Daniel **Fitz**

43961229

University of Queensland

CSSE3010 – Embedded System Design

Lecture Summary

Table of Contents

Analog Interfacing

Accuracy, Precision, Resolution

Accuracy

Proximity of measurement results to the true value

Precision

Repeatability or reproducibility of the measurement

Measurement resolution

The smallest change in the underlying physical quantity that produces a response in the measurement

Sampling

Time Quantization

Signal value read/available only in specific times (usually at the same interval). This can cause aliasing – frequency ambiguity of signal components

Amplitude Quantisation

Amplitude of each sample can only take one of a finite number of different values. This adds **quantisation noise**: an irreversible corruption of the signal

Sampling Theorem

Nyquist Theorem

A signal having no spectral components above f_m Hz can be determined uniquely by values sampled at the rate:

$$f_s > 2f_m$$

$f_s > 2f_m$ is called the Nyquist rate

Aperture time

The time during which ADC is continuously converting the varying analog input

$$\text{Max slope} = \frac{\Delta V}{\Delta t} = \omega \times V_{peak} = 2\pi f V_{peak}$$

Example

$$\frac{\Delta V}{\Delta t} = 2\pi f V_{peak} \qquad (f = (\Delta V/\Delta t)(1/2\pi V_{peak}))$$

$$\Delta V = \frac{1}{4}LSB = \frac{1}{4}\left(\frac{10V}{4096}\right) = 0.6mV$$

$$\Delta t = 10us$$

$$f = \left(\frac{\Delta V}{\Delta t}\right)\left(\frac{1}{2\pi V_{peak}}\right) = \left(\frac{0.6mV}{10us}\right)\left(\frac{1}{2\pi 5V}\right) = 2Hz$$

Signal to Noise Ratio (SNR)

- Ratio of the maximum sine wave level to the noise level
- Maximum sine wave has an amplitude of $\pm 2^{n-1}$ which equals an RMS value of:

$$0.71 \times 2^{n-1} = 0.35 \times 2^n$$

- SNR is:

$$20\log_{10}\left(\frac{0.35 \times 2^n}{0.3}\right) = 20\log_{10}(1.2 \times 2^n) = 1.8 + 6n\,dB$$

Sample and Hold

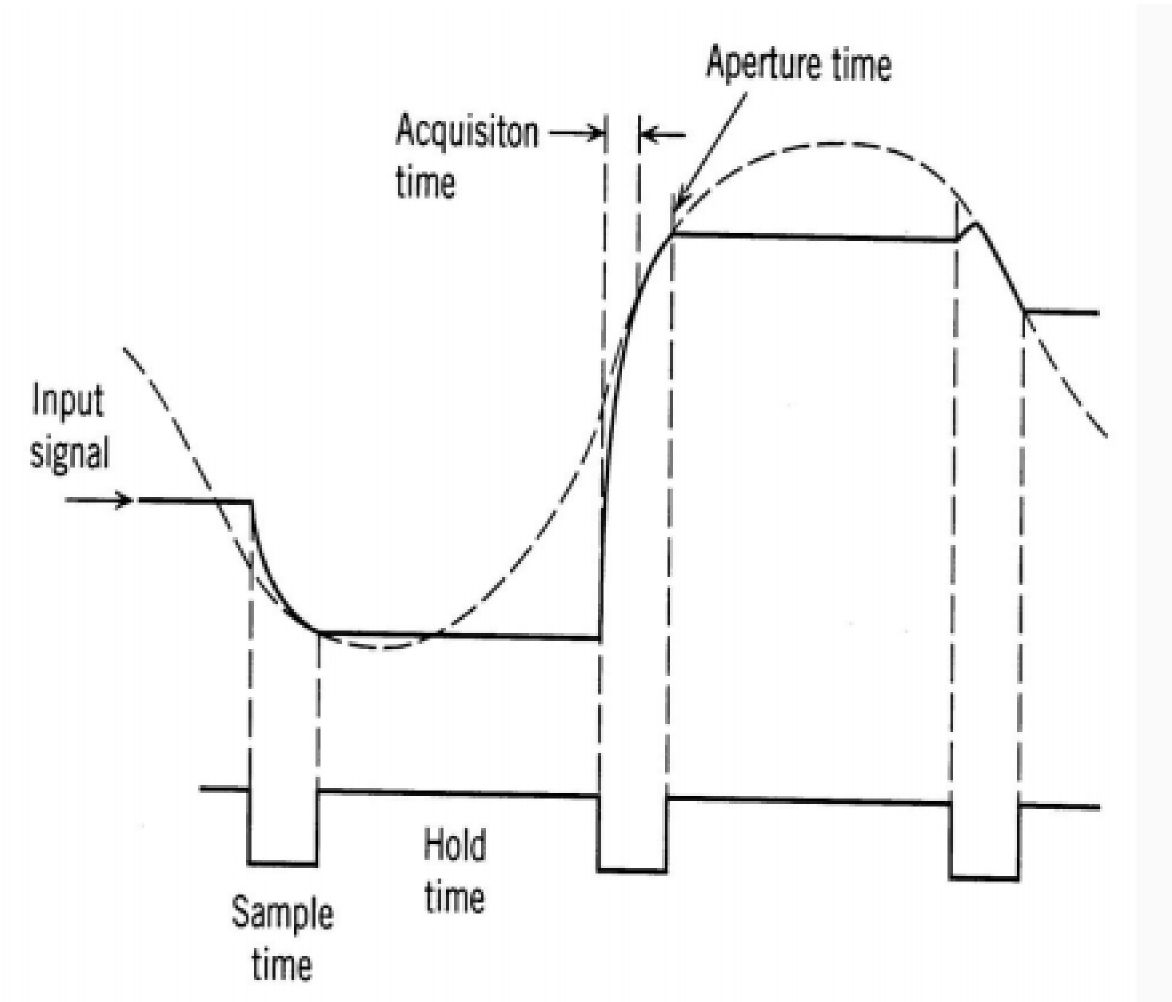

Figure 1: Sample and Hold

Resolution and Dynamic Range

Number of Binary Bits (n)	Full-Scale Decimal Value $(2^n - 1)$	LSB Weight % of Full-Scale Range	LSB Voltage for 1-V Full-Scale Range	Quantization Error Percent of Full-Scale Range	Dynamic Range (From LSB to Full Scale) (dB)
4	15	6.25	60 mV	3.12	24.08
6	63	1.56	16 mV	0.78	36.12
8	255	0.3906	3.9 mV	0.195	48.16
10	1023	0.0977	0.98 mV	0.0488	60.21
12	4095	0.0244	0.24 mV	0.0122	72.25
14	16383	0.00610	61 uV	0.00305	84.29
16	65535	0.00153	15 uV	0.00075	96.33
18	262143	0.000382	4 uV	0.0002	108.37
20	1048575	0.0000954	1 uV	0.00005	120.41

Conclusions

- Interface to analogue world requires thorough understanding and analysis of physical properties this is why it is difficult
- The A/D D/A on-chip converters on microcontrollers are average precision and would require off chip hardware to make conversions more accurate or faster
- Always start interfacing with analysis of the properties and requirements of the analogue side. Digital is always faster

Timing Interfacing

- Use of timing bistate (high or low) waveforms or 'square wave' for interfacing
- A timing waveform can 'mimic' an analog voltage
  - Note Analog voltages an be approximated with specific square waves frequencies and duty cycle
- Commonly used for Pulse Width Modulation, Waveform Frequency or Time Spacing Interfaces
- Timing Interfacing consists of three parameters:
  - Period
  - Frequency
  - Duty Cycle

