2017 MASTERs Conference

Creating Advanced PIC32 Embedded Applications Using MPLAB® Harmony

21017 DEV6

Class Objectives

When you walk out of this class, you will …

- **Be able to explain the MPLAB® Harmony key concepts**

- **Be able to create MPLAB Harmony based project with multiple applications that use peripheral drivers and system services**

- **Be able to run multiple applications, drivers and system services in an RTOS environment**

Class Agenda

MPLAB® Harmony Key Concepts

Harmony Drivers and System Services

Lab1: Create a MPLAB Harmony Application using MPLAB Harmony Drivers and System Services.

Harmony Drivers Advanced Usage

Lab2: Use Harmony Driver in Multi Instance Configuration

Using MPLAB Harmony in an RTOS environment

Lab3: Add RTOS to the Application

Summary

Class Agenda

MPLAB® Harmony Key Concepts

Harmony Drivers and System Services

Lab1: Create a MPLAB Harmony Application using MPLAB Harmony Drivers and System Services.

Harmony Drivers Advanced Usage

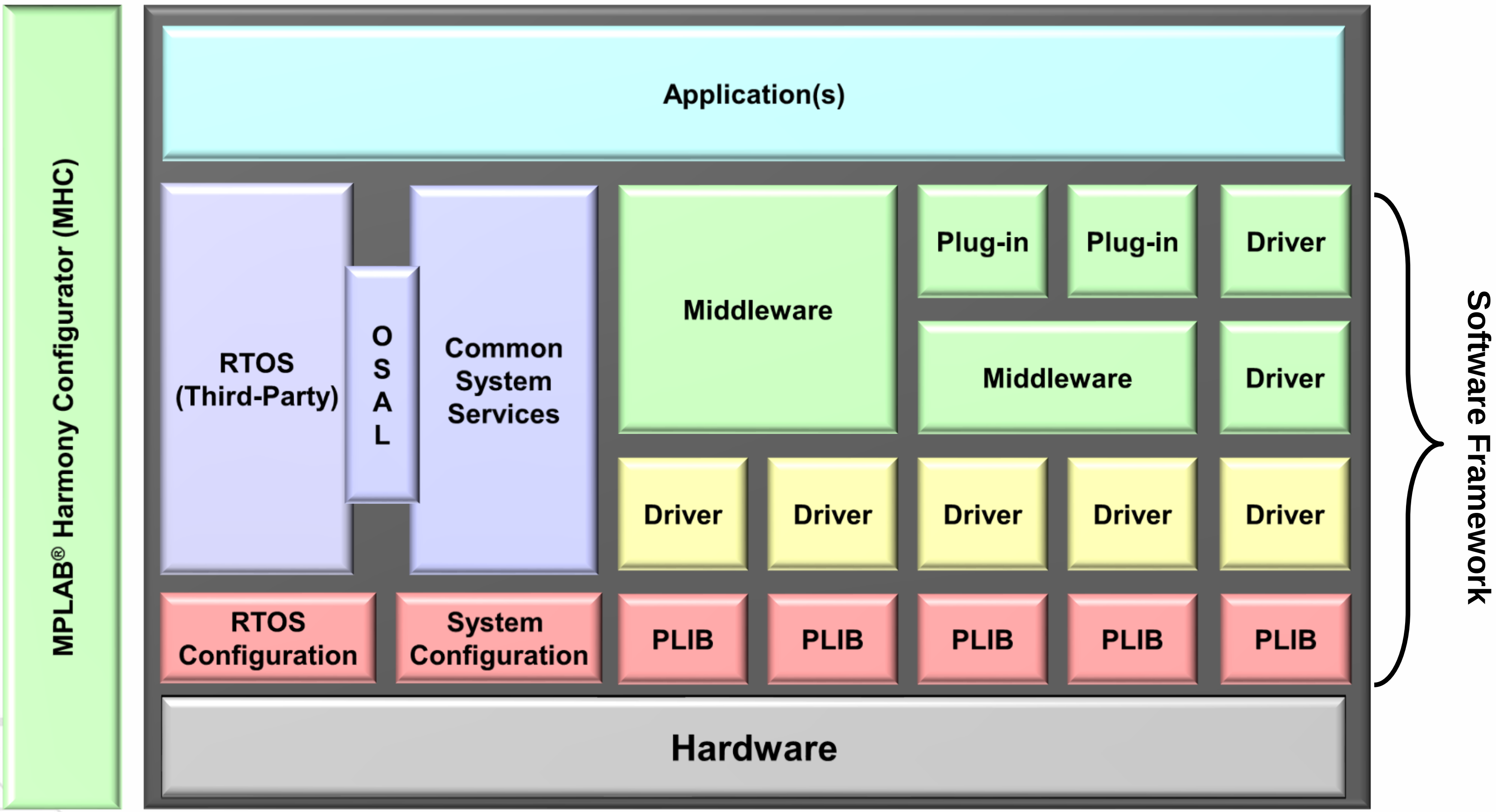
Lab2: Use Harmony Driver in Multi Instance Configuration

