EGH456 Embedded Systems

Lectures 8

Thread Intercommunication: Messages, Events and Mailboxes

Dr Chris Lehnert

Contents

- Message Queues
- Mailboxes
- Events
- Clock Module

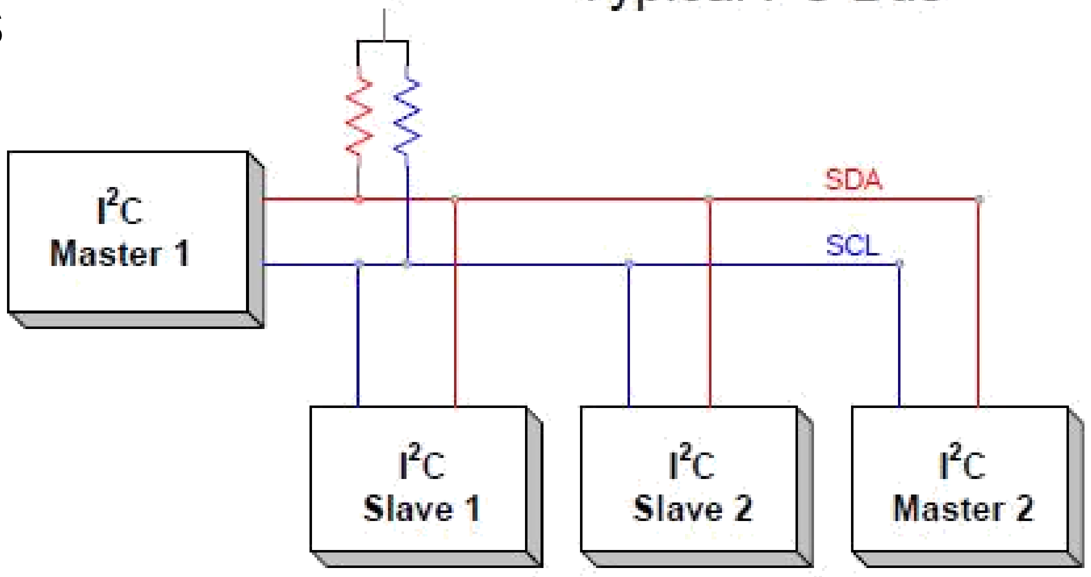
NOTE: SYS/BIOS User Guide (not the TI-RTOS) has the relevant real-time ‘kernel’ information. BIOS Getting Started Guide for information on projects.

Inter-Integrated Circuit (I2C)

- Multi-master, multi-slave
- Half duplex, synchronous transmission
- Low speed short distance
- Two wires like UART but supports multiple slave devices
  - Clock signal (SCL)
  - Data signal (SDL)
- Open drain
  - Slave can pull low

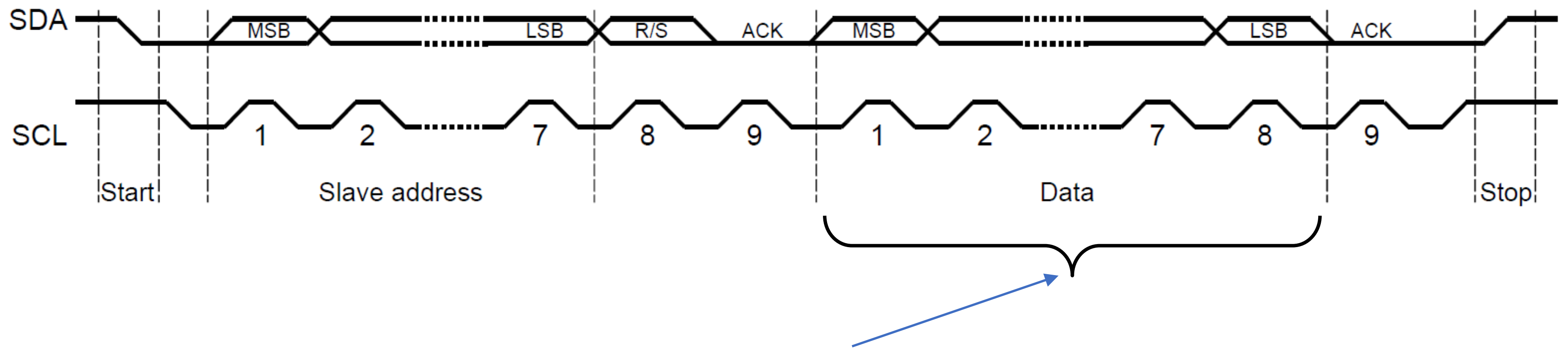
  but can't drive high

Typical I^{2}C Bus



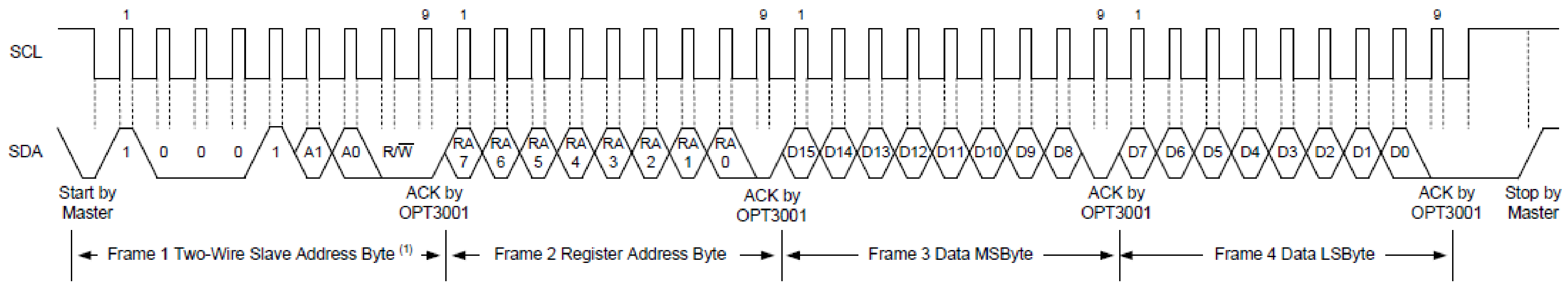
Inter-Integrated Circuit (I2C) - Protocol

- Specific protocol to adhere to, handled by device
  - Start/stop condition, address frame (7 bits + R/W), data frame
  - ACK/NACK – ‘1’ represents slave did not respond



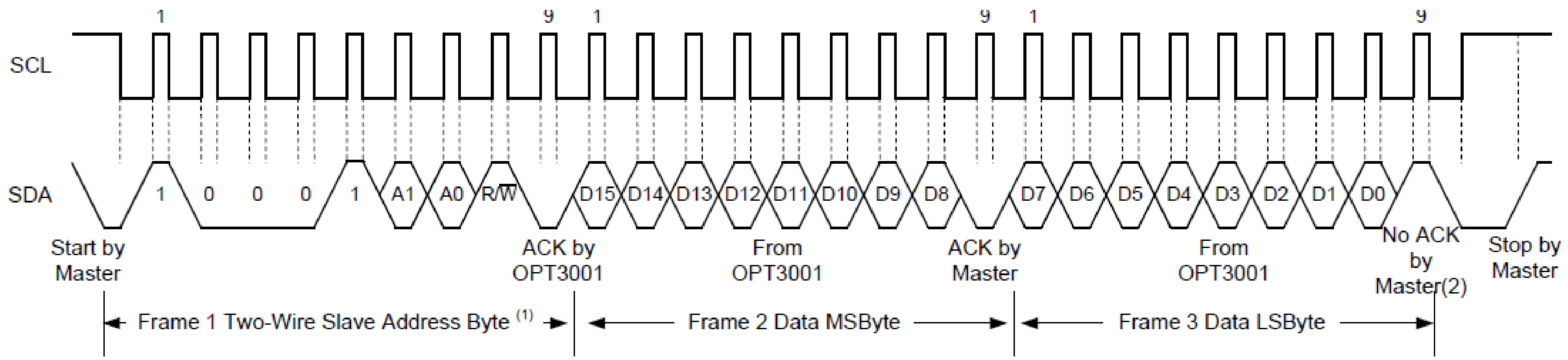
Usually Register Address first then followed by data to write or data being read

OPT3001 Register Read/Write Example



(1) The value of the slave address byte is determined by the setting of the ADDR pin; see Table 1.

Figure 20. I²C Write Example



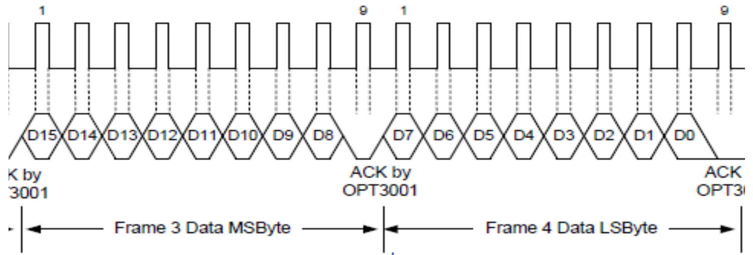
(1) The value of the slave address byte is determined by the ADDR pin setting; see Table 1.

(2) An ACK by the master can also be sent.

