is ready for transfer (for example when UART is ready to transmit new byte)
    - Interface needs to be configured before data transfer (for example set UART baud rate or frame size of network adapter)
    - State of interface can read from status register(s).
        - Status registers are mapped into RAM or IO-address space

LPC1549 GPIO pins

- Pins are configurable. Can be configured as
  - Digital pins
    - Output pins
    - Input pins
      - Pullup
      - Pulldown
      - High impedance (no pullup/pulldown)
      - Repeater
  - Analog pins (not available on all pins)
- If chip library is used the pins are configured using the following functions:
  - Chip_IOCON_PinMuxSet
    - Analog/digital
    - Input properties (inverted, pullup, etc.)
  - Chip_GPIO_SetPinDIROutput
  - Chip_GPIO_SetPinDIRInput

GPIO Pin Mux configuration constants

- `void Chip_IOCON_PinMuxSet(LPC_IOCON_T *pIOCON, uint8_t port, uint8_t pin, uint32_t modefunc)`

Value is formed by combining constants that define properties of the pin with OR

Always needed with digital pins

Always use this constant here

```
Chip_IOCON_PinMuxSet(LPC_IOCON, 0, 8, (IOCON_MODE_PULLUP | IOCON_DIGMODE_EN | IOCON_INV_EN));
```

Alternatives:
IOCON_MODE_INACT
IOCON_MODE_PULLDOWN
IOCON_MODE_PULLUP
IOCON_MODE_REPEATER

Use IOCON_MODE_INACT when pin is used for output. For inputs use when an active device (for example a logic chip) is connected to the input.

When present enables inverter on the input. Just omit this if inverter is not needed. Does not affect output!

Select pin direction

- When Pin Mux has been configured then choose the direction of the pin
  - Chip_GPIO_SetPinDIRInput – makes the pin an input
  - Chip_GPIO_SetPinDIROutput – makes the pin an output
- GPIO pins are part of a port
  - A port contains 32 pins – some of the pins may not be available on all variants of the processor.

Always use this constant here

```
Chip_GPIO_SetPinDIRInput(LPC_GPIO, 0, 8);
Chip_GPIO_SetPinDIROutput(LPC_GPIO, 0, 8);
```

Port number – get from board schemactic

Pin number – get from board schemactic

Read input / write output

- When pin is configured as input you can read the value by calling Chip_GPIO_GetPinState

Always use this constant here

  - `Chip_GPIO_GetPinState(LPC_GPIO, port, pin)` returns a boolean value indicating the pin state. If inverter is enabled then the returned value opposite of the actual pin value.

- When pin is configured as an output the value is set with Chip_GPIO_SetPinState

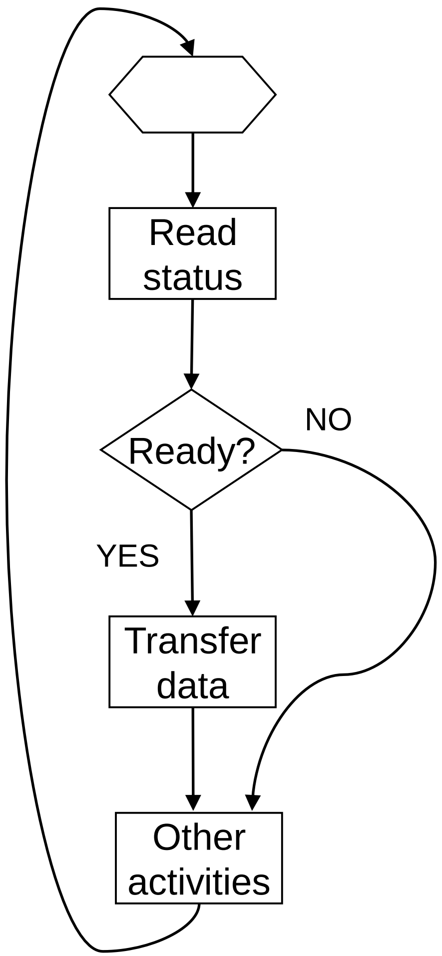
Always use this constant here

  - `Chip_GPIO_SetPinState(LPC_GPIO, port, pin, state)`

Boolean value that is set to output:
true → 1 (3.3V)
false → 0 (0 V)

Programmed IO

- Polling
  - Read interface status
  - If interface is ready then transfer data
  - If interface is not ready continue with other activities
  - Read interface status...
- Other activities can delay interface access (data can be lost or transfer speed is decreased)
- Polling wastes processor resources if device is not ready