Waveform Basics

- Period (s) $= T_{high} + T_{low} = T_{period}$
- Frequency (Hz) $= \frac{1}{T_{period}}$
- Duty Cycle (%) $= \frac{100 \times T_{high}}{T_{high} + T_{low}} = 100 \times \frac{T_{high}}{T_{period}}$

Waveform Time Spacing Measurement

- Time spacing of a waveform used to convey information
- Useful for 'irregular' waveforms (high low times are not the same)
  - E.g. time spacing between pulses
- Implemented using Timer Input Capture interrupts
  - A timer counter value is recorded, each time a transition (rising or falling) occurs on the input line ### Frequency Measurement
- The frequency of a waveform can also convey information
- Useful for 'regular' waveforms (High and low times are the same)
- Typically used for optical or mechanical based systems

Example: Wheel Encoder The wheel encoder works by shining light through a pin wheel and detecting the frequency of the light passing through. The frequency of the light passing through is proportional to the speed of the wheel

Waveform Frequency/Period Measurement

- Implemented using Timer Input Capture interrupts
- Can measure using period/frequency by timing transitions.
  Disadvantage: Must rely on accurate timer with enough resolution/precision (e.g. 1ns resolution)
- The number of transitions or zero crossings within a time window, is proportional to the waveform frequency/period.
  Advantage: Does not need high resolution.
  Disadvantage: Only works for regular waveforms

Pulse Width Modulation (PWM)

- Pulse Width Modulation (PWM) uses duty cycle to convey information
- PWM can be used approximate analog (multi-value) waveforms
- Used for controlling mechanical systems such as motors and servo motors

PWM precision/resolution

$$period = N \times \Delta$$

$\Delta =$ resolution
$N =$ PWM precision

Example:

$$Period\,20ms, \Delta = 20us \rightarrow N = 1000 \rightarrow 10bits$$

Timer

Timers features:

- Update interrupts – cause an update interrupt (periodic or not)
- PWM – pulse width modulation (used for controlling servos)
- Timer Input Capture interrupts – cause an interrupt, when a rising or falling edge is detected on an input signal – captures value of timer
- Timer Output Compare – toggle an output pin high or low, when a compare value matches the timer value

ADC

- 3 ADCs: ADC1 (master), ADC2 and ADC3 (slaves)
- Maximum frequency of the ADC analog clock is 36MHz
- 12-bits, 10-bits, 8-bits or 6-bits configurable resolution
- ADC conversion rate with 12 bit resolution is up to:
  - 2.4 M.samples/s in single ADC mode
  - 4.5 M.samples/s in dual interleaved ADC mode
  - 7.2 M.samples/s in triple interleaved ADC mode
- Conversion range: 0 to 3.6 V
- ADC supply requirement: VDDA = 2.4V to 3.6V at full speed and down to 1.65V at lower speed
- 3 ADC1 internal channels connected to:
  - Temperature sensor
  - Internal voltage reference: Vrefint (1.2V typ)
- External trigger option for both regular and injected conversion
- Single and continuous conversion modes
- Scan mode for automatic conversion of channel 0 to channel 'n'
- Left or right data alignment with in-built data coherency
- Channel by channel programmable sampling time
- Discontinuous mode
- Dual/Triple mode (with ADC1 and ADC2 or all 3 ADCs)
- DMA capability
- Analog Watchdog on high and low thresholds
- Interrupt generation on:
  - End of Conversion
  - End of Injected conversion
  - Analog watchdog
  - Overrun

Embedded Design Methodology

Top Down Design

- Embedded System design methodology
  - A complex system is created to meet specific design attributes
- Top down is a process in which a complex design is first organised as a top or high level view
  - The high level overview of the design is divided into sub-components
  - Each sub-componenet is a distinct section of the top level design

            * The sub-components can be further broken down into elements

Valvano

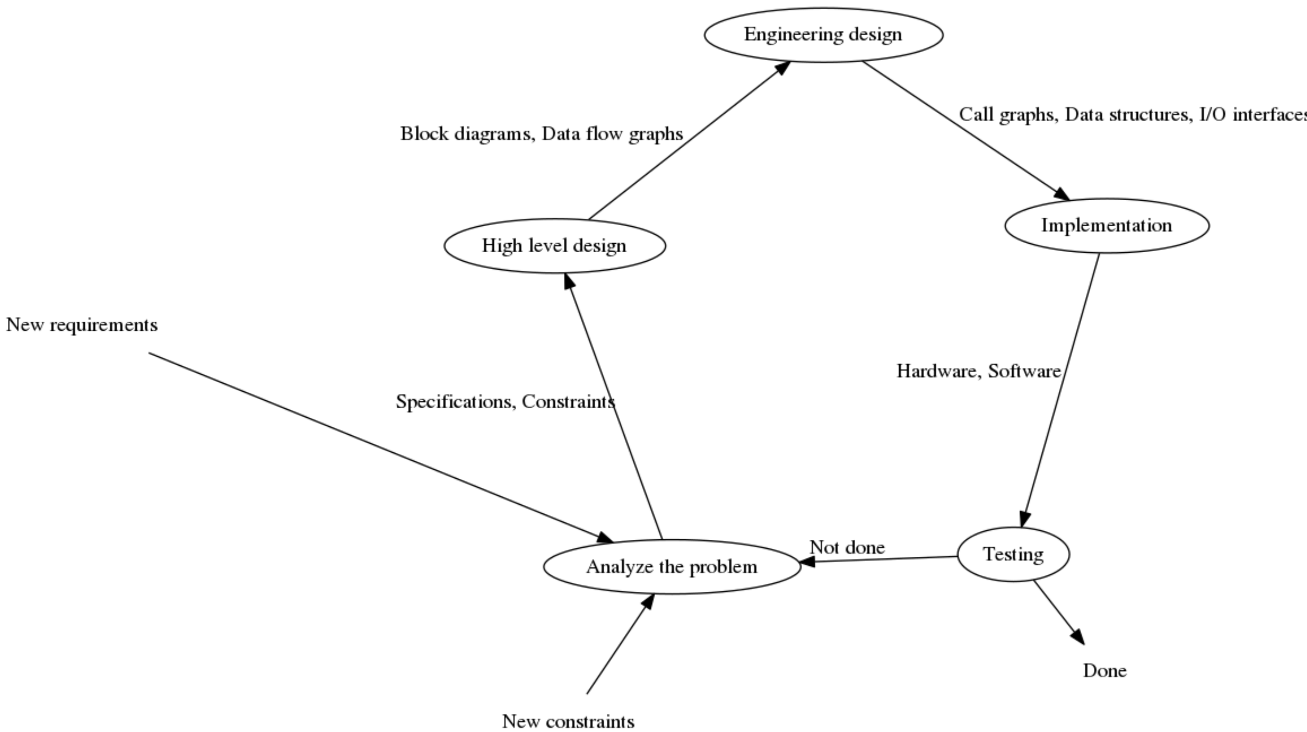

Figure 2: Top Down Valvano

High Level Design Overview

Consists of a number of concepts:

- System Flow:
  - Schematic – shows system inputs and outputs connections
  - Signal/Data Flow diagram
    * Shows the connections of the inputs, all the way to the outputs
      · Shows each stage of connecting the input to the output
- Program Flow:
  - State Diagrams
    * Embedded System Programming main loop can be abstracted as a State Controller
      · A microcontroller program must enter and exit certain states, as it executes
  - Flow Charts
    * Software abstraction of microcontroller program