Using MPLAB Harmony in an RTOS environment

Lab3: Add RTOS to the Application

Summary

MPLAB® Harmony Architecture



MPLAB® Harmony Key Concepts

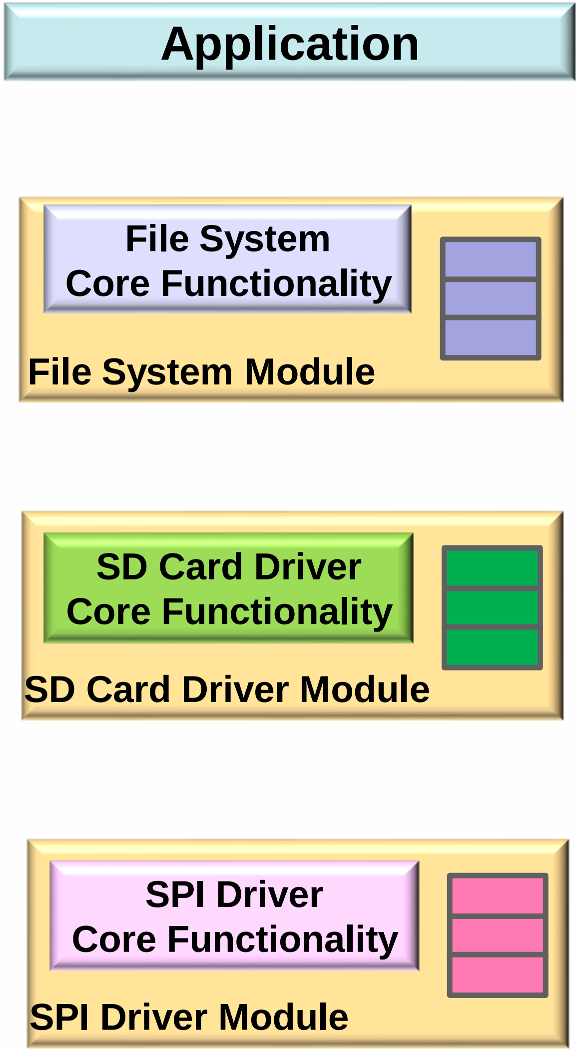
Modularity

Compatibility - State Machine Model

Modularity

Modular software is…

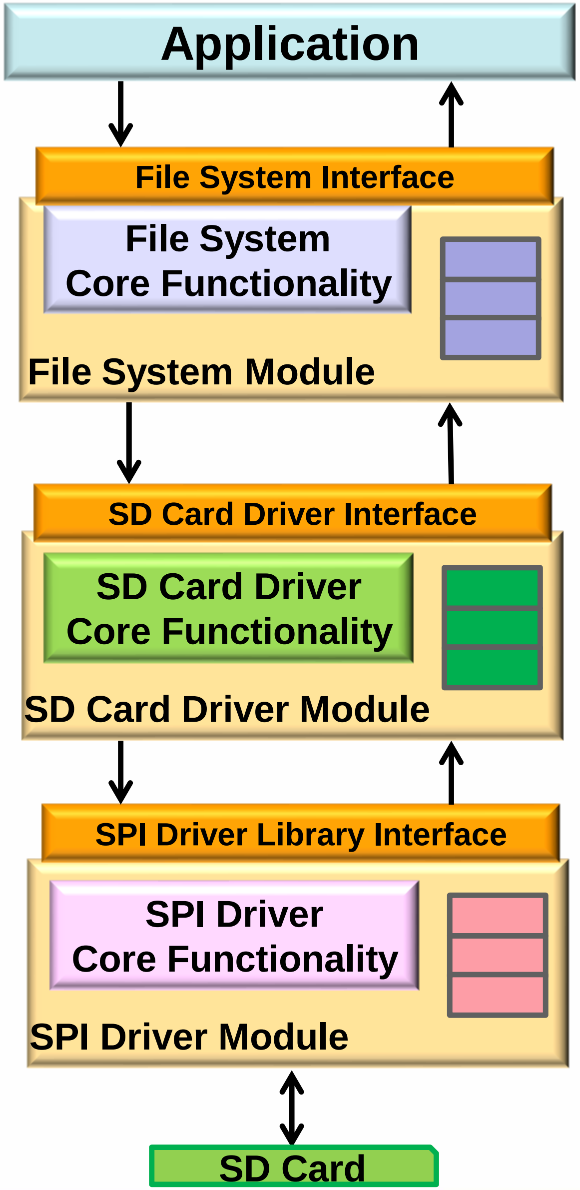
- **Highly Cohesive**
- Provides Interface functions and manages its own resources
- Loosely Coupled – Modules only interact with interface and does not know its internal implementation
- Prevents conflicts by protecting the shared resources