Figure 21. I²C Read Example

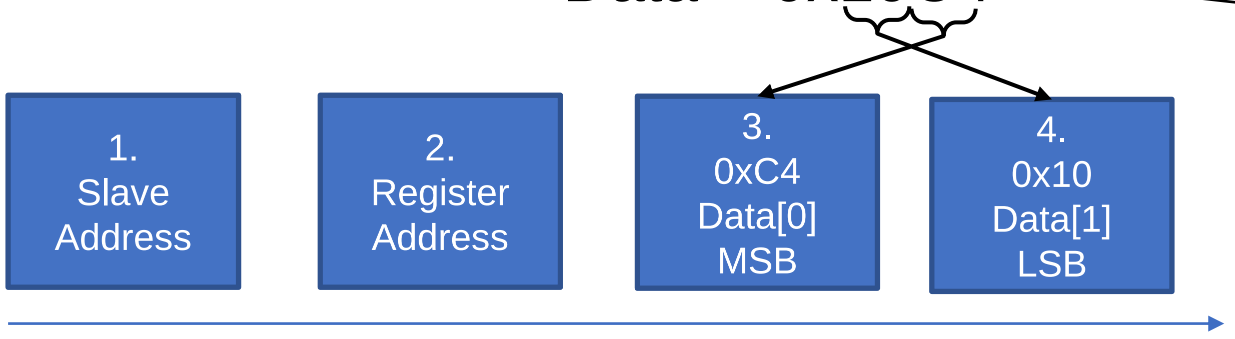
16 bit Register Write Example

Configuration Register Address 0x1h



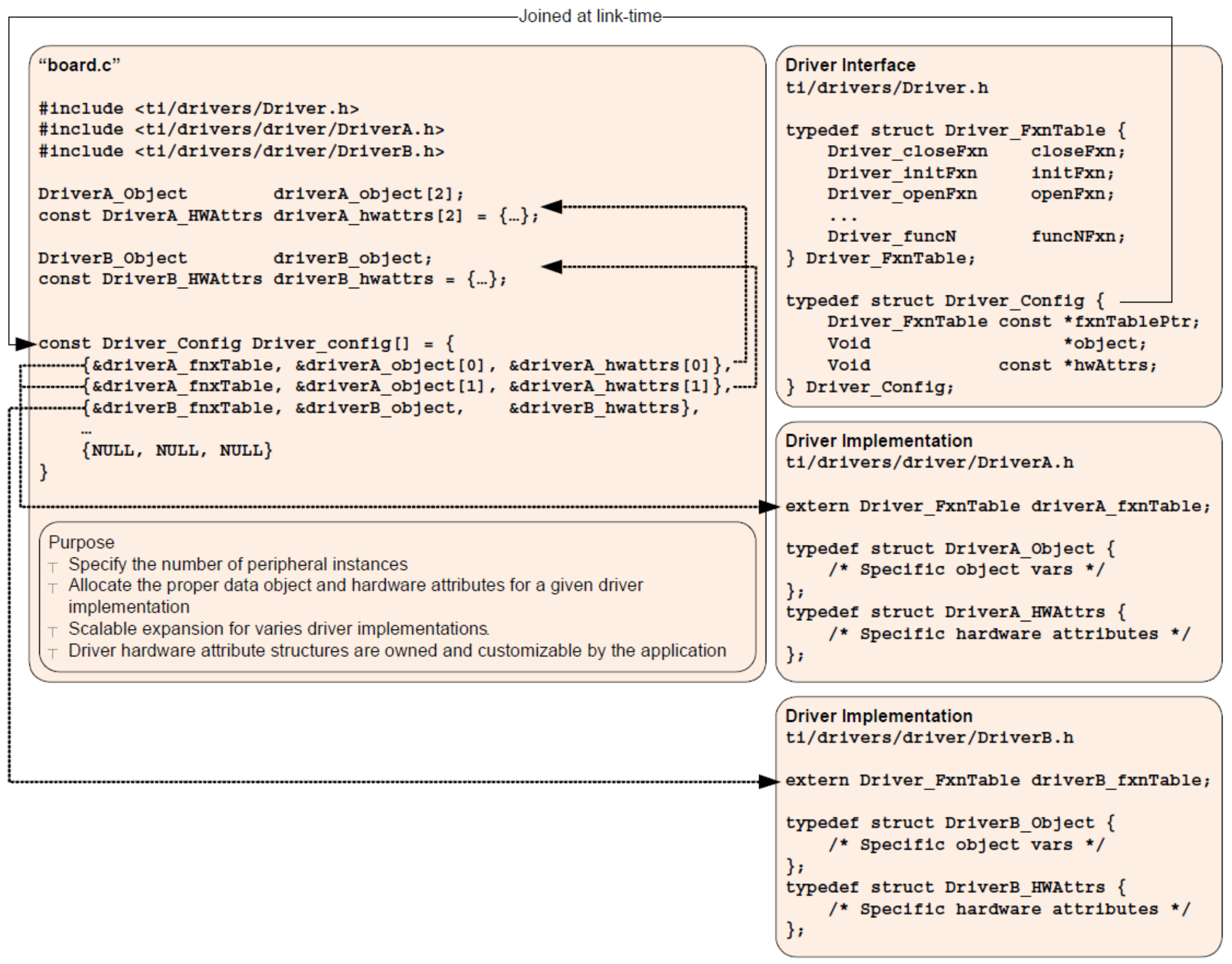
MSB								LSB							
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
RN3	RN2	RN1	RN0	CT	M1	M0	OVF	CRF	FH	FL	L	POL	ME	FC1	FC0
1	1	0	0	0	1	0	0	0	0	0	1	0	0	0	0
C4								10							

Data = 0x10C4 ← Stored in MCU memory



Order of transmission

TI-RTOS Drivers



“board.c”

```
#include <ti/drivers/Driver.h>
#include <ti/drivers/driver/DriverA.h>
#include <ti/drivers/driver/DriverB.h>

DriverA_Object        driverA_object[2];
const DriverA_HWAttrs driverA_hwattrs[2] = {…};

DriverB_Object        driverB_object;
const DriverB_HWAttrs driverB_hwattrs = {…};


const Driver_Config Driver_config[] = {
    {&driverA_fnxTable, &driverA_object[0], &driverA_hwattrs[0]},
    {&driverA_fnxTable, &driverA_object[1], &driverA_hwattrs[1]},
    {&driverB_fnxTable, &driverB_object,    &driverB_hwattrs},
    …
    {NULL, NULL, NULL}
}
```

Purpose
- Specify the number of peripheral instances
- Allocate the proper data object and hardware attributes for a given driver implementation
- Scalable expansion for varies driver implementations.
- Driver hardware attribute structures are owned and customizable by the application

Driver Interface

```
ti/drivers/Driver.h

typedef struct Driver_FxnTable {
    Driver_closeFxn    closeFxn;
    Driver_initFxn     initFxn;
    Driver_openFxn     openFxn;
    ...
    Driver_funcN       funcNFxn;
} Driver_FxnTable;

typedef struct Driver_Config {
    Driver_FxnTable const *fxnTablePtr;
    Void                  *object;
    Void            const *hwAttrs;
} Driver_Config;
```

Driver Implementation

```
ti/drivers/driver/DriverA.h

extern Driver_FxnTable driverA_fxnTable;

typedef struct DriverA_Object {
    /* Specific object vars */
};
typedef struct DriverA_HWAttrs {
    /* Specific hardware attributes */
};
```

Driver Implementation

```
ti/drivers/driver/DriverB.h

extern Driver_FxnTable driverB_fxnTable;

typedef struct DriverB_Object {
    /* Specific object vars */
};
typedef struct DriverB_HWAttrs {
    /* Specific hardware attributes */
};
```

TI-RTOS Drivers

Each driver's interface defines a configuration data structure as:

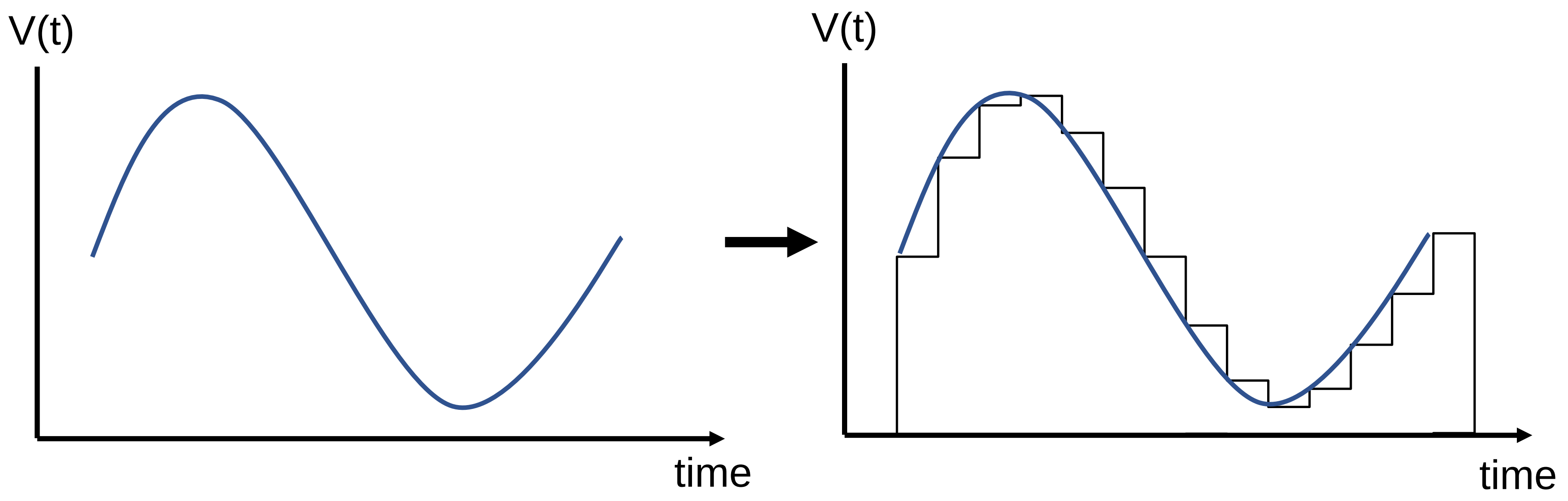
```
typedef struct Driver_Config {
    Driver_FxnTable const *fxnTablePtr;
    Void *object;
    Void const *hwAttrs;
} Driver_Config;


static DriverA_Object driverAObject;

const DriverA_HWAttrs driverAHWAttrs = {
    type field0;
    type field1;
    ...
    type fieldn;
};
```

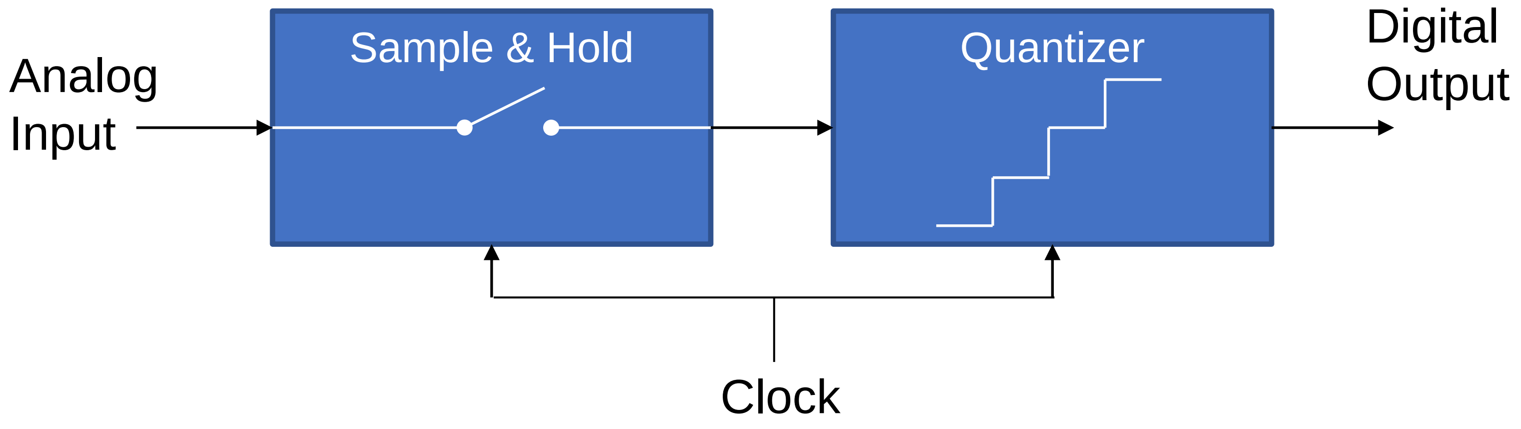
TI-RTOS Driver - Common Issues

1. Index (0,1,2 etc…) into hw_config array
   - `UART_open(Board_UART0, &uartParams);`
   - `Board_UART0 is an index not a base memory address`
   - `Base memory addresses typically used in Tivaware drivers`
2. Modify .h files to number of instances being used
   - `EK_TM4C1294XL.h`
3. Need to add your own device init code
   - `EK_TM4C1294XL_initI2C(void)`

```
typedef enum EK_TM4C1294XL_GPIOName {
    EK_TM4C1294XL_USR_SW1 = 0,
    EK_TM4C1294XL_USR_SW2,
    EK_TM4C1294XL_D1,
    EK_TM4C1294XL_D2,
      ADD MORE HERE (INCREASES COUNT)

    EK_TM4C1294XL_GPIOCOUNT
} EK_TM4C1294XL_GPIOName;
```

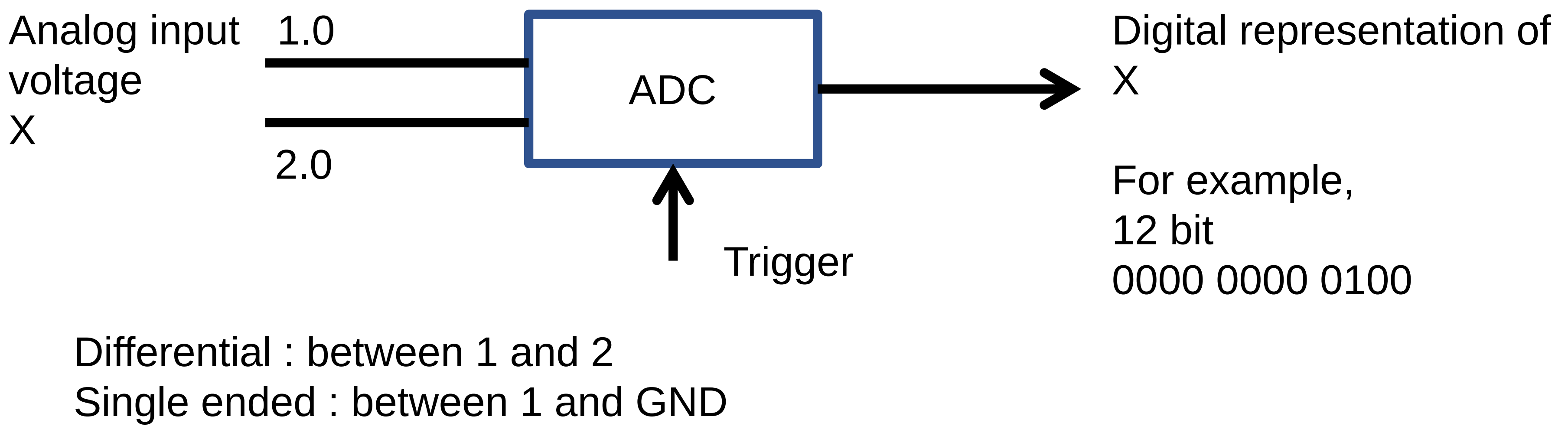
Analog to Digital Conversion





Analog Data and ADC

- Analog data must be converted to digital using Analog to digital convertors (ADC)
- Your Tiva MCU has 2x ADC (ADC0 & ADC1) and is supported with an ADC driver.