System Flow – Signal/Data Flow

- Signal/Data Flow diagram
  - Shows the connections of the inputs, all the way to the outputs
- Shows each stage of connecting the input to the output
- Differs to block diagram – is not an overview of the system
- Useful for identifying which software/hardware modules to use
- Useful for debugging and identifying:
  - Break points – where your code will definitely break
  - Weak points – where your code could potentially break
  - Bottle necks – where your system's performance is limited – i.e. 'slow' to respond

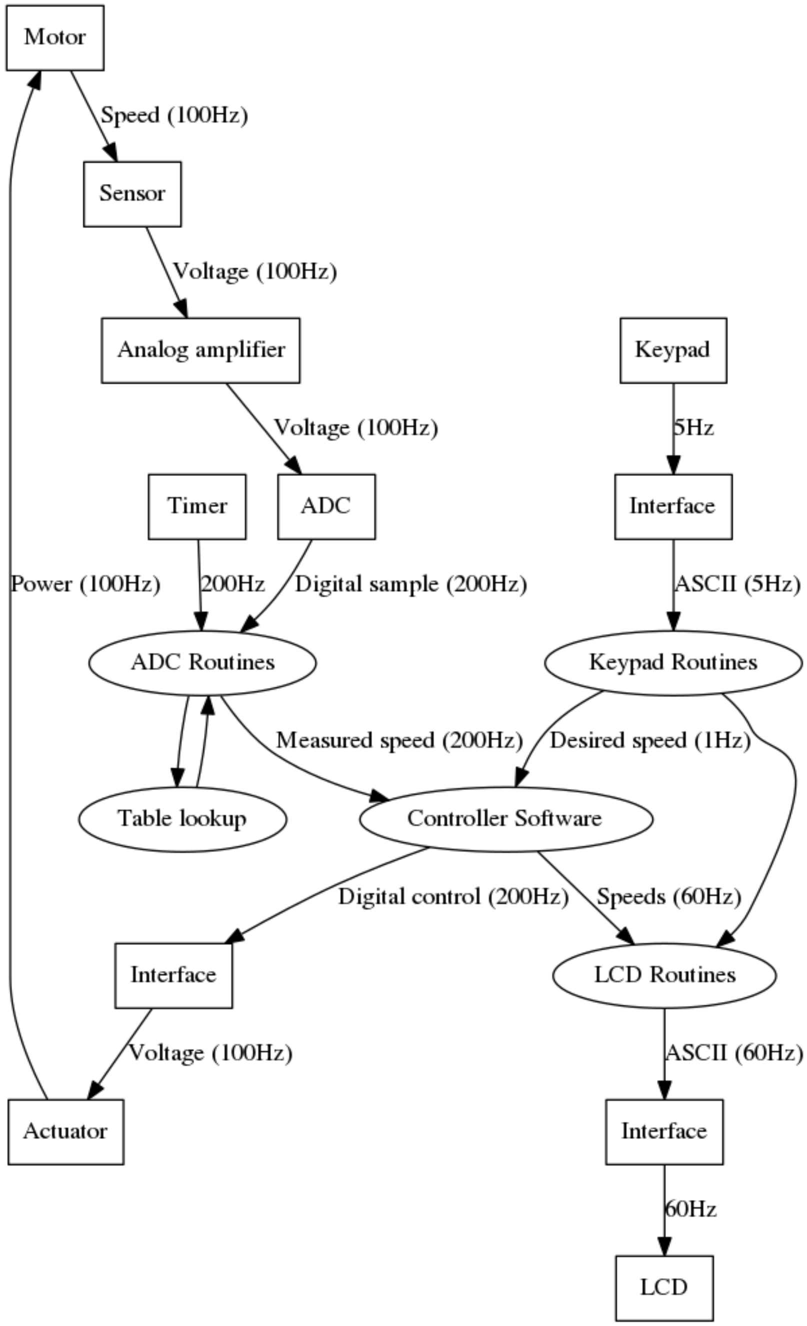

Figure 3: Motor controller Signal/Data Flow diagram

Program Flow

Your program flow should consist of:

- Main loop
- Hardware Initialisation function
- Functions – callable block of code
- Subroutine (not a function but a unit of code)
- Interrupt Service Routines

Program flow is described as:

- State Diagrams
  - main loop

- Flow Charts
  - subroutines

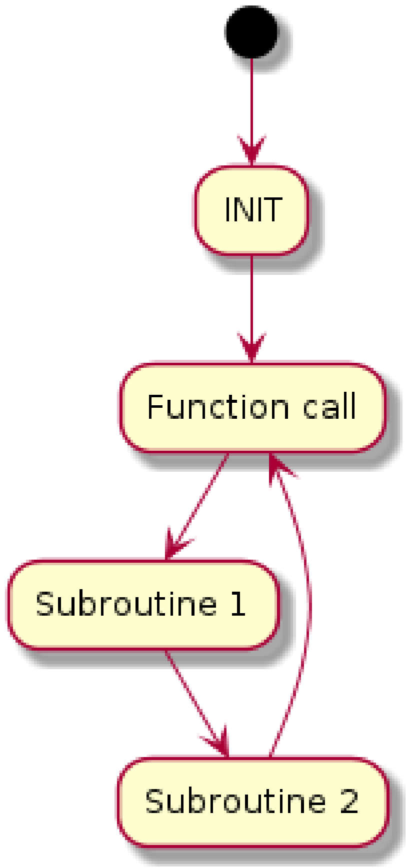

Figure 4: Program Flow – Outline

ISRs have a lightning symbol

Cyclic Executive

- Control loop, using an infinite loop
- Easy to implement
- Has disadvantages
  - unable to prioritising functions/features
  - no realtime control
  - can cause deadlocks, if more than one is used
- Use ONLY one Cyclic Executive to prevent deadlocks

Good Example:

```
while (1) {
    function_a();
    ...
    function_x();
}
```

Bad Example:

```
while (1) {
    while (1) {
        function_a();
```

```
            break;
        }
        ...
        while (1) {
            function_x();
            break;
        }
    }
```

- Appears to support multi-tasking by taking advantage of relatively short processes in a continuous loop:

```
    while (1) {
        function_a();
        ...
        function_x();
    }
```

- Different timing of operations can be achieved by repeating functions in the list:

```
    while (1) {
        function_a();
        function_a();
        function_b();
        function_x();
        function_b();
    }
```

Controller

- For more complex designs – need to implement a controller
- Use Cyclic Executive to implement a controller
- The controller should enter different 'states' of operation
- e.g. initialisation, operating, waiting, etc
- Controllers are typically implemented with Finite State Machines

Basics of Communication

Terminology

Simplex

Communication channel that sends information in one direction only

Half-duplex

Communication in both directions, but only one direction at a time

Full-duplex

Communication in both directions, simultaneously

Serial

One signal path

Parallel

Multiple signal paths

Baseband

Is the signal modulated at (or around) DC, (e.g. Wired transmission)

Bandpass

Or is it modulated onto a higher (carrier) frequency (e.g. Wireless LAN, Radio, TV)