Modularity

Modular software is…

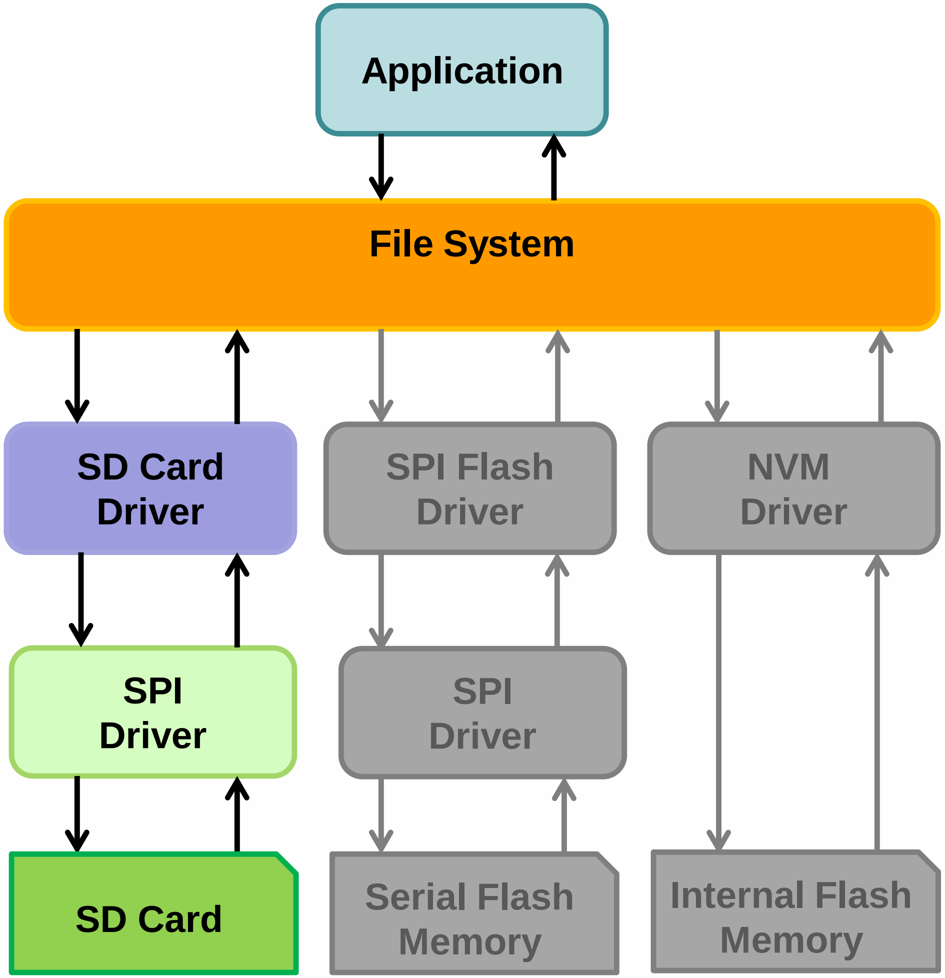
- Highly Cohesive
- **Provides Interface functions and manages its own resources**
- Loosely Coupled – Modules only interact with interface and does not know its internal implementation
- Prevents conflicts by protecting the shared resources