Analog input voltage X
1.0
2.0
ADC
Trigger
Digital representation of X
For example,
12 bit
0000 0000 0100
Differential : between 1 and 2
Single ended : between 1 and GND

ADC Module

- Trigger for conversion can be hardware (many options) or software generated
- An analog module contains other hardware – control/status registers, comparator circuitry, sample sequencers, buffers etc
- Voltage references are provided for scaling and calibration
- Your microcontroller has 2 ADC modules ADC0 and ADC1. They have 20 shared analog channels.
- ADC is 12 bit with max sampling rate of 2M samples/sec
- Can be triggered via software, timer, analog comparator, PWM and GPIO
- Can transfer data to memory without CPU processing via DMA

Resolution

- 12 bit ADC yield output in the range 0 to 0 to 2^{12}-1 (if unsigned) or -2^{11} to 2^{11}-1 (if signed)
- If positive input range = 0 to 3.3V, for reading from 0 to 4095, resolution

$$\Delta s = \frac{3.3\text{V}}{4095} = 0.00080586\text{V/step} = 0.806\text{mV/step}$$

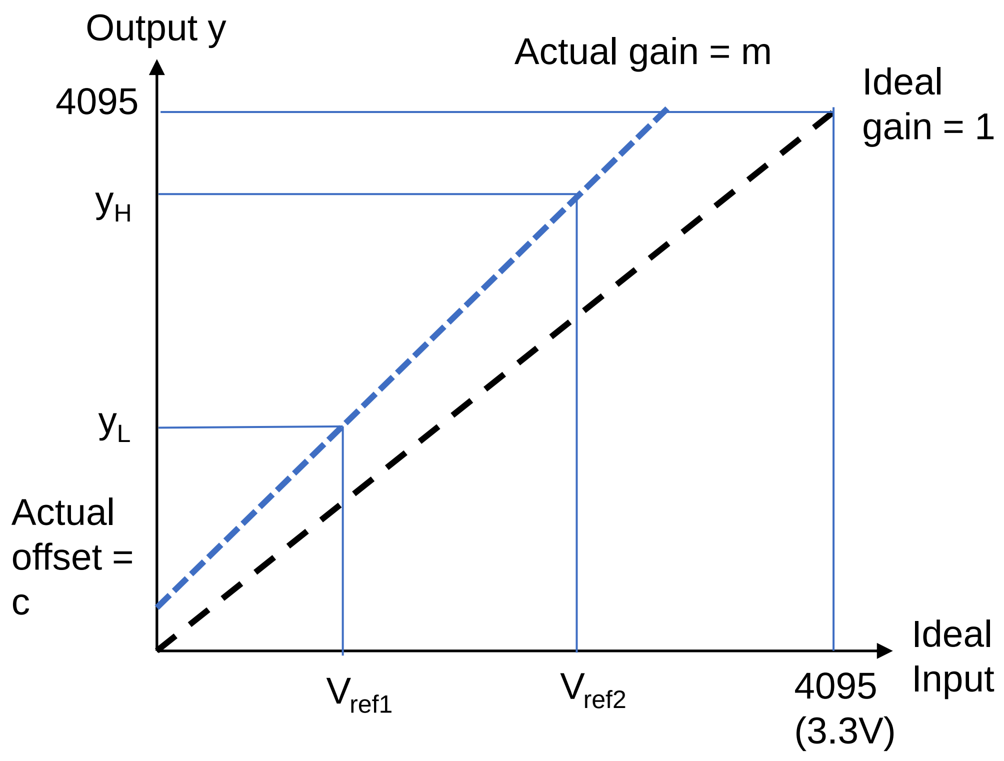
Calibration

- If reading for $\boldsymbol{V_{ref1}}$ is $\mathbf{y_L}$ and reading for $\mathbf{V_{ref2}}$ is $\mathbf{y_H}$, slope and offset can be calculated for a straight line approximation. An unknown reading **x** then is equal to voltage $\mathbf{V_u}$:

$$m = \frac{\mathrm{V}_{\mathrm{ref\,2}} - \mathrm{V}_{\mathrm{ref\,1}}}{y_H - y_L}$$

$$c = \mathrm{V}_{\mathrm{ref\,1}} - \frac{\mathrm{V}_{\mathrm{ref\,2}} - \mathrm{V}_{\mathrm{ref\,1}}}{y_H - y_L} y_L$$

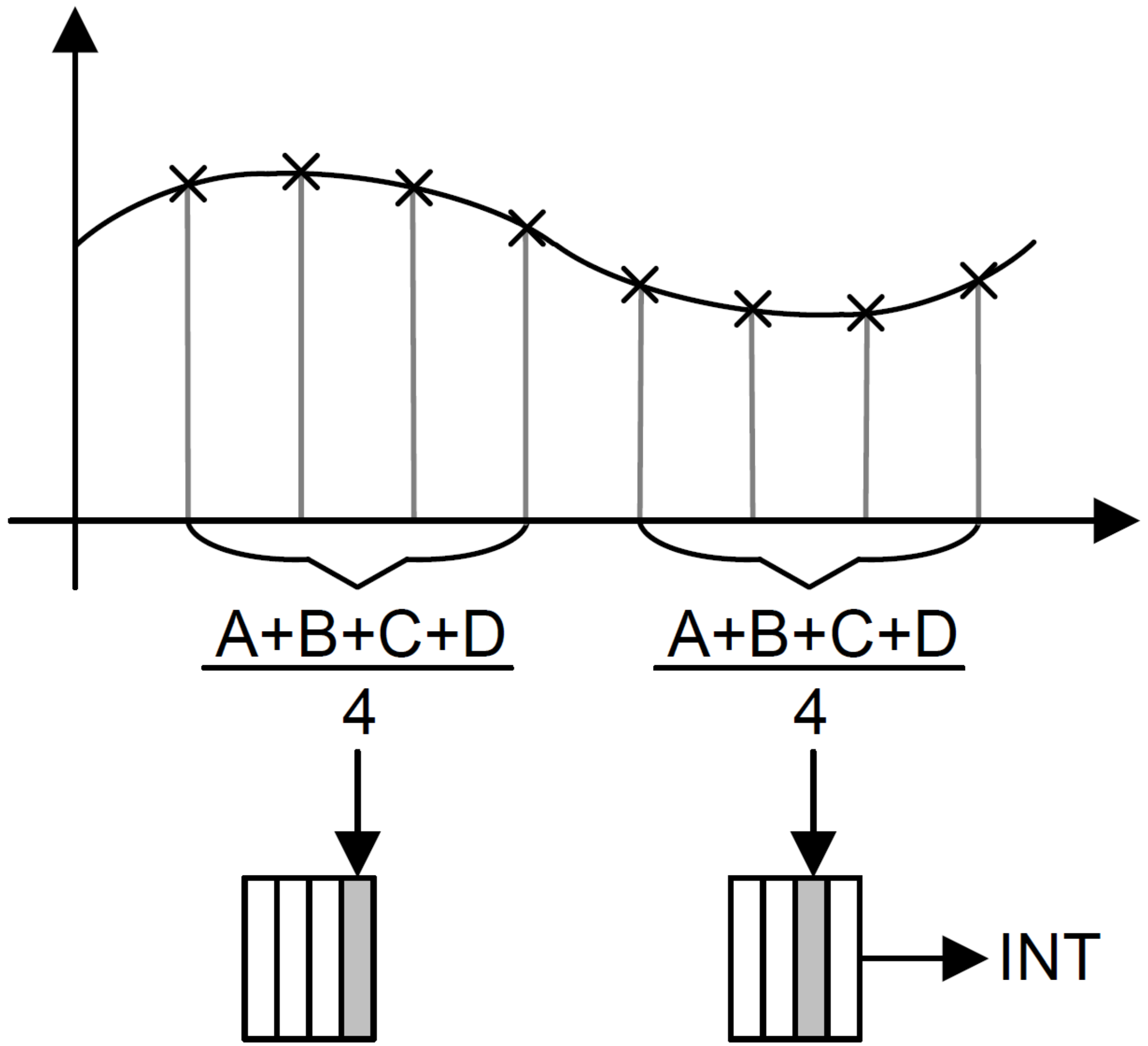
$$\mathrm{V}_{\mathrm{u}} = mx + c$$



Sample sequencers

- Sampling control and data capture are handled by sequencers that can put several samples into a FIFO buffer (1 to 8 samples)
- SS0 captures 8 samples, SS1 and SS2 capture 4 samples and SS3 captures 1 sample.
- Samples can be collected from multiple sources and using different modes if desired
- Note: A changed value will appear only after old values in the buffer are read
- Can choose different ADC pins for single sample in the multiple sample sequence

Sample Averaging

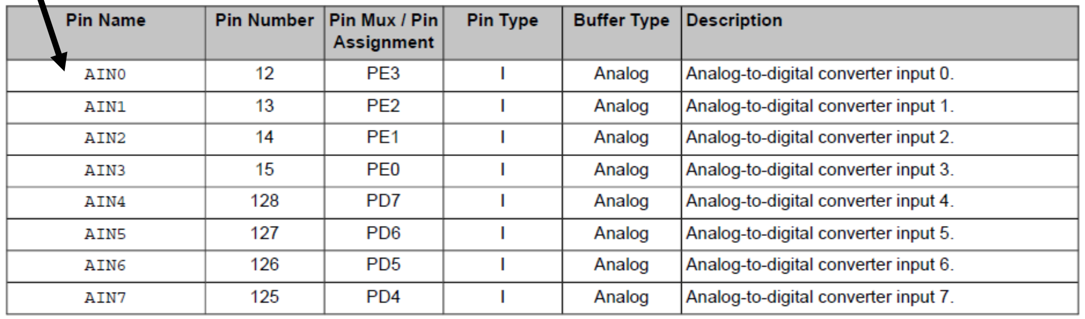


ADC driver

- No TI-RTOS driver included in this version. Use Tivaware driverlib for ADC
- **ADC initialization**
  - Configure and enable the sequencer (sequencer, channel, trigger type, priority)
  - Configure each step
- **ADC capture**
  - Trigger the processor
  - Wait until capture sequencer completes
  - Call the get data API function
- Trigger
  - Can setup a timer in ADC trigger mode to trigger an ADC sequence

ADC Sequence Init Example

```c
#include "driverlib/adc.h"

void ADC0_Init() //ADC0 on PE3 {

    SysCtlPeripheralEnable( SYSCTL_PERIPH_ADC0 );

    SysCtlPeripheralEnable( SYSCTL_PERIPH_GPIOE );

    //Makes GPIO an INPUT and sets them to be ANALOG

    GPIOPinTypeADC( GPIO_PORTE_BASE, GPIO_PIN_3 );

    //uint32_t ui32Base, uint32_t ui32SequenceNum, uint32_t ui32Trigger, uint32_t
    ui32Priority

    #define ADC_SEQ 1;

    #define ADC_STEP 0;

    ADCSequenceConfigure( ADC0_BASE, ADC_SEQ , ADC_TRIGGER_PROCESSOR, 0 );

    //uint32_t ui32Base, uint32_t ui32SequenceNum, uint32_t ui32Step, uint32_t ui32Config

    ADCSequenceStepConfigure( ADC0_BASE, ADC_SEQ , ADC_STEP , ADC_CTL_IE | ADC_CTL_CH0 |
    ADC_CTL_END );

    ADCSequenceEnable( ADC0_BASE, ADC_SEQ );

    ADCIntClear( ADC0_BASE, ADC_SEQ );

}
```



Pin Name	Pin Number	Pin Mux / Pin Assignment	Pin Type	Buffer Type	Description
AIN0	12	PE3	I	Analog	Analog-to-digital converter input 0.
AIN1	13	PE2	I	Analog	Analog-to-digital converter input 1.
AIN2	14	PE1	I	Analog	Analog-to-digital converter input 2.
AIN3	15	PE0	I	Analog	Analog-to-digital converter input 3.
AIN4	128	PD7	I	Analog	Analog-to-digital converter input 4.
AIN5	127	PD6	I	Analog	Analog-to-digital converter input 5.
AIN6	126	PD5	I	Analog	Analog-to-digital converter input 6.
AIN7	125	PD4	I	Analog	Analog-to-digital converter input 7.