Baseband Modulation

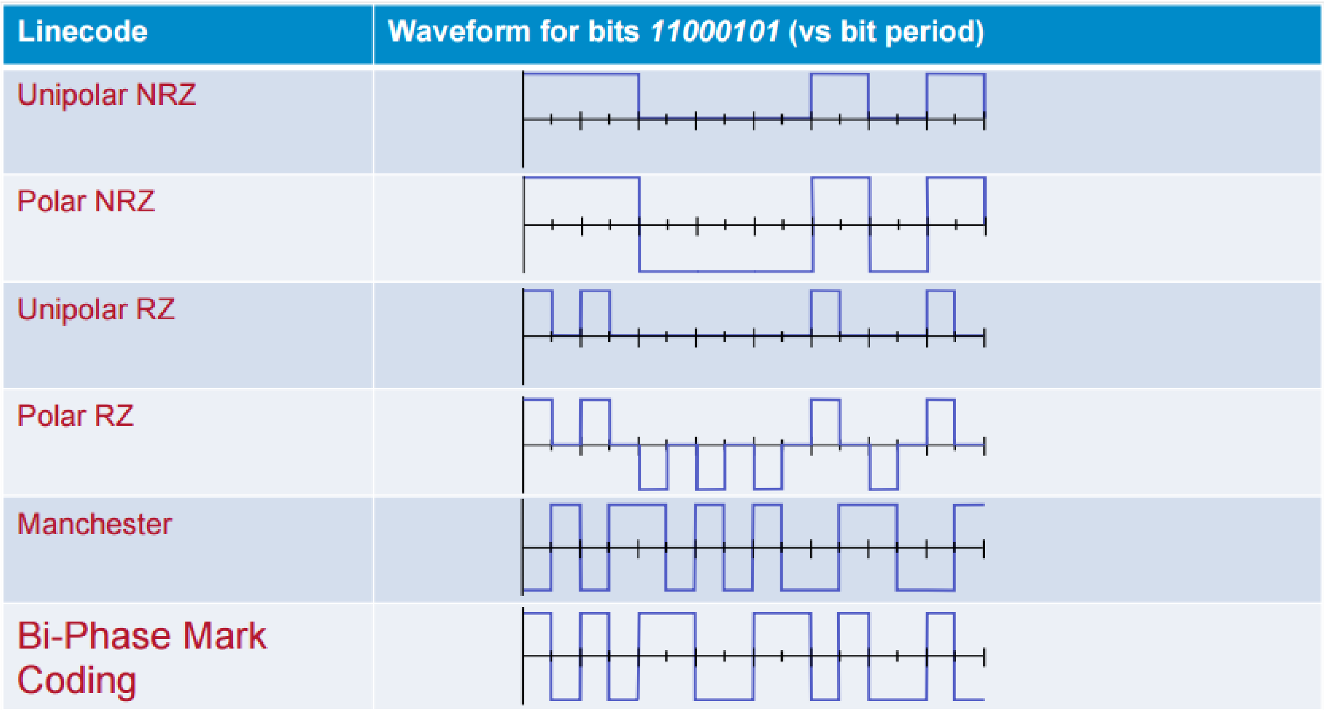

Figure 5: Baseband

Benefits Analysis of Modulation

The previous modulation techniques can be evaluated in terms of:

- Minimal DC component
- BW usage
- Polarity Inversion
- Timing Information
- Frequency Spectrum

Block Coding

- Defined as a (n, m) block code
- 'n' is the number of encoded bits
- 'm' is the number of data bits
- Implemented in different ways
- Here we will use the Generator matrix (G) and Parity Check matrix (H)
  - $y = x\,G$ (encoding data)
  - $s = H\,y^T$ (calculating the syndrome)
  - y is the code word, x is the data, s is the syndrome

Hamming (7, 4) in Matrix form

- Hamming (7,4) Matrix

$$\mathbf{G} = \left[\ \mathbf{I}\ \vdots\ \mathbf{P}\ \right]$$

Generator Matrix

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \left[\begin{array}{cccc:ccc} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}\right] \end{array}$$

$$\mathbf{H} = \left[\ \mathbf{P}^{T}\ \vdots\ \mathbf{I}\ \right]$$

Parity-Check Matrix

$$\left[\begin{array}{cccc|ccc} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{array}\right]$$

Figure 6: Hamming (7, 4) example matrix

Infrared Communications

- Infrared (IR) communications is a short-range form of wireless communications
- IR communications uses the infrared spectrum for transmitting and receiving information
- IR communications is widely as a remote control interface for entertainment and other interfacing applications

Finite State Machine

- Finite State Machine is an abstraction of a controller
- Commonly used design methodology for controllers
- Implemented with either microcontrollers or logic circuitry
  - Our focus on microcontrollers
  - Logic Implementations
- Consists of three sections:
  - Input Processing
  - Next State Processing
  - Output Processing

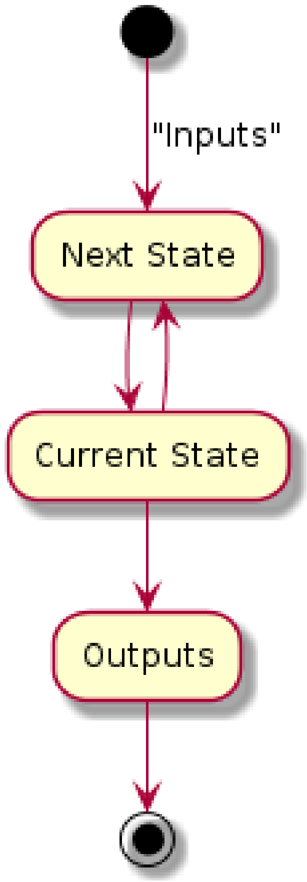

Figure 7: FSM Architecture – Moore Machine

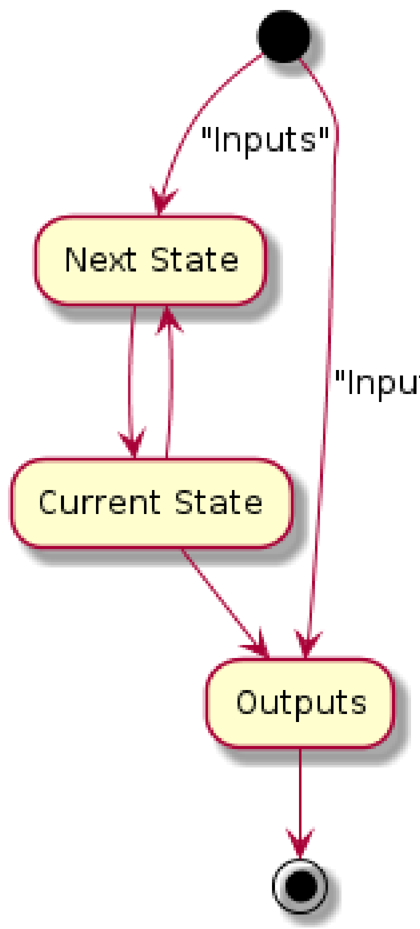

Figure 8: FSM Architecture – Mealy Machine

- Can combine both Mealy and Moore
  - Mealy outputs (depends on input only)
  - Moore outputs (depends on current state only)
- Implemented as cyclic executive

Algorithmic State Machine (ASM)

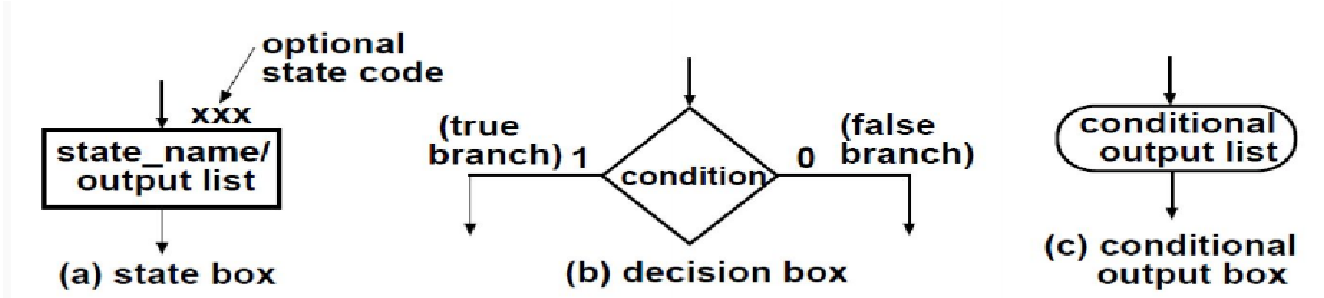

Figure 9: ASM Chart Symbols

- Constructed from ASM Blocks
- Each ASM Block consists of ONE state, together with decision boxes and conditional output boxes
- All the operations in the ASM block happen concurrently when the machine is in the given state
- One entrance, many exits
- A link path is a path from entrance to exit, determined by conditions that are true
- All outputs variables encountered on the active link path are true, all others are false