Modularity

Modular software is…

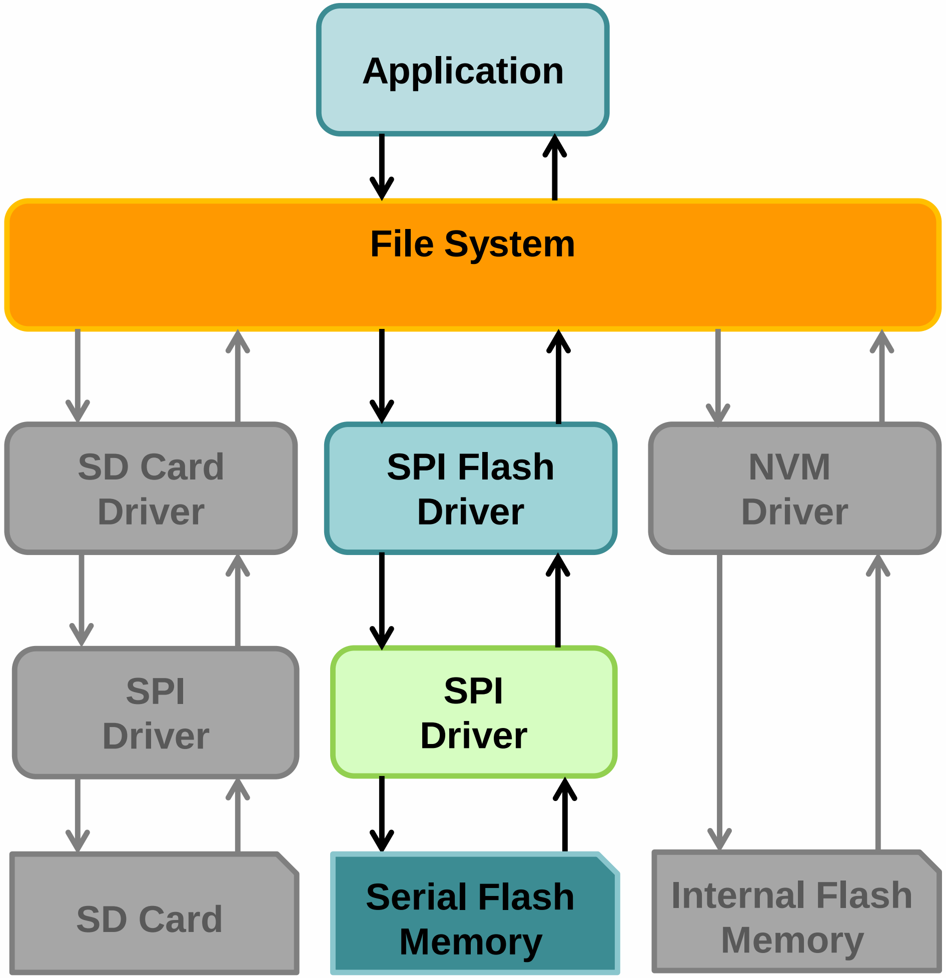
- Highly Cohesive
- Provides Interface functions and manages its own resources
- **Loosely Coupled – Modules only interact with interface and does not know its internal implementation**
- Prevents conflicts by protecting the shared resources



Modularity

Modular software is…

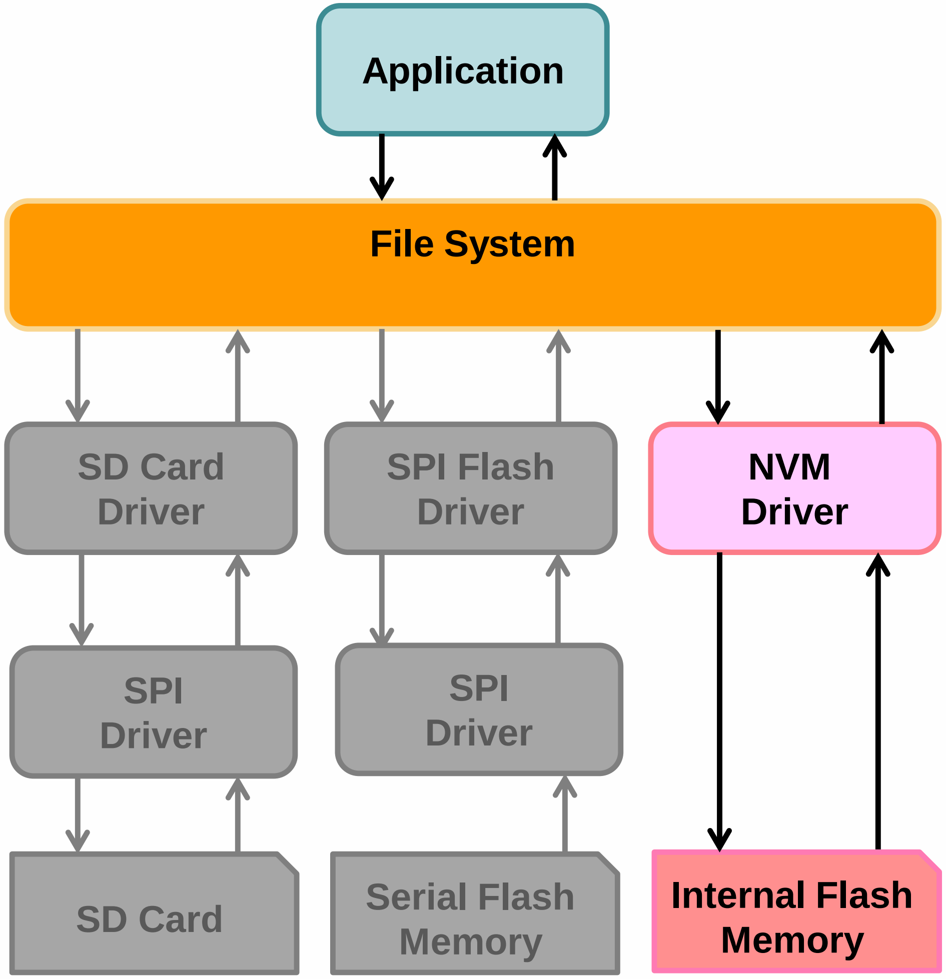
- Highly Cohesive
- Provides Interface functions and manages its own resources
- **Loosely Coupled – Modules only interact with interface and does not know its internal implementation**
- Prevents conflicts by protecting the shared resources