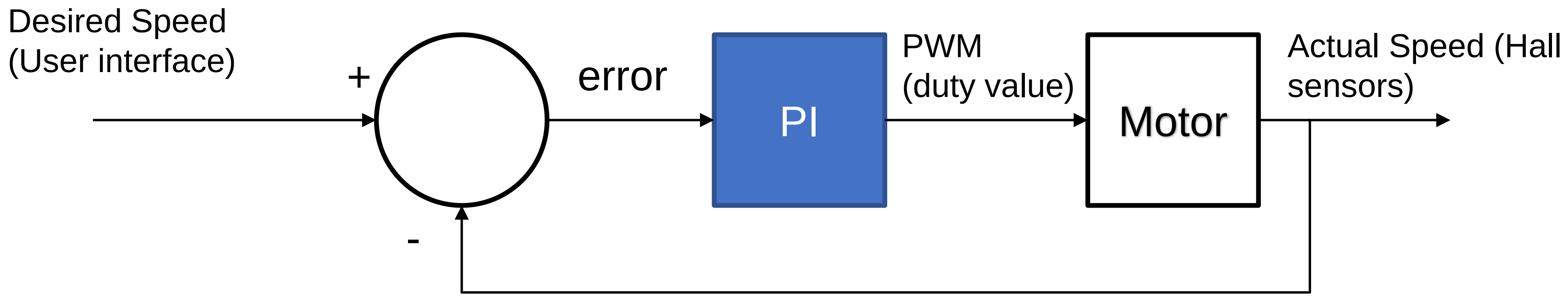
ADC Sequence Example

```c
uint32_t ADC0_Read() {
        uint32_t pui32ADC0Value[1];
        // Trigger the ADC conversion.
        ADCProcessorTrigger( ADC0_BASE, ADC_SEQ );
        // Wait for conversion to be completed.
        while(!ADCIntStatus( ADC0_BASE, ADC_SEQ , false) ) { }
        //Clear ADC Interrupt
        ADCIntClear( ADC0_BASE, ADC_SEQ );
        // Read ADC FIFO buffer from sample sequence.
        ADCSequenceDataGet( ADC0_BASE, ADC_SEQ , pui32ADC0Value );
        return ( pui32ADC0Value[0] );
 }
```

This can also be done via an interrupt service routine and wrapped in a Hwi

Simple PI Control

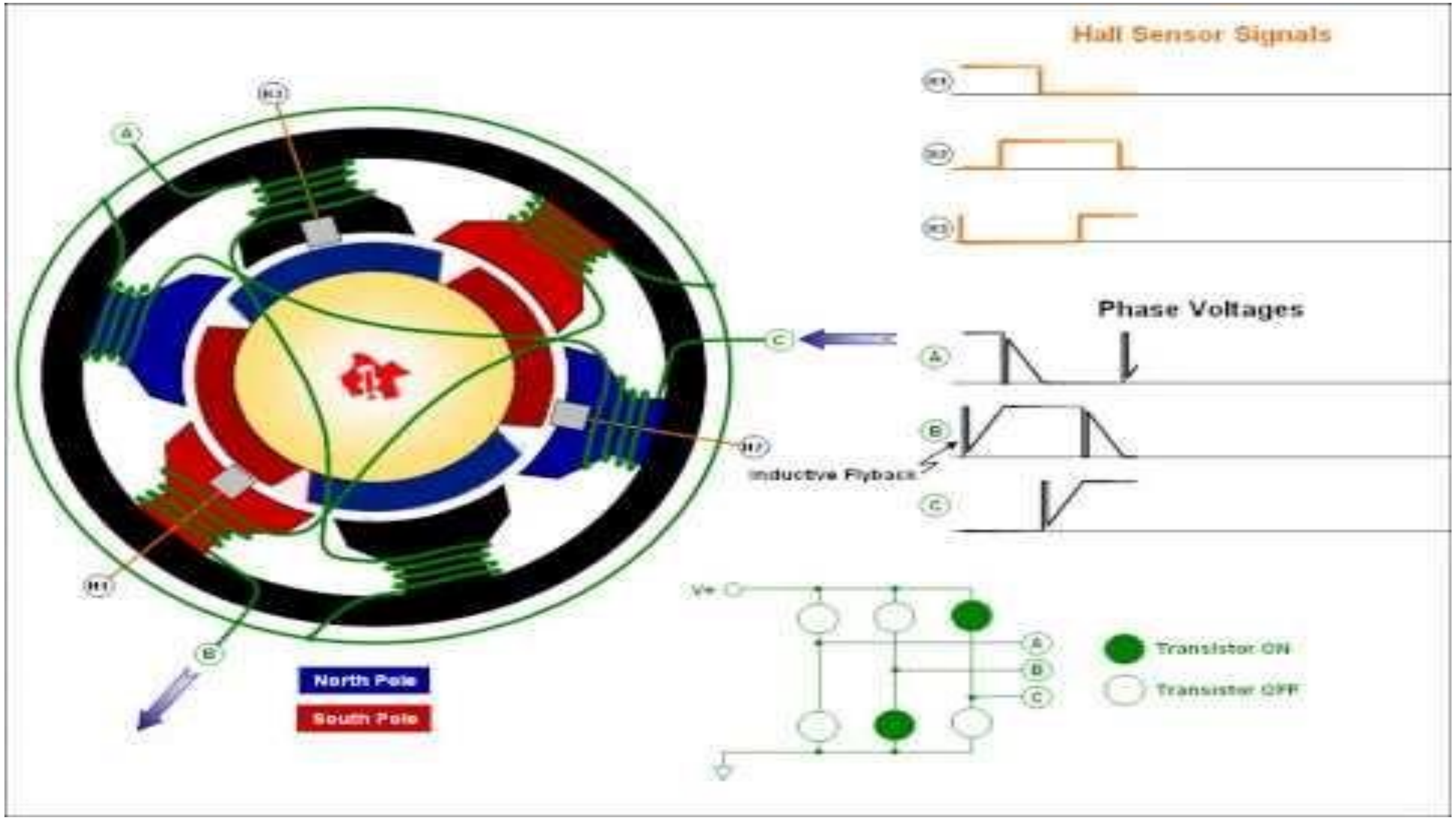
- Proportional (P) and Integral (I) Control
  - Simple method to control the speed of a motor
- Power (PWM) = K_p*error + K_i*∑(error)
  - error = (desired speed – actual speed)
  - K_p & K_i user defined constants
    - Can be calculated but usually experimentally determined by trial and error
  - Requires storing previous error in memory
  - Summation of integral should be capped to a maximum



PI Example

```
Integral_error = 0
Control Loop Start (use a timer):
    1.  error = desired_speed – actual_speed
    2.  Integral_error = integral_error + error*dt     (can leave out dt as it's a constant, helps to set an max value)
    3.  output = Kp*error + Ki*integral               (experiment with Kp, Ki and Kd gains)
    4.  set_duty(output)                              (Kp gains should be chosen to put output into the correct scale)
Loop
```

Brushless DC Motor



Assignment Hints & FAQ

Motor Kit

The DRV8232 Motor Driver Kit has the following settings (Please see the DRV8232RH datasheet for more detail):

- The GAIN pin has been tied to GND with a 47k resistor so has a gain of 10 for the current amplifier
- The current shunt resistor values are 0.007 Ohms for each current sense line.
- See section 8.3.4.1 of driver datasheet for an equation to calculate current from the voltage measurement
- **Vref** is 3.3V for the motor driver and **VSOx** is the voltage that you are measuring.
- This is actually not accurate as it depends on the number of pole pairs within the brushless motor. Easiest way to check how many hall effect sensor transitions occur is to manually rotate the motor by a number of rotations and count the number of times the hall effect sensors trigger.
- The Motor is being supplied with 24V from the 24V DC powerpack. This can be used to calculate the motor power usage.

Sensors

- Feel free to use opensource libraries for sensors if they are available (make sure to reference them in your code and reports)
- Some examples can be found in the sensorlib folder within Tivaware (C:\ti\tivaware_c_series_2_1_4_178\sensorlib)
- The **TMP107** can be challenging as the **Tx** line actually is tied to the Rx line of the UART. Therefore you will need to assume all **Tx** bytes will be echoed to the **Rx** line and you need to ignore them in your code.
- Similar to the byte ordering for the OPT3001, the TMP107 command and address bytes have a specific bit ordering (consult the datasheet). The bit ordering is reversed for the TMP107.

Touch Screen

- The touchscreeninthandler needs to be setup within a hwi for it to work within the TI-RTOS framework
- Use int number 33 as the interrupt
- The touchscreen also uses ADC0 within it's implementation so make sure you don't cause conflict by using ADC0 for current sensing
- Timer 1 is used by the touch screen so make sure you also don't use it for your application, or it will interfere with the touch interface

Motor Driver

1. Init Hall GPIO lines
2. Init Motor Lib
3. Start Motor
   - Force single `updateMotor` ()
4. Control Loop
   - Update pwm from PI code
   - `setDuty()`

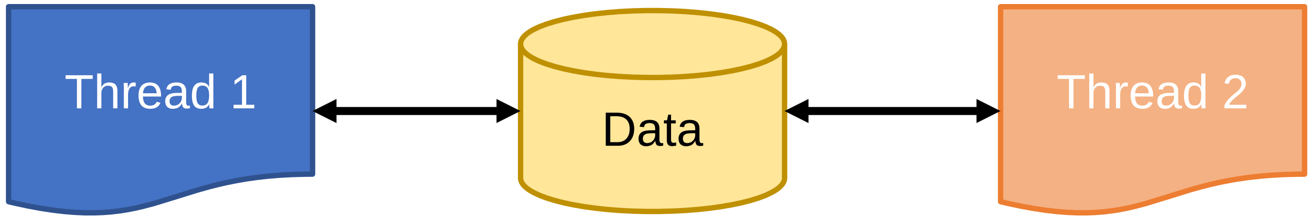
GPIO Hwi Interrupt for Hall Lines

1. Read Hall Sensors (GPIO)
2. Increment speed counter
3. `updateMotor()`

Swi Interrupt for Hall Lines

1. Read Hall Sensors (GPIO)
2. `updateMotor()`

Data sharing



Problem

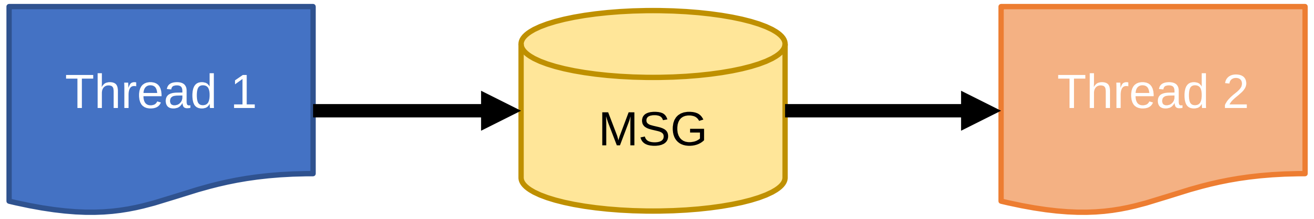
- Data may need to be moved between threads
- Multiple threads operating on shared data must do so without corrupting data structure integrity