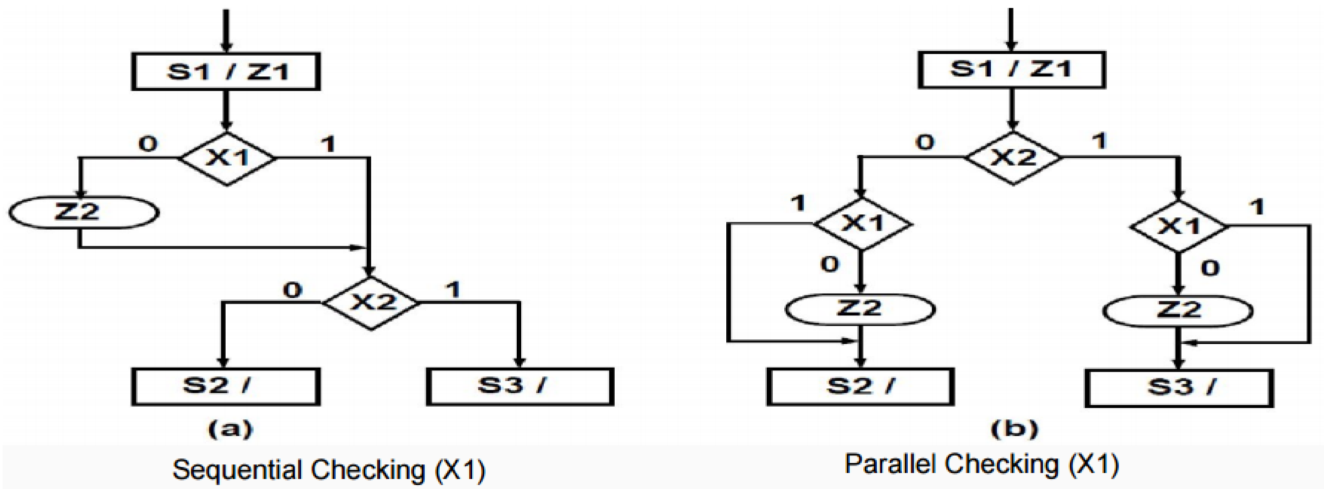

Figure 10: ASM Check conditions in parallel

Parallel Form of ASM

- There may be more than one link path which is true, so that different conditions may be evaluated in parallel
- However, for every unique combination of input variables, they can only have ONE exit path, leading to the next state

Conclusions

- Finite State Machine – FSM defines a sequence of operations that can be implemented in software or hardware
- ASM is a graphical representation of that sequence and easiest to conceptualise
- ASM is a formal specification if it obeys some clear rules
- It can be converted to C code or hardware automatically by appropriate software
- FSM is a very handy concept in defining control sequences and often used in real time embedded systems since it defines very well timing of events and can be checked for correctness by formal methods

Serial Interfacing

- Allow communication between digital devices using a sequential based signalling – e.g. square waves
- Allows for complex and continuous communication, using a few connections (or wires). E.g. to send 8 bits of data – either use 8 wires or a single wire, with 8 low/high transitions
- Variants:
  - Simple Signalling
  - I2C – Inter IC Communications
  - SPI – Serial Peripheral Interface
  - Universal Asynchronous Receive Transmit (UART)

Simple Signalling

- Signalling between digital devices
- Involves signal level transitions:
  - low-high or high-low

- Use to initiate, acknowledge or terminate data transfers
- Also known as 'handshaking'
  - Used for synchronous and asynchronous serial data transfers

I2C

- Only two bus lines are required
- No strict baud rate requirements like for instance with RS232, the master generates a bus clock
- Simple master/slave relationships exist between all components
- Each device connected to the bus is software-addressable by a unique address
- I2C is a true multi-master bus providing arbitration and collision detection
- Relatively slow bus in terms of data throughput

Serial Peripheral Interface (SPI)

- Synchronous Serial Protocol
- Seperate Transmit, Receive and Clock lines
- Select line is used for handshaking between master and slave device

Universal Asynchronous Receive Transmit (UART)

- Serial protocol that is not synchronous (e.g. no share clock signal)
- Relies on:
  - Initial handshake signalling
  - Agreed data transfer rate (baud rate)
  - Oversampling (disadvantage – requires more complex Hardware)
- Separate connections for receive and transmit
- Prone to error at high data rates
- RS232 & RS485 UART protocols are designed for consumer and industrial applications

Conclusions

- I2C bus is a 'proper' serial bus with a protocol for addressing devices and acknowledge signals for both master and slave which are all connected to the same data and clock
- SPI bus implements slave selection through separate enable lines and therefore requires more wires than I2C. From this perspective it is NOT a full-fledged serial bus
- UART bus is an asychronous protocol that requires oversampling (more complex Hardware)

Noise and Synchronisation

Hamming Distance

The number of bits which differ between two words

Cyclic codes

- Easily implemented in hardware
- Represent data bits using a polynomial e.g. message x encodes to $x(p)$, where:
  - $x(p) = x_{n-1}p^{n-1} + \ldots + x_1 p + x_0$
- More concrete example:
  - $x = [1\,0\,1\,1]$ (LSB first)
  - $x(p) = 1p^3 + 1p^2 + 0p + 1 = p^3 + p^2 + 1$
  - NOTE: lowest power always corresponds to LSB
- Cyclic codes are a special type of block code where every cyclic shift of a valid code gives another valid codeward

A cyclic shift, moves bits around from the least significant around.

For LSBit ordered bits: $[1\,0\,1\,1] \rightarrow [1\,1\,0\,1]$

In polynomial form:

- $x(p) = x_{n-1}p^{n-1} + \ldots + x_1 p + x_0$ is shifted to
- $x'(p) = x_{n-2}p^{p-1} + \ldots + x_0 p + x_{n-1} = px(p) + x_{n-1}(p^n + 1)$

Encoding Cyclic Codes

- Codes are encoded using a "generator polynomial" which is a factor of $p^n + 1$ of order $q = n - k$
  - NOTE: n = coded bits, k = msg bits
- Transmitted codewords $x(p)$ are in the form:
  - $x(p) = q_m(p)g(p)$
- Or in terms of the message bits $m(p)$
  - $x(p) = p^q m(p) + c(p)$
- $c(p)$ is the important bit and equals $\frac{p^q m(p)}{g(p)}$

Decoding Cyclic Codes

- Syndrome is calculated by division by $g(p)$ and taking the remainder
- If we take the correctly encoded data $[0\,1\,0 \;\mid\; 1\,1\,0\,0]$ or $p^6 + p^5 + p$ and divide by $g(p)$, we will get no remainder

FreeRTOS

Problems Encountered

Latency unable to guarantee function completion in time
Reliability unable to guarantee function commencement
Priority difficult to manage
Unpredictable difficult to control ISR execution
Limited unable to expand functionality easily
Inflexible fixed settings
Portability too cumbersome to port to other platforms