Modularity

Modular software is…

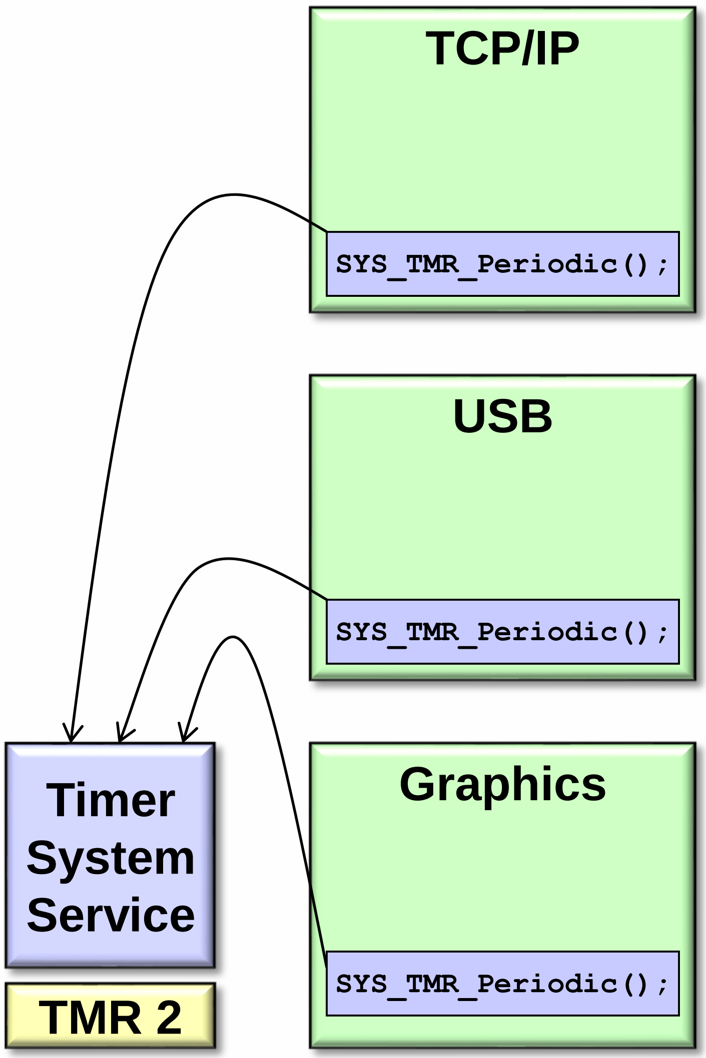
- Highly Cohesive
- Provides Interface functions and manages its own resources
- **Loosely Coupled – Modules only interact with interface and does not know its internal implementation**
- Prevents conflicts by protecting the shared resources



Modularity

Modular software is…

- Highly Cohesive
- Provides Interface functions and manages its own resources
- Loosely Coupled – Modules only interact with interface and does not know its internal implementation
- **Prevents conflicts by protecting the shared resources**