Solution

- Just use globals and don't worry about!
- Synchronization primitives (Mutex, Semaphore) may not be sufficient.
- Other mechanisms such as message queues, mailboxes, gates and Events

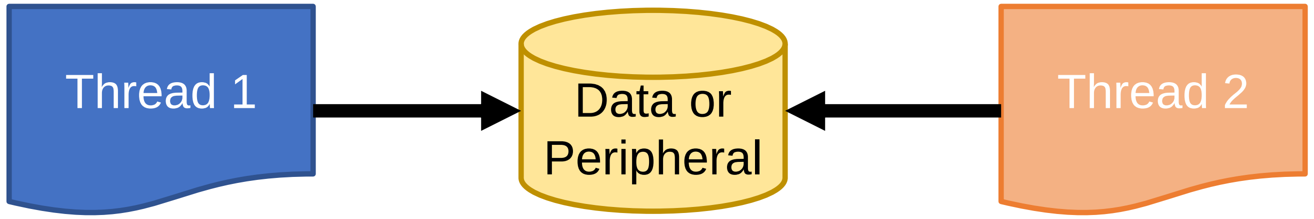
Resource Sharing

Producer – Consumer Model



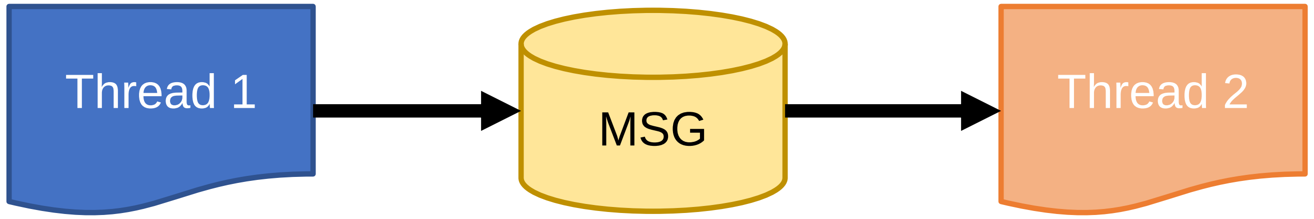
- Thread 1 produces a buffer or Msg and places into BIOS “container”
- Thread 2 blocks until available, then consumes it when signalled (no contentions

Concurrent Access Model



- Any thread could access “Resource at any time (no structured protocol)
- Pre-emption of one thread by another can cause contention

Producer – Consumer Model



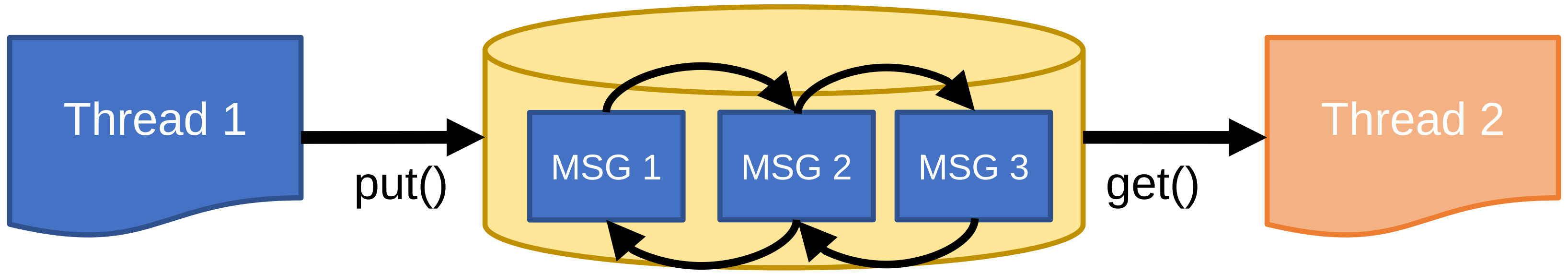
BIOS Container Object

- Data communicated in one direction - One thread waits for the other to **produce**
- Thread 2 often BLOCKS until data is **produced** which is then **consumed**
- Contention is avoided by threads using same protocol

Examples:

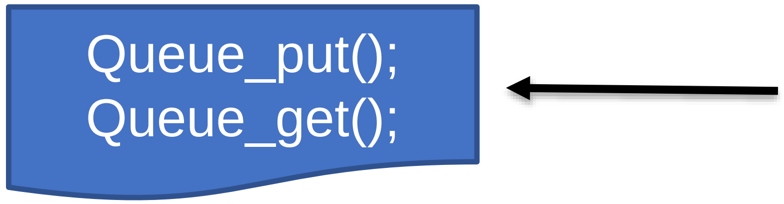
- BIOS: **Queue, Mailbox, Stream IO**
- Either built in synchronisation or user based signalling via semaphores and events

Message Queues



Queue

- A **Queue** is a bios object that can contain *anything*
- Data in a queue is called a Msg – simply a userdefined struct
- API:

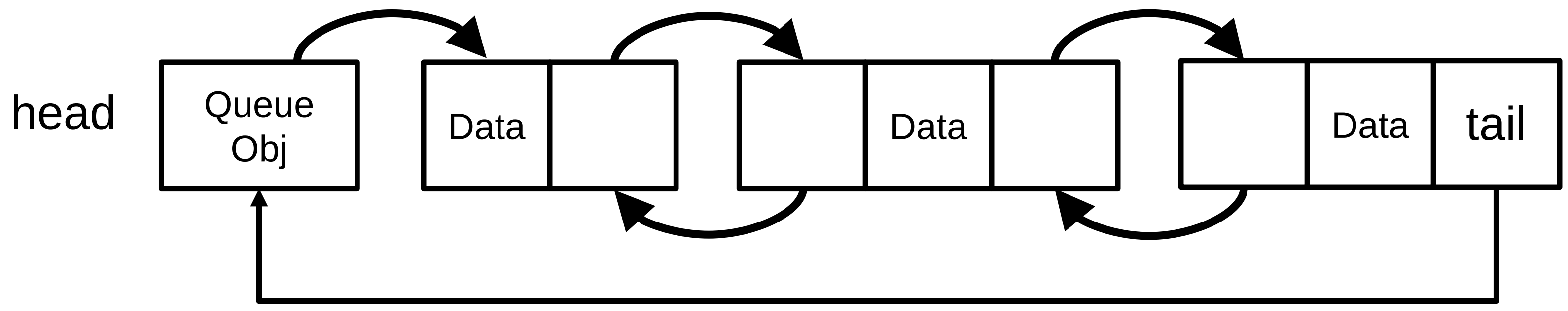


Queue_put();
Queue_get();

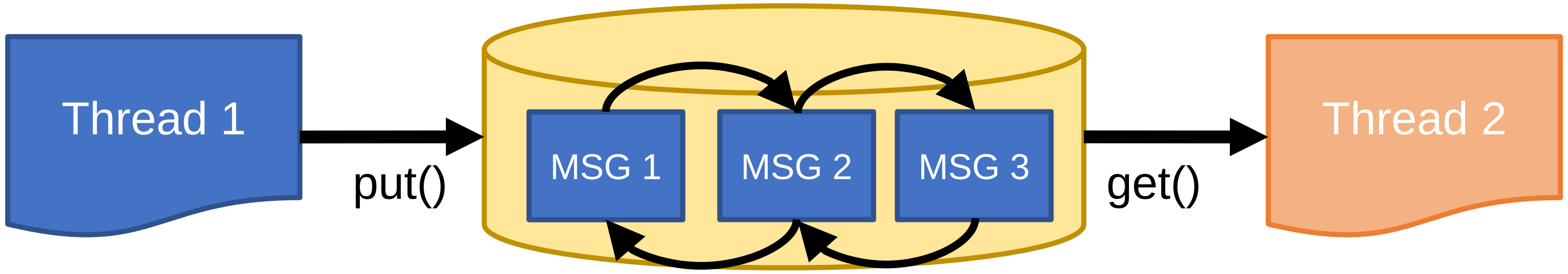
- Simple but no signalling built in

Queues

- Implemented as doubly-linked lists
- Contains head, data and links (pointers) to next node and previous node
- Elements can be inserted or removed anywhere
- No theoretical maximum size (limited to memory)



Syncronising Queues



Queue

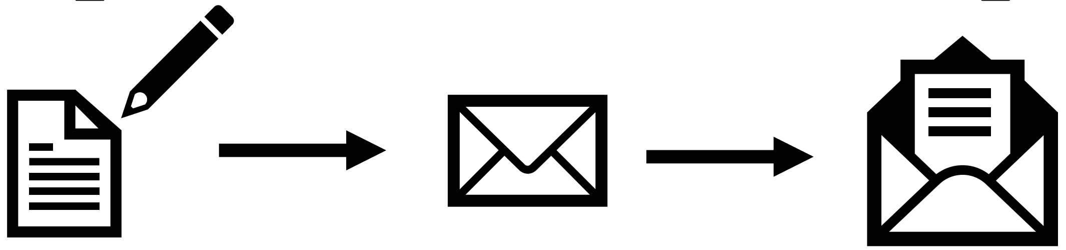
- Use semaphore to synchronise writer/reader

Writer

```
Queue_put(&myQ, msg)
Semaphore_post(&Sem)
```

Reader

```
Semaphore_pend(&Sem)
Msg = Queue_get(&myQ)
```

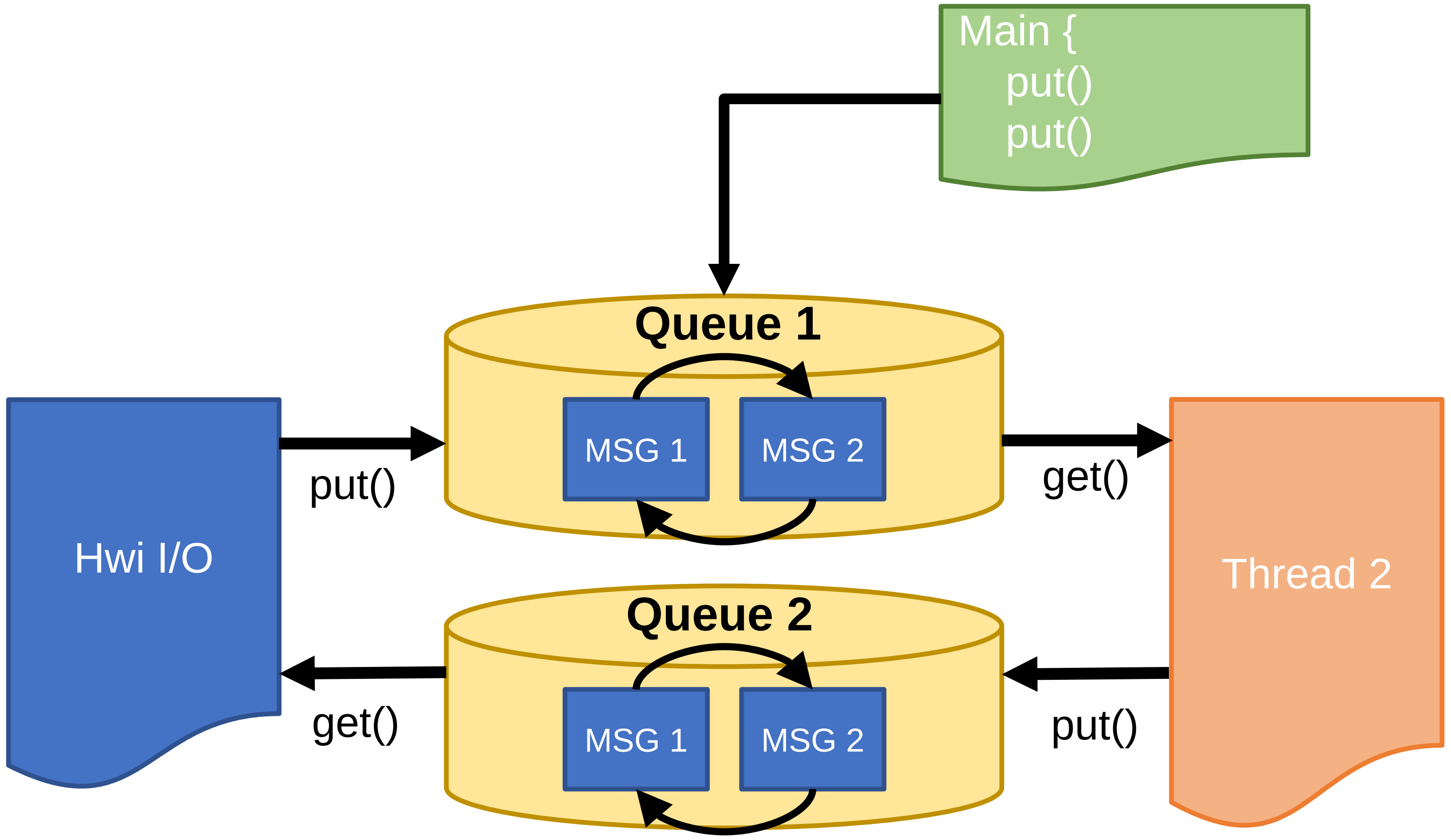


Message Queue Structure

- Requires **Queue_Elem** to be the first field in struct
- Good practice to include user defined thread ID of producer/writer
- The **fxn** that creates the message owns the memory – the receiver simply gets a pointer to that memory – so it is not copy-based

```
typedef struct MsgObj {
        Queue_Elem elem; /* first field for Queue */
        int id; /* writer task id */
        Char data; /* message value */
} MsgObj;
```