Button polling example

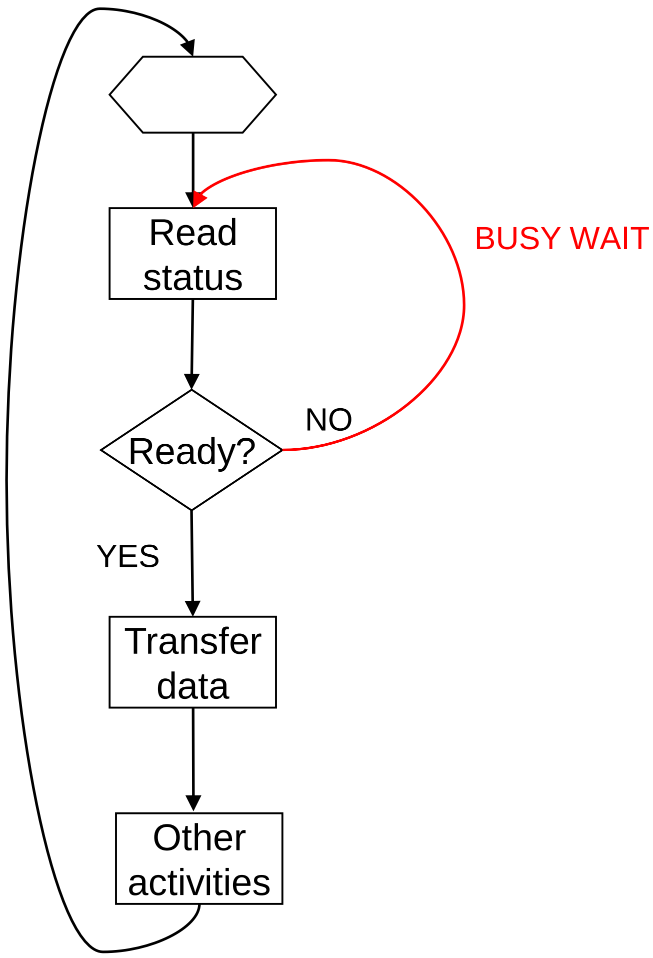
```
while(1) {
    if(Chip_GPIO_GetPinState(LPC_GPIO, 0,8)) {
        dice.SetValue(7);
    }
    if(Chip_GPIO_GetPinState(LPC_GPIO, 1,6)) {
        dice.SetValue(0);
    }
    if(!Chip_GPIO_GetPinState(LPC_GPIO, 1,6)) {
        dice.SetValue(counter);
    }
}
```

Programmed IO

- Busy wait stops other activities and continues only after interface is ready
- Slows down other activities
- Practical if
  - There is only one IO-device

  or
  - Execution may not continue until the device is ready (for example wait until initial configuration is done)

  or
  - Your applications response time is still acceptable



Busy wait loop

```
while(1) {
    if(Chip_GPIO_GetPinState(LPC_GPIO, 1,6)) {
        dice.SetValue(0);
        while(Chip_GPIO_GetPinState(LPC_GPIO, 1,6));
        dice.SetValue(counter);
    }
}
```

Empty statement

- In this example the loop body is an empty statement
    - The loop runs the test over and over again at high speed until the button is released

A little sidetrack to switch bounce

- Buttons typically exhibit switch bounce when the switch is closed
- When a button is pressed it takes a short while for contacts to settle and during that time the contacts may open and close multiple times
- Without filtering the switch bounce may be counted as multiple key presses
- The simplest way of filtering is to wait for typical switch bounce time after a detected key press until the switch state is checked again
    - The switch bounce varies – usually switches from the same batch exhibit similar behavior
    - Can be anything from none to 150 ms – usually in the order of couple of milliseconds (but don't count on that)
- If response time is not a critical issue a busy loop can be used in filtering