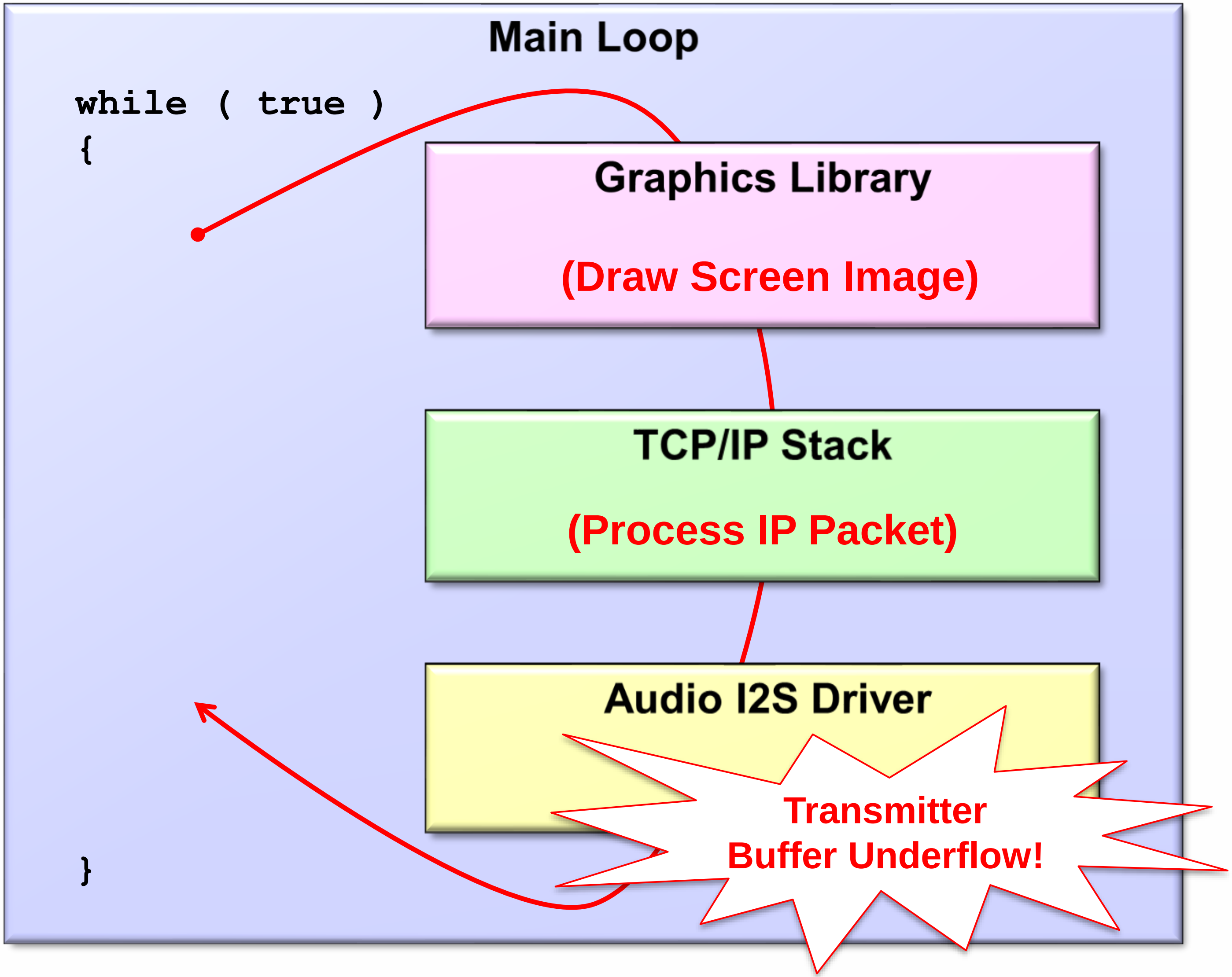


MPLAB® Harmony Key Concepts

Modularity

Compatibility - State Machine Model

Why State Machines?



Why State Machines?