Using Queues in a System



Example : Main program

```
Queue_Handle queue;
MsgObj msg_mem[NUMMSGS];

int main(void) {
    int i;
    MsgObj *msg = &msg_mem;
    my_queue = Queue_create(NULL, NULL)

    /* Put all messages in Queue */
    for (i = 0; i < NUMMSGS; msg++, i++) {
        Queue_put(my_queue, msg->elem);
    }

    BIOS_start();
    return(0);
}
```

Dynamic memory allocation only at initialization

NUMMSGS message structures. Initially put in the ‘free’ queue.

pointers are used

Example: writer

```
void writer(int id) {
     MsgObj *msg
     int i;
     for (i = 0; i < NUMMSGS; i++) {

         /* Get msg from the free list. Since reader is higher
         priority and only blocks on sem, list is never empty. */
         msg = Queue_get(my_queue);

        /* fill in value */
         msg->id = id;
         msg->data = (i & 0xf) + 'a';

         /* put message */
         Queue_put(my_queue, msg->elem);
         /* post semaphore */
         Semaphore_post(sem);
     }
}
```

Only when the reader block does the writer run here.

In this example, messages are just characters. After putting every message, the writer does a semaphore post

Example: reader

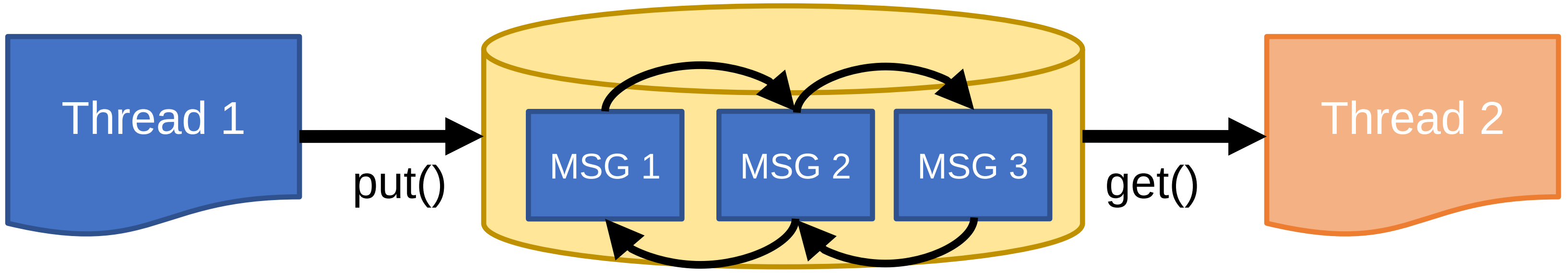
```
void reader() {
     MsgObj *msg;
     int i;

     for (i = 0; i < NUMMSGS; i++) {
          /* Wait for semaphore to be posted by writer(). */
          Semaphore_pend(sem, BIOS_WAIT_FOREVER);

          /* get message */
          msg = Queue_get(my_queue);

          /* free msg */
          Queue_put(my_queue, msg->elem);
     }
}
```

- The reader pends on a semaphore if the message queue is empty.
- When the writer does a post, it will run, read a message and replace it with a 'free' message.

Summary

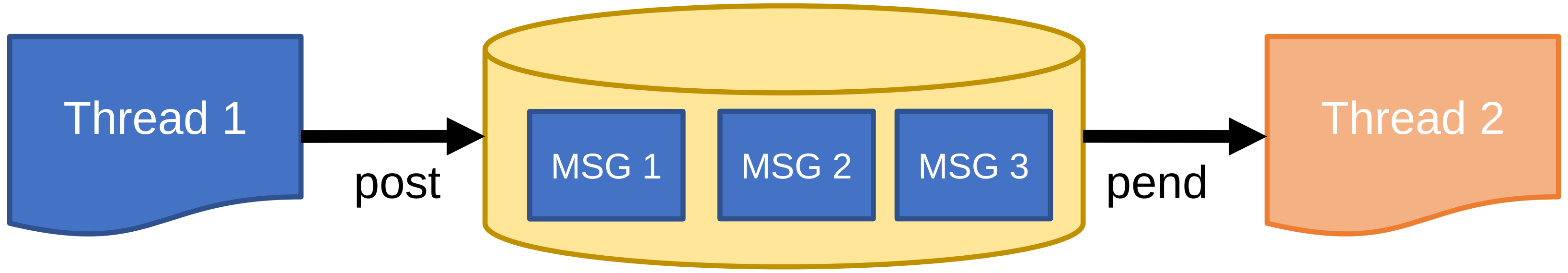
User Setup

1. Declare Queue in CFG
2. Define (typedef) structure of Msg
3. Fill in Msg – define “elements”
4. Send/receive data from the queue



Queue

Mailbox



Mailbox

- Fixed size BIOS object that can contain anything you like
- Fixed length – Number of Msgs (length) and Message size
- **Advantages**
  - simple FIFO, easy to use, contains ***built-in semaphore*** for signaling
- **Disadvantage**
  - Copy-based (both reader/writer own a copy) – best if used to pass pointers or small Msgs

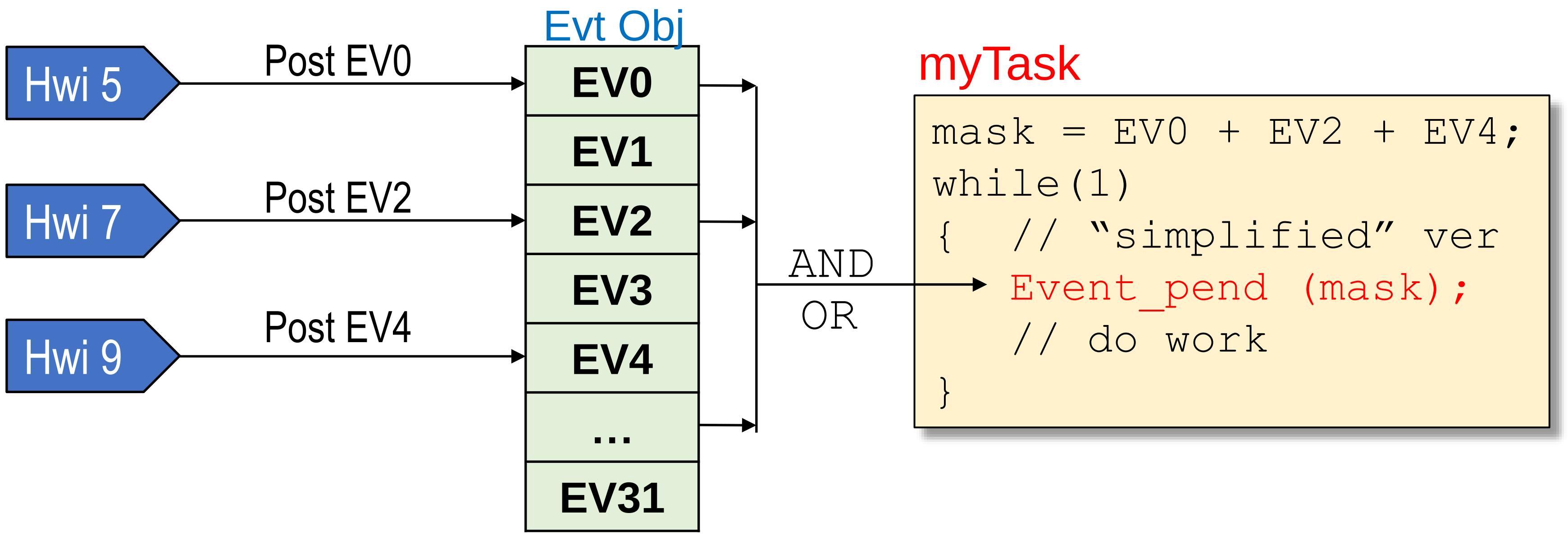
```
Mailbox_post (&Mbx, &Msg, timeout);   /*blocks if full*/
Mailbox_pend (&Mbx, &Msg, timeout);   /*blocks if empty*/
```

SYS/BIOS mailbox

```
Mailbox_handle Mailbox_create(SizeT msgSize,
                              UInt numMsgs,
                              const Mailbox_Params *params,
                              Error_Block *eb);

Bool Mailbox_pend(Mailbox_Handle handle, Ptr buf, UInt timeout);
```

- Mailbox module copies the buffer to fixed size internal buffers
- The size and number of the internal buffers is specified (as above)
- A mailbox provides configuration parameters to allow specification of events that tasks can use to specify events to wait on such as **readerEvent** and **readerEventID**, **writerEvent** and **writerEventID**

Events



```
mask = EV0 + EV2 + EV4;
while(1)
{   // "simplified" ver
    Event_pend (mask);
    // do work
}
```

- Sometimes multiple conditions need to be satisfied before action
- A semaphore by itself is not enough. The semaphore object is modified to have an event field and support events.
- Semaphore_pend() only waits on one flag – a semaphore.
- Use Events. Can **OR** or **AND** event **ID**s with bit masks

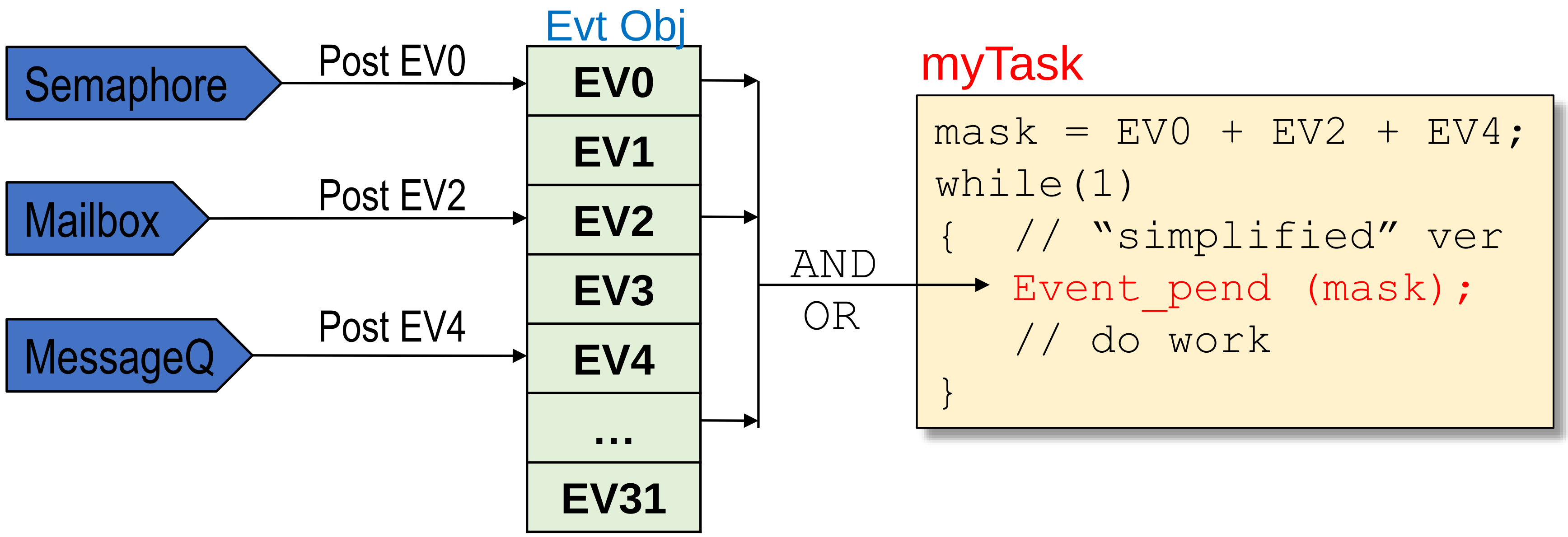
Event Module API

- The key "Explicit Post" and Pend APIs are:

```
Event_post (Event_Handle, Event_Id_xx);
Event_pend (Event_Handle, andMask, orMask,
timeout);
```

- Only one task can pend on an event
- A single Event instance can manage up to 32 events
- Only tasks can call Event_pend() but Hwi, Swi and tasks can call Event_post()

Implicit Semaphore Events



```
mask = EV0 + EV2 + EV4;
while(1)
{   // "simplified" ver
    Event_pend (mask);
    // do work
}
```

- Other APIs, as shown above, can also post events – implicitly – the eventId is part of the params structure (e.g. Semaphore):
- So, even a standard `Semaphore_post(Sem)` can post an event !