Switch bounce on an input with a pull up resistor

Busy loop switch bounce filter

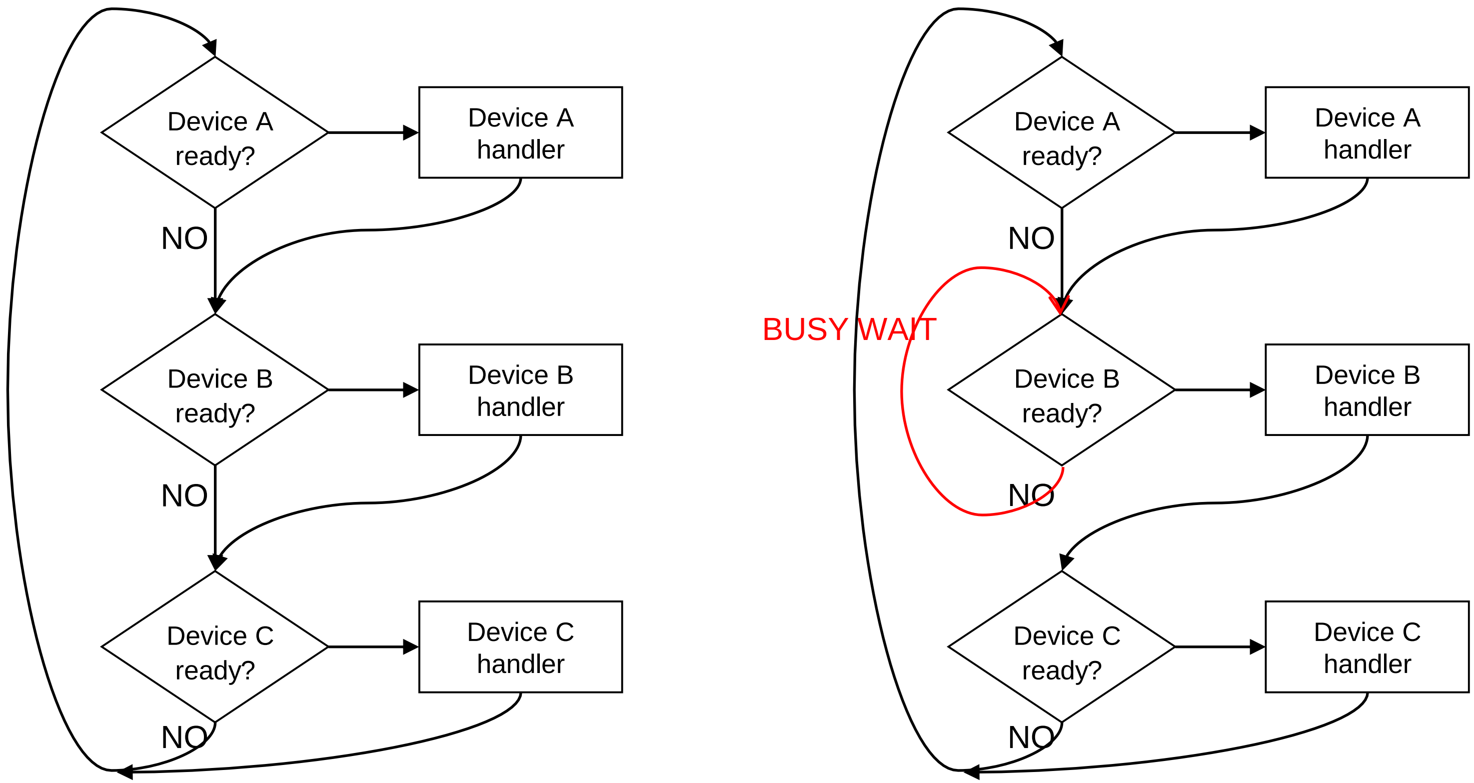
```
bool pressed(void)
{
  int press = 0;
  int release = 0;
  while(press < 3 && release < 3) {
    if(Chip_GPIO_GetPinState(LPC_GPIO, 1,6)) {
      press++;
      release = 0;
    }
    else {
      release++;
      press = 0;
    }
    Sleep(10); // wait 10 ms
  }
  if(press > release) return true;
  else return false;
}
```

This loop runs until we have read the same value three times in a row

Wait a while to allow switch to stabilize

What is the minimum execution time of this function?
What is the maximum execution time?

Programmed IO



If you have more than one device a busy wait stops the polling loop! Even a modest delay in a polling loop can have a huge impact on the responsiveness of the program.

Programmed IO

- Example
  - USB keyboard is attached to a microcontroller that runs at 72 MHz
  - USB HID transfer rate is max 64 kbps (8 kilobytes per second)
  - If polling takes 220 clock cycles and one poll can read one character we spend about 2.5 percent of the CPU time for polling
  - If user types twelve eigth character words per minute (~100 chars per minute) we still need to poll 8192*60 times per minute to ensure that all characters are read.
  - Yet 99,98% of pollings return no data!

$$\frac{8192 \times 220}{72 \times 10^6} \approx 2,5\,\%$$

Note: In a real world a device with this high transfer rate would never be implemented with polling