Main Loop

```
while ( true )
{
```

GFX_Task_State = Task3

Graphics Library Sub Task 1

Graphics Library Sub Task 2

Graphics Library Sub Task n

TCPIP_Task_State = Task3

TCP/IP Stack Sub Task 1

TCP/IP Stack Sub Task 2

TCP/IP Stack Sub Task n

Audio_Driver_State = Task3

Audio I2S Driver Sub Task 1

Audio I2S Driver Sub Task 2

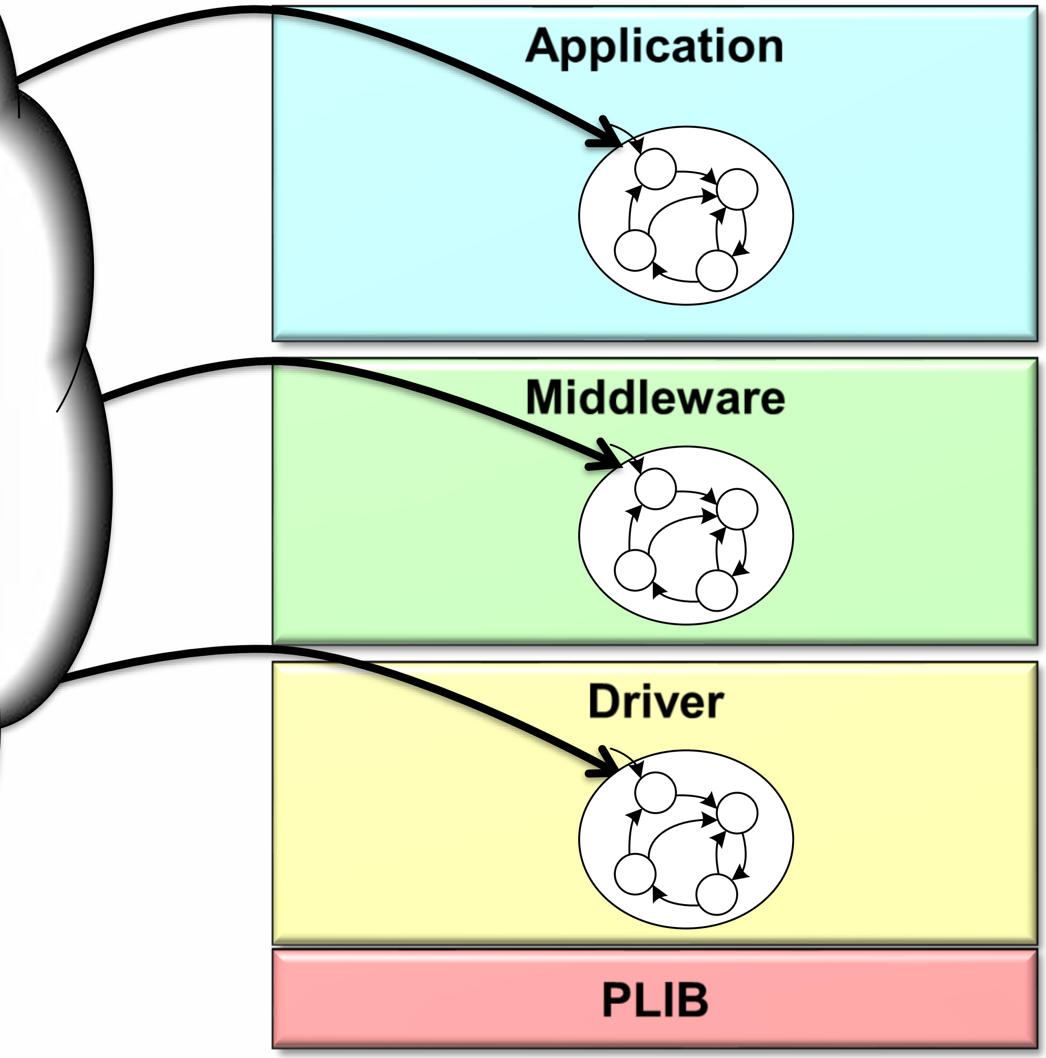
Audio I2S Driver Sub Task n

```
}
```

Compatible State Machines

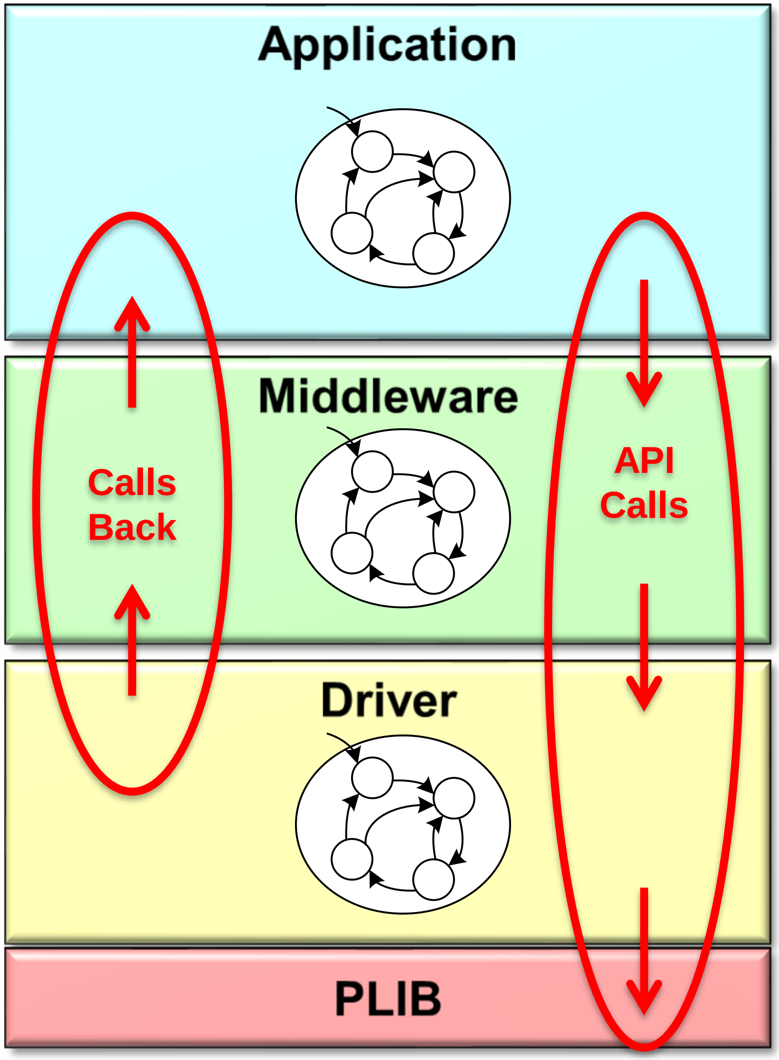
Effectively Independent

No Assumptions of Sequence or Timing Between modules



Compatible State Machines

Modules Interact With Each Other Only Through Interface Functions



Class Agenda

MPLAB® Harmony Key Concepts

Harmony Drivers and System Services

Lab1: Create a MPLAB Harmony Application using MPLAB Harmony Drivers and System Services.