Example

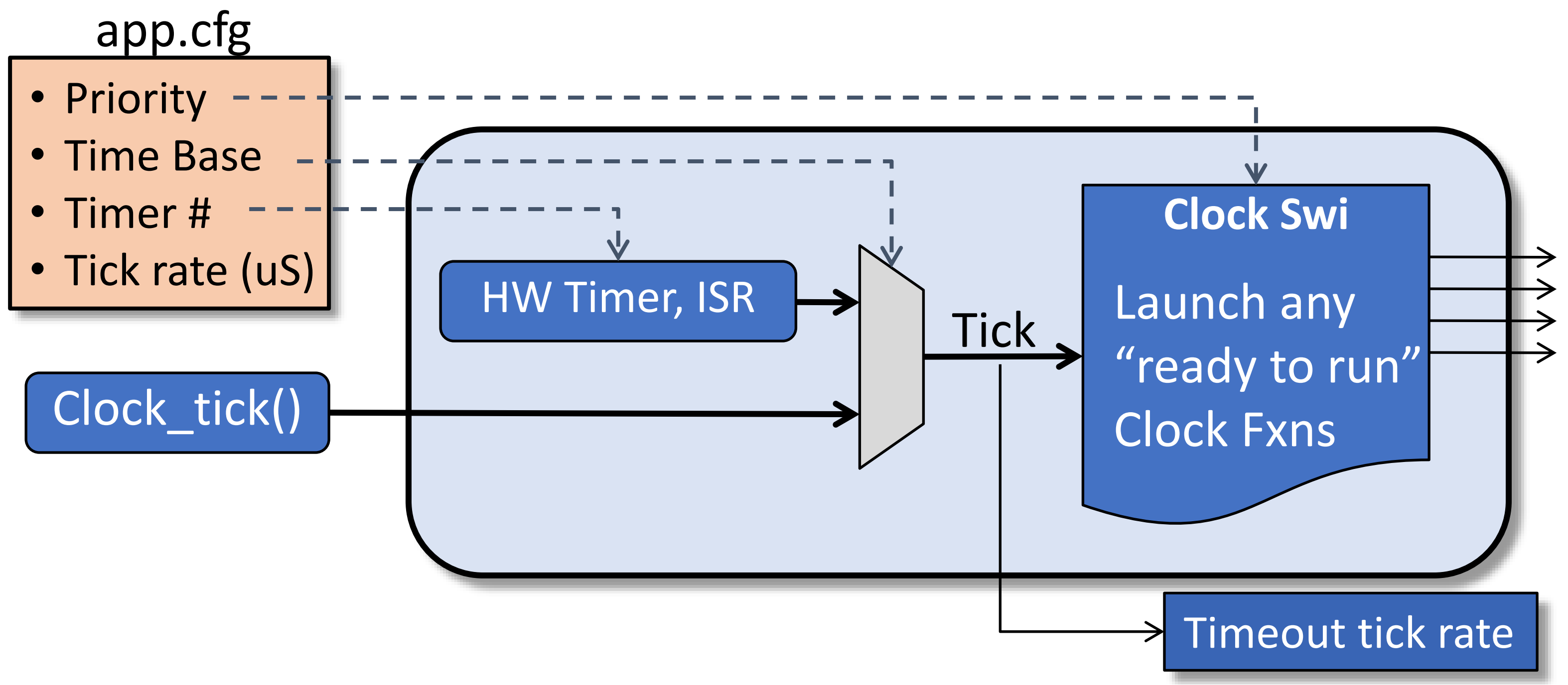
```
main() {
    ...
    /* create an Event object. All
events are binary */
    myEvent = Event_create(NULL, &eb);
    if (myEvent == NULL) {
        System_abort("Event create
          failed");
    }
}

isr0() {
    ...
    Event_post(myEvent, Event_Id_00);
    ...
}

isr1() {
    ...
    Event_post(myEvent, Event_Id_01);
    ...
}
```

```
task() {
    UInt events;
    while (TRUE) {
    /* Wait for ANY of the ISR events to
be posted */
        events = Event_pend(myEvent,
Event_Id_NONE, Event_Id_00 +
Event_Id_01, BIOS_WAIT_FOREVER);

    /* Process all the events that have
    occurred */
        if (events & Event_Id_00) {
                processISR0();
        }
        if (events & Event_Id_01) {
                processISR1();
        }
    }
}
```

SYS/BIOS Timing Services

- **Clock Module**
  - Creates a periodic system tick
    - Clock module uses the Timer module to get a hardware-based tick.
  - Can be configured to trigger Clock Swi functions at regular intervals
  - All SYS/BIOS APIs interpret timeouts in terms of Clock ticks.
- **Timer module**
  - Provides a standard interface for using timer peripherals
- **Seconds module**
  - provides a means for maintaining the current time and date, as defined by the number of seconds since 1970
- **Timestamp**
  - module provides simple timestamping services for benchmarking code and adding timestamps to logs.

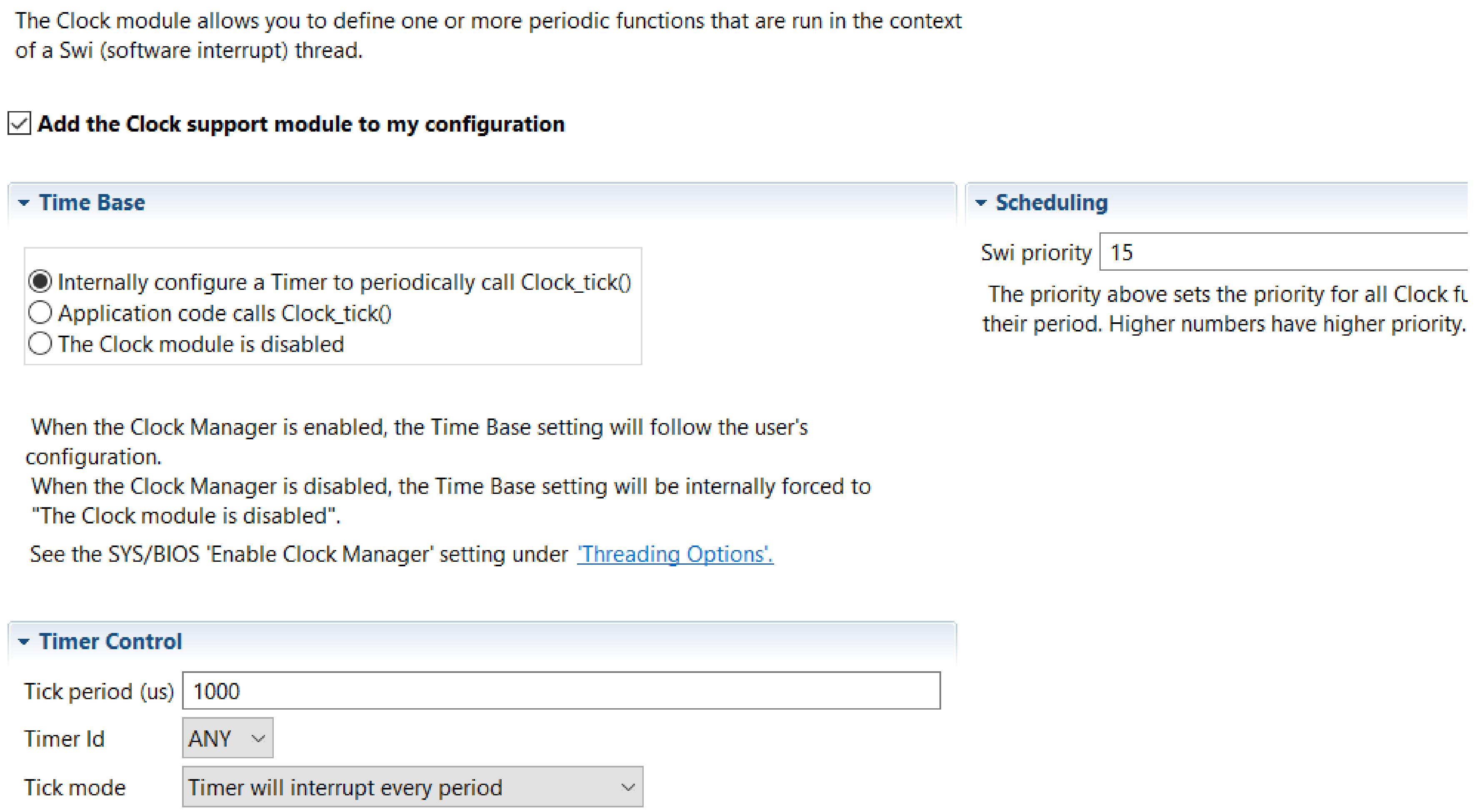
SYS/BIOS Clock Module



- Makes setting up hardware timer easy
- Create different event rates from a timer
- Clock Swi used to launch periodic functions and N ticks
- Uses Timer 0 using default CPU clock source

Adapted from TI

Configuring Clock Module



The Clock module allows you to define one or more periodic functions that are run in the context of a Swi (software interrupt) thread.

☑ **Add the Clock support module to my configuration**

Time Base

● Internally configure a Timer to periodically call Clock_tick()
○ Application code calls Clock_tick()
○ The Clock module is disabled

When the Clock Manager is enabled, the Time Base setting will follow the user's configuration.
When the Clock Manager is disabled, the Time Base setting will be internally forced to "The Clock module is disabled".

See the SYS/BIOS 'Enable Clock Manager' setting under 'Threading Options'.