Advantages

- Technical:
  - Concurrent Execution
  - Multitasking
  - Prioritising of Threads/Processes of Execution (Tasks)
  - Interrupt Handling
  - Synchronisation of variables/functions
- Hardware Abstraction Layers (HAL)
- Feature Scalability
- Platform Independence
- Resource Management
- Simulators/Debugging Tools
- Organisation:
  - Code Style Standard and Organisation
  - Code Modularity
  - Code Reuse
  - Online Community of Users
  - Maintenance and Improvement
  - Licensing – GPL/MIT/Royalty Free

Difference between RTOS and OS

RTOS – Real Time means Right Now	OS
- Designed to run on resource constrained devices	- Designed to run on resource rich devices
- Caters for hardware specific features, i.e. power management	- Designed for interoperability (plug and play)
- Autonomous operation	- Designed for advanced user interfaces
- Industrial control applications	
- Ease of hardware peripheral interfacing	

RTOS – Real Time means Right Now	OS
- Extensive configurability options	

Features

- A RTOS provides:
  - Concurrent Tasks (Thread/Process of Execution)
  - Multitasking
  - Data/Parameter (Sharing Queues)
  - Prioritisation
  - Synchronisation (Semaphores)
  - Hardware Abstraction Layers (HAL)
  - Time Management
  - Resource Management
- Advanced Features
  - Embedded Networking Stack
  - File System

Kernel

Processing Core of the RTOS:

- Handles:
  - Concurrent and multitask execution
  - Interrupt Requests
  - Resource and Time Management
- Contains:
  - Realtime Scheduler
  - Context Switching
  - Resource Manager
  - Hardware Interface

Scheduler

Determines when a task or interrupt request can execute

Multitasking Allows more than one task to execute

Concurrent Execution Allows more than one task to execute at the same time

Many types of Task Schedulers:

- Priority pre-emptive
  - Allows a task to be interrupted when executing
  - Uses priority of tasks to determine schedule
- First in first out
  - First task in the 'queue' executes
- Shortest Time Remaining
  - Execute the task with the smallest running time
- Round-robin
  - Execute each task in no sequence

Operation:

- Suspend Kernel
- Suspend and Swap out task
- Resume and Swap in a task
- Determine if a task is using a hardware resource
  - Take appropriate blocking/locking action
- Execute task

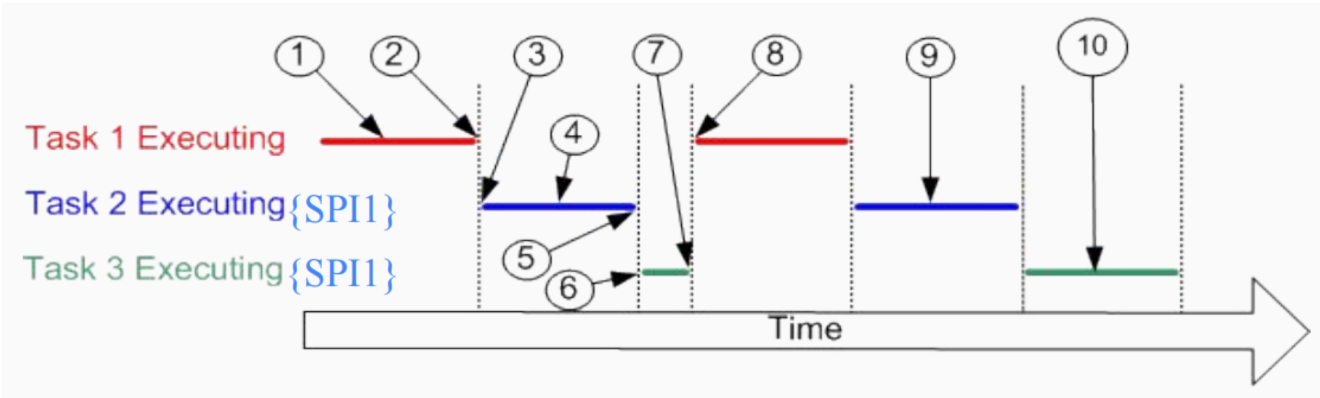

Figure 11: Example of a Typical Task Scheduling

1) task 1 is executing
2) the kernel suspends (swaps out) task 1
3) resumes task 2
4) while task 2 is executing, it locks a processor peripheral (e.g. SPI1) for its own exclusive access
5) the kernel suspends task 2
6) resumes task 3
7) task 3 tries to access the same processor peripheral (e.g. SPI1), finding it locked task 3 cannot continue so suspends itself
8) the kernel resumes task 1
9) the next time task 2 is executing it finishes with the processor peripheral and unlocks it
10) the next time task 3 is executing it finds it can now access the processor peripheral and this time executes until suspended by the kernel

Context Switching

The process of saving the context of a task being suspended and restoring the context of a task being resumed

- Executing task uses resources:
  - Process / microcontroller registers
    - Instruction register
    - Stack pointer register
  - Memory Access
- Resources used from the executing task's "context" or state
- A task does not know when it is going to get suspended (swapped out or switched in) or resumed (swapped in or switched in) by the kernel
- To prevent errors, upon resumption a task has a "context" identical to that immediately prior to its suspension
- The kernel saves the context of a task when suspended
- Upon resumption – task's saved context is restored by the kernel prior to its execution

Resource Management

Manage memory and other hardware peripheral resources

- Embedded platforms are usually 'resource poor'
- The standard C library *malloc()* and *free()* functions can sometimes be used but:
  - Not always available on embedded systems
  - Take up valuable code space
  - Not thread safe
  - Not deterministic (the amount of time taken to execute that function will differ from call to call)

Resource Management

Different memory allocation algorithms:

- Use thread-safe *malloc()* and *free()* functions

- Best Fit
  - Allocate the smallest block memory
- Worst Fit
  - Allocate the largest block memory
- First Fit
  - Find the first block of memory that fits
- Next Fit
  - Variant of First Fit. Find the next block of memory that fits

Timing

- The FreeRTOS real time kernel measures time using a tick count variable
- A timer interrupt increments the tick count with strict temporal accuracy
- Allowing the real time kernel to measure time to a resolution of the chosen timer interrupt frequency
- Each time the tick count is incremented the real time kernel must check to see if it is now time to unblock or wake a task
- A task may have woken or be unblocked during the tick ISR will have a priority higher than that of the interrupted task
  - The tick ISR should return to the newly woken/unblocked task – effectively interrupting one task

Task

- A real time application can be structured as a set of tasks
- Only one task can be executing at any point in time
- The scheduler may start and stop each task (swap each task in and out) as the application executes
- Each task has its own memory stack (TCB – Task Control Block)
- When the task is swapped out the execution context is saved to the stack of that task so it can also be exactly restored when the same task is later swapped back in
- Tasks are assigned a priority level, used by the scheduler to determine which task to execute or suspend

Task Control Block (TCB)

Block of Memory used by task

- Used to save the Task's:
  - Local variables
  - Current values of processor registers
  - Current Task state
- Used for context switching:
  - Suspend or resume a task
  - Ensures that no 'glitches' occur after a context switch
  - Allows a task to be suspended or resumed without knowing

Task States

A task can exist in one of the following states:

- Running:
  - When a task is actually executing it is said to be in the Running state and is using the CPU
- Ready:
  - Ready tasks are those that are able to execute but not currently executing because a different task of equal or higher priority is already in the Running state
- Blocked:
  - Task is waiting for an event
    - Temporal event: a task calls *vTaskDelay()* it will block (be placed into the Blocked state) until the delay period has expired
    - External event: Tasks can also block waiting for queue and semaphore events
  - Tasks in the Blocked state always have a timeout, after which the task will be unblocked
  - Blocked tasks are not available for scheduling
- Suspended:
  - Tasks will only enter or exit the suspended state using: *vTaskSuspend()* and *vTaskResume()*

- A ‘timeout’ period cannot be specified
- A suspended task cannot be scheduled

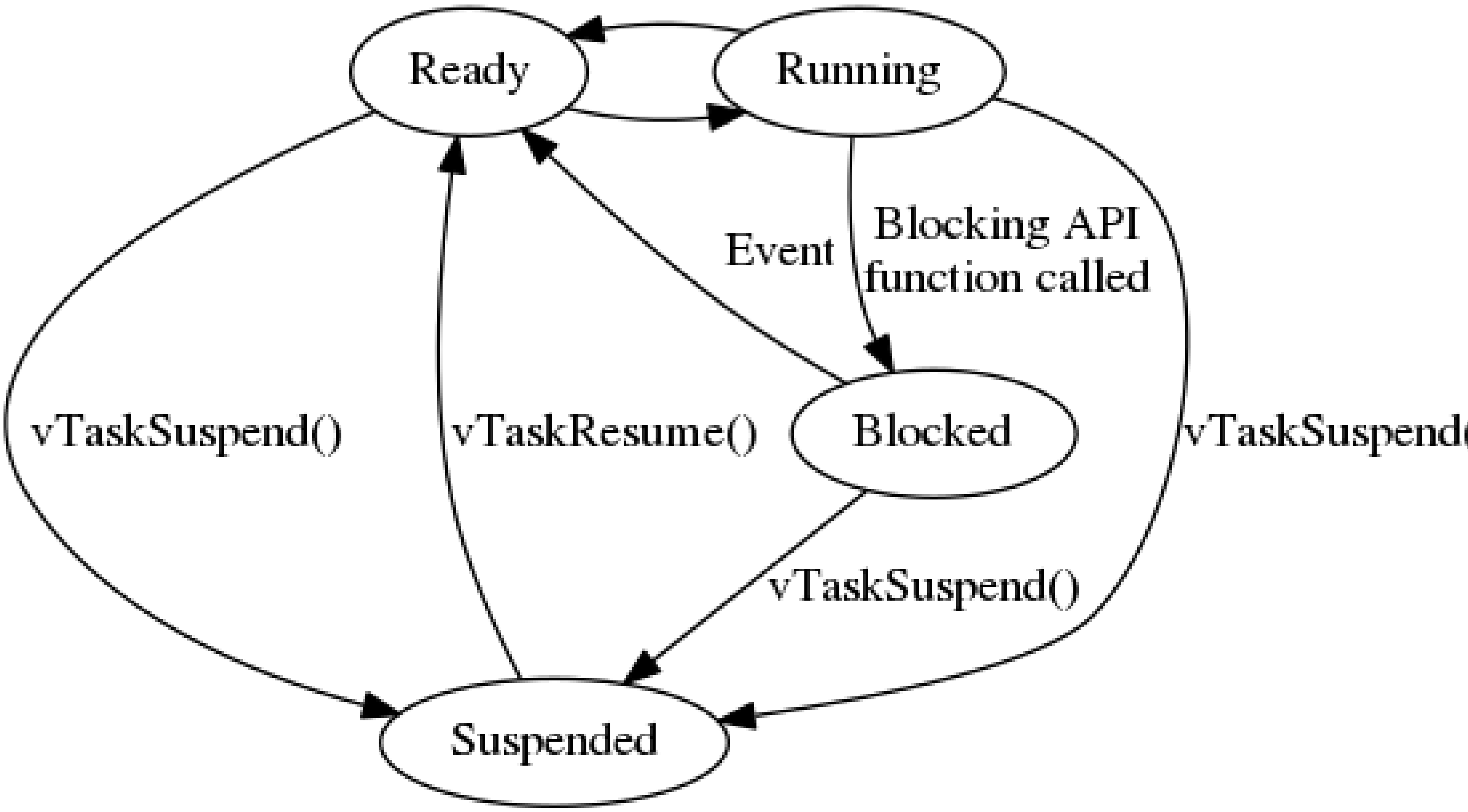

Figure 12: RTOS Task State Diagram

Task Priorities

- Each task is assigned a priority from 0 to *configMAX_PRIORITIES - 1* (*FreeRTOSConfig.h*)
- Low priority numbers denote low priority tasks
- The idle task has priority zero (*tskIDLE_PRIORITY*)
- Task in the Running state will always be the highest priority task that is able to run
- Ready state tasks of equal priority are scheduled using a time sliced round robin scheduling scheme

Idle Task

- The idle task is created automatically when the RTOS scheduler is started
- Ensures there is always at least one task that can run
- Lowest priority to ensure it does not use any CPU time if other higher priority application tasks in the ready state
- Used for freeing memory allocated to tasks that have been deleted
- The idle task must have other active functions that can be starved of CPU time under all conditions
- Can enable/disable the idle task in *FreeRTOSConfig.h*

Task Function Calls

Function Name	Description
xTaskCreate	Creates a task
vTaskDelete	Deletes a task
vTaskDelay	Delays a task for a certain period
vTaskDelayUntil	Variation of vTaskDelay

Function Name	Description
vTaskPrioritySet	Sets the priority level of a task
uxTaskPriorityGet	Gets the priority level of a task

Co-Routine

A Co-Routine is similar to tasks but differs with:

- Stack usage:
  - All the co-routines within an application share a single stack. This is to reduce the amount of RAM required compared to using tasks
- Scheduling and priorities:
  - Use prioritised cooperative scheduling with respect to other co-routines
  - Scheduler must be called repeatedly
- Sharing a stack between co-routines results in much lower RAM usage
- Reduces problems with reentrancy
- Portable across architectures
- Fully prioritised relative to other co-routines, but can always be preempted by tasks
- Restrictions on where API calls can be made
- Co-operative operation only amongst co-routines

States

- Running:
  - A co-routine that is utilising the CPU
- Ready:
  - Co-routines are those that are able to execute (not blocked) but are not currently executing
  - A co-routine may be in the Ready state because:
    * Another co-routine of equal or higher priority is already in the Running state
    * A task is in the Running state
- Blocked:
  - A co-routine is in the Blocked state if it is currently waiting for either a temporal or external event

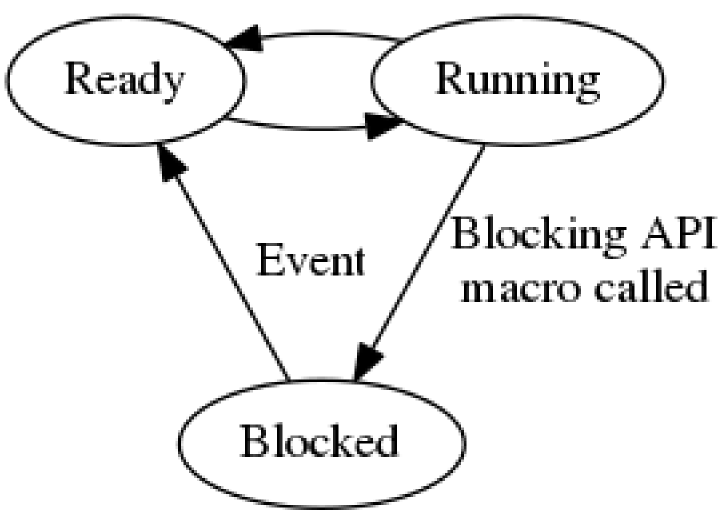

Figure 13: RTOS Co-Routine State Machine

Scheduling

Co-Routines are scheduled by repeated calls to *vCoRoutineSchedule()*. Idle task is used to call *vCoRoutineSchedule()*

Semaphore

- Semaphores are used for synchronisation and mutual exclusion
- Semaphores permit a block time to be specified
  - Block time indicates the maximum number of 'ticks' that a task should enter the Blocked state
- If more than one task blocks on the same semaphore then the task with the highest priority will be the first to be unblocked
- Typically used for synchronising task and interrupts (i.e. a task will block on a semaphore until signaled by an interrupt)
- Three types:
  - Binary Semaphore:
    * Blocking wait and unblocks when signalled
  - Counting Semaphore:
    * Returns a counter value when signalled
  - Mutex:
    * Used to provide mutual exclusion

Binary/Mutex

- Binary semaphores are used for both mutual exclusion and syncronisation purposes
- Similar to mutexes but key differences are:
  - Mutexes include a priority inheritance mechanism, binary semaphores do not
  - Binary semaphores are used mainly for synchronisation (between tasks or between tasks and an interrupt)
  - Mutexes are used for simple mutual exclusion

Function Calls

Function Name	Description
xSemaphoreCreateBinary	Creates a binary semaphore
xSemaphoreCreateCounting	Create a counting semaphore
xSemaphoreCreateMutex	Create a mutex semaphore
xSemaphoreGiveFromISR	Single a semaphore from an interrupt service routine
xSemaphoreTake	Block on a semaphore in a task

Queue

- Used for inter-task communication
- Used to send messages between tasks and between interrupts and tasks (i.e. FIFOs buffers)
- Considered to be thread safe FIFO buffers
- Queues can contain 'items' of fixed size
- Queue items can be added to the front or back of a queue
- Items are placed into a queue by copy, not by reference
  - Need to keep the size of each item placed into the queue to a minimum
- Multiple tasks cannot access the data simultaneously
- A queue does ensure mutual exclusion
- Queue API permit a block time to be specified
- Should more than one task block on the same queue then the task with the highest priority will be the task that is unblocked first

Function Calls

Function Name	Description
xQueueCreate	Creates a queue
xQueueSendToBack	Send an item to the back of the queue
xQueueSendToFront	Send an item to the front of the queue
xQueueReceive	Receive and remove items from a Queue (Blocks)

Function Name	Description
xQueuePeek	Receives but does not remove items from a queue
uxQueueMessagesWaiting	Returns the number of items are in the queue

Software Timer

- A software timer allows a function to be executed after a set time
- The function to be executed by the software timer is called using the software timer's callback function
- The time between a software timer being started, and its callback function being executed, is called the period
- The software timer's callback function is executed when the software timer's period expires

Funciton Calls

Funciton Name	Description
xTimerCreate	Creates a software time
xTimerIsActiveTimer	Checks if a software timer is active
pvTimerGetTimerID	Return ID of software timer
xTimerStart	Start Timer
xTimerStop	Stop Timer
xTimerReset	Reset Timer

Memory Management

- Memory (RAM) is used each time a task, queue, mutex, software timer or semaphore is created
- FreeRTOS has four sample memory allocation implementations:
  - Heap 1: Does not allowing freeing of memory
  - Heap 2: Best Fit algorithm and allows memory to be freed
  - Heap 3: Implements Malloc and Free functions
  - Heap 4/5: First Fit algorithm and allows memory to be freed

Heap 1

- Does not permit memory to be freed once it has been allocated
- Can be used if application never deletes a task, queue, semaphore, mutex, etc.
- Is deterministic (always takes the same amount of time to execute)
- Implemented by subdividing a single array into smaller blocks as RAM is requested

Heap 2

- Uses the best fit memory allocation algorithm
- Allows previously allocated blocks to be freed
- Does not combine adjacent free blocks into a single large block
- Can be used even when an application repeatedly deletes tasks, queues, semaphores, mutexes, etc
- Is not deterministic
- Is suitable for most applications that have to dynamically create tasks
- Should not be used if the memory (task stack and queue sizes) being allocated and freed is of a random size

Heap 3

- Implements a wrapper for the standard *malloc()* and *free()* functions
- The wrapper simply makes the *malloc()* and *free()* functions thread safe
- Requires the linker to setup a heap, and the compiler library to provide *malloc()* and *free()* implementations
- Is not deterministic
- Can be used if the memory (task stack and queue sizes) being allocated and freed is of a random size

Heap 4/5

- Uses the First Fit memory allocation algorithm
- Does combine adjacent free memory blocks into a single large block

- Include a coalescence algorithm
- Can be used even when the application repeatedly deletes tasks, queues, semaphores, mutexes, etc
- Less likely to result in a heap space that is badly fragmented into multiple small blocks – even when the memory being allocated and freed is of random size
- Is not deterministic

Function Calls

Function Name	Description
configTOTAL_HEAP_SIZE	Sets total amount of memory available
xPortGetFreeHeapSize	Returns the amount of memory unallocated
pvPortMalloc	Wrapper for *malloc()* (Heap 3 and 4)
pvPortFree	Wrapper for *free()* (Heap 3 and 4)

Kernel Control

Macro Name	Description
taskYIELD	Force a context switch
taskENTER_CRITICAL	Mark the start of a critical code region. Context switches cannot occur when in a critical region
taskEXIT_CRITICAL	Mark the end of a critical code region
taskENABLE_INTERRUPTS	Enable interrupts
taskDISABLE_INTERRUPTS	Disable interrupts
portSAVE_CONTEXT	Save the context of a task
portRESTORE_CONTEXT	Restore the context
vTaskStartScheduler	Start the scheduler. Must be called from the main function
vTaskEndScheduler	Stops the scheduler
vTaskSuspendAll	Suspends all kernel activity but allows interrupts to occur
vTaskResumeAll	Resumes all kernel activity

Summary

- RTOS Fundamentals
  - RTOS Features: Tasks, Synchronisation, Sharing, Priorities, HAL, etc.
- RTOS Kernel
  - Core of an RTOS
  - Performs Scheduling, Context Switching, Resource Management
- Scheduling
  - Determines when a task can execute
- Context Switching
  - Allows tasks to be suspended and resumed without errors
- Resource Management
  - Memory allocation and de-allocation for RTOS elements
- FreeRTOS Timing depends on a single timer interrupt to generate a timing tick
- Threads of execution (Tasks):
  - Small, schedulable and sequential programmable units
  - Concurrent operation and multitasking
- Synchronisation (Semaphores):
  - Allows tasks to be blocked or suspended until signaled
  - Provides mutual exclusion
- Parameter Sharing (Queues):

  - Used to share variables between tasks
  - Allows synchronised reading/writing of items by multiple tasks
    * i.e. Prevents multiple write errors
- Task:
  - Used by as a concurrent, sequential programming units
- Co-routine:
  - Similar to a task but shares a stack amongst the co-routines
- Semaphore:
  - Provides synchronisation and mutual exclusion
  - Types: Binary, Counting, and Mutex
- Queue:
  - Used to provide message passing between tasks
- Software timer:
  - Causes a function to execute after a period of time
- Memory Management:
  - Heap 1: Does not allow freeing of memory
  - Heap 2: Best Fit algorithm and allows memory to be freed
  - Heap 3: Implements Malloc and Free functions
  - Heap 4: First Fit algorithm and allows memory to be freed
- Kernel Management:
  - Various macros and functions to suspend and restart kernel activity