Scheduling

Swi priority: 15

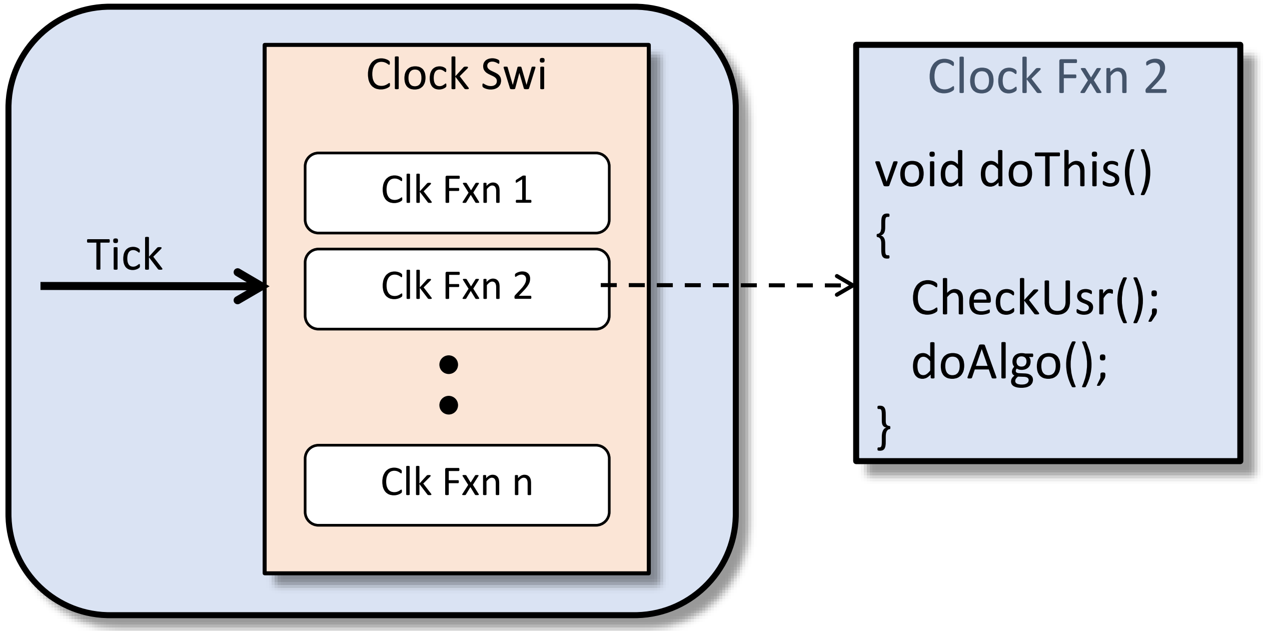
The priority above sets the priority for all Clock fu
their period. Higher numbers have higher priority.

Timer Control

Tick period (us)	1000
Timer Id	ANY
Tick mode	Timer will interrupt every period

Clock Functions

- For each Clock Function, user specifies function to run and # ticks between runs (period)
- "Tick" launches Clock Swi which compares running "tick count" with "period" to determine if each fxn should run:

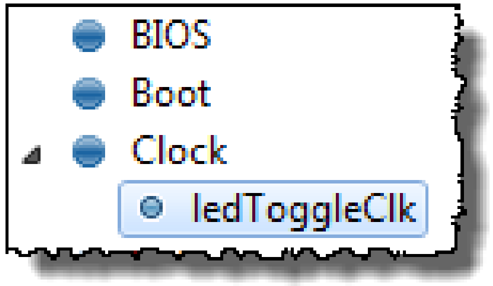


```
void doThis()
{
   CheckUsr();
   doAlgo();
}
```

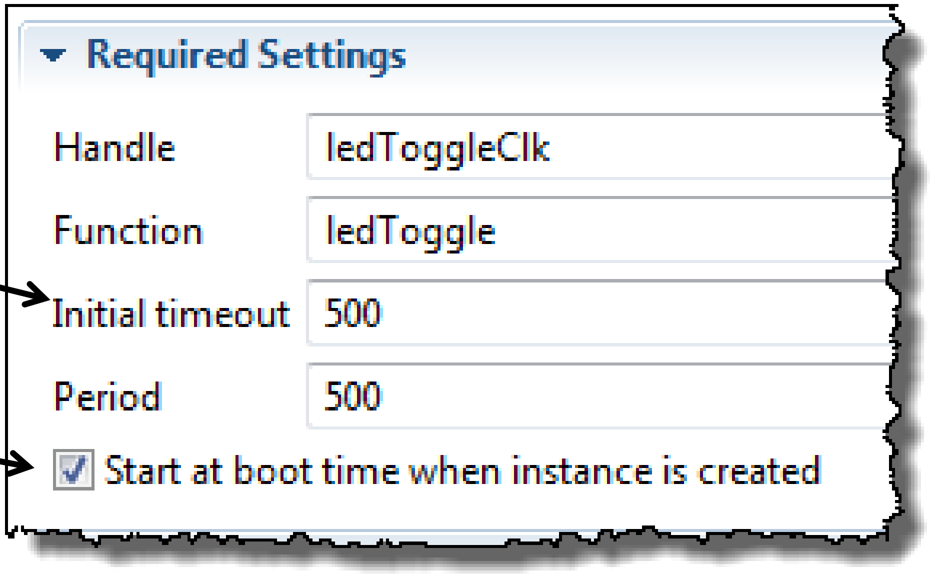
- Clock Functions must complete within one System Tick
- Break long functions into multiple threads (if needed)

Configuring a *Clock Fxn* – *Statically via* GUI

1 Insert new Clock Fxn *(Outline View)*



2 Configure Clock Fxn – Object name, function, init timeout, period:



Required Settings

Handle	ledToggleClk
Function	ledToggle
Initial timeout	500
Period	500

[✓] Start at boot time when instance is created

For "one-shot", set initial timeout to"*value", then* set period = 0

To START the Clock Fxn at runtime, check this box

Using Clock Functions

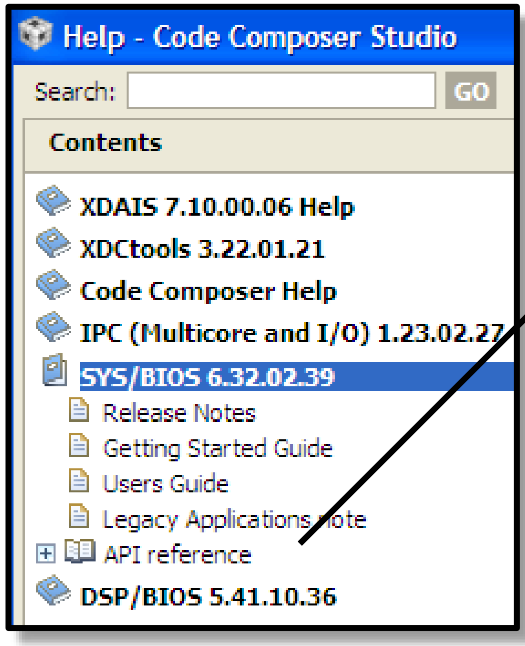
- For each Clock function, user specifies function to run and # ticks between runs (period)
- “Tick()” launches Clock Swi which compares running “tick count” with “period” to determine if each fxn should run:

Using Clk Fxn in task based system

- **Clk Fxn – Swi (Check keyboard) Swi Pri #15**
- **PID Loop – Task Pri #15**
- **Other task – lower Priority**

Clk Swi fxn posts a semaphore that unblocks a keyboard task of lower priority

More Help

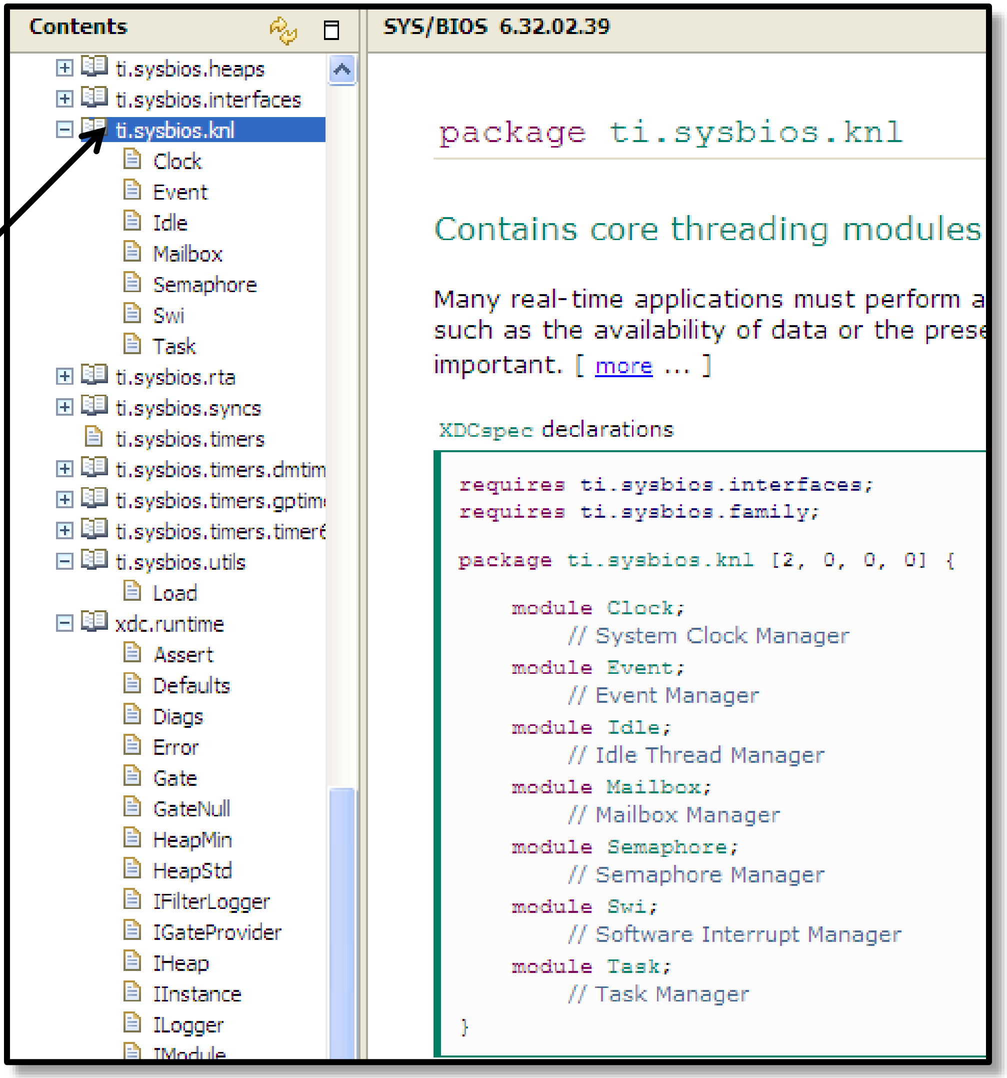


Help - Code Composer Studio

Search: GO

Contents

- XDAIS 7.10.00.06 Help
- XDCtools 3.22.01.21
- Code Composer Help
- IPC (Multicore and I/O) 1.23.02.27
- SYS/BIOS 6.32.02.39
  - Release Notes
  - Getting Started Guide
  - Users Guide
  - Legacy Applications Note
  - API reference
- DSP/BIOS 5.41.10.36



Contents

- ti.sysbios.heaps
- ti.sysbios.interfaces
- ti.sysbios.knl
  - Clock
  - Event
  - Idle
  - Mailbox
  - Semaphore
  - Swi
  - Task
- ti.sysbios.rta
- ti.sysbios.syncs
- ti.sysbios.timers
- ti.sysbios.timers.dmtim
- ti.sysbios.timers.gptim
- ti.sysbios.timers.timer6
- ti.sysbios.utils
  - Load
- xdc.runtime
  - Assert
  - Defaults
  - Diags
  - Error
  - Gate
  - GateNull
  - HeapMin
  - HeapStd
  - IFilterLogger
  - IGateProvider
  - IHeap
  - IInstance
  - ILogger
  - IModule

SYS/BIOS 6.32.02.39

package ti.sysbios.knl

Contains core threading modules

Many real-time applications must perform a
such as the availability of data or the prese
important. [more ...]

XDCspec declarations

```
requires ti.sysbios.interfaces;
requires ti.sysbios.family;

package ti.sysbios.knl [2, 0, 0, 0] {

    module Clock;
        // System Clock Manager
    module Event;
        // Event Manager
    module Idle;
        // Idle Thread Manager
    module Mailbox;
        // Mailbox Manager
    module Semaphore;
        // Semaphore Manager
    module Swi;
        // Software Interrupt Manager
    module Task;
        // Task Manager
}
```

TI-RTOS FAQ

- #include <xdc/cfg/global.h>   //header file for statically defined objects
- Kernel Object calling context can be found in help docs (**demo**)
- Put high level init functions in Idle task or other task threads.
  - Interrupts are not enabled until BIOS_start() is called.