Harmony Drivers Advanced Usage

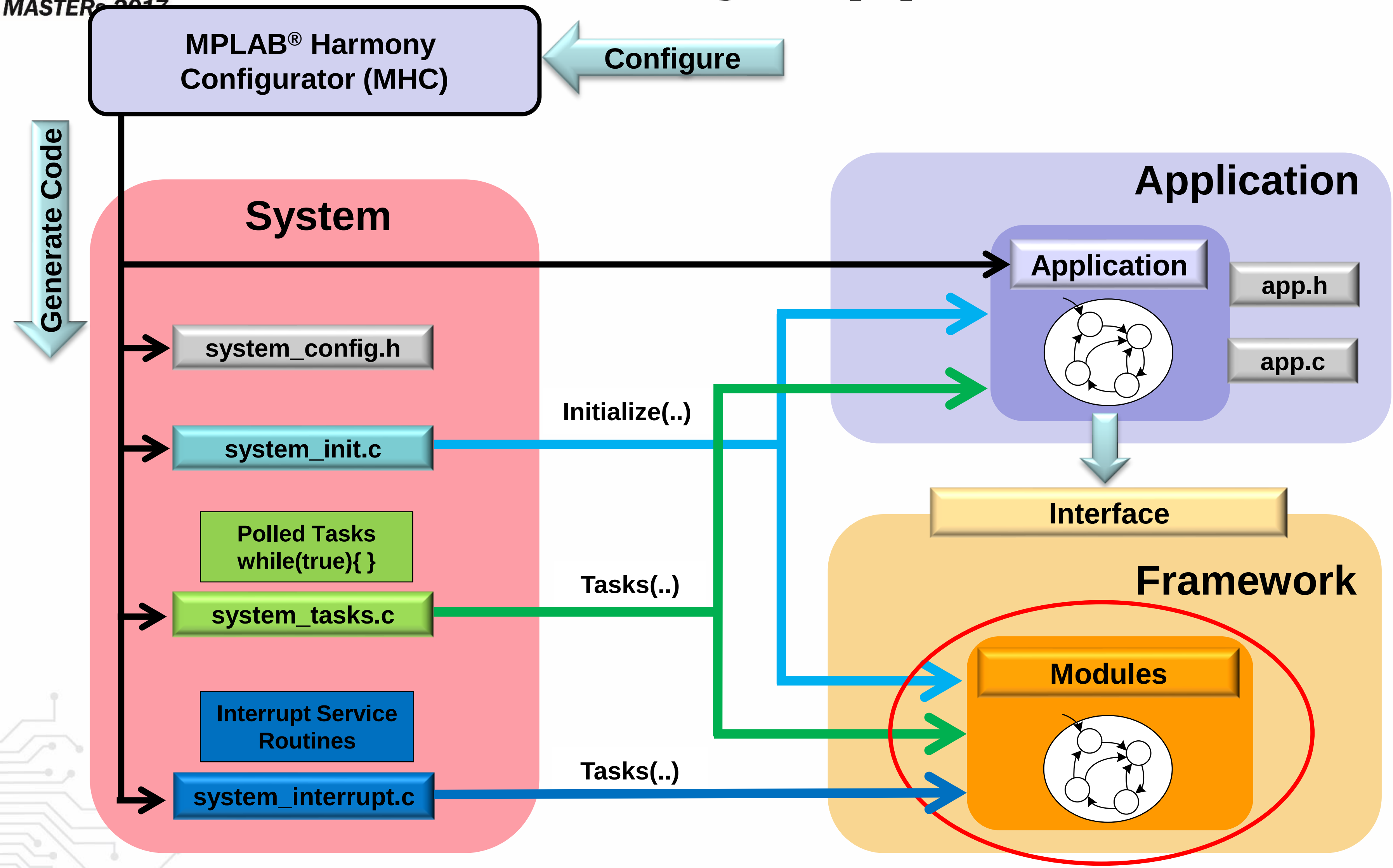
Lab2: Use Harmony Driver in Multi Instance Configuration

Using MPLAB Harmony in an RTOS environment

Lab3: Add RTOS to the Application

Summary

Harmony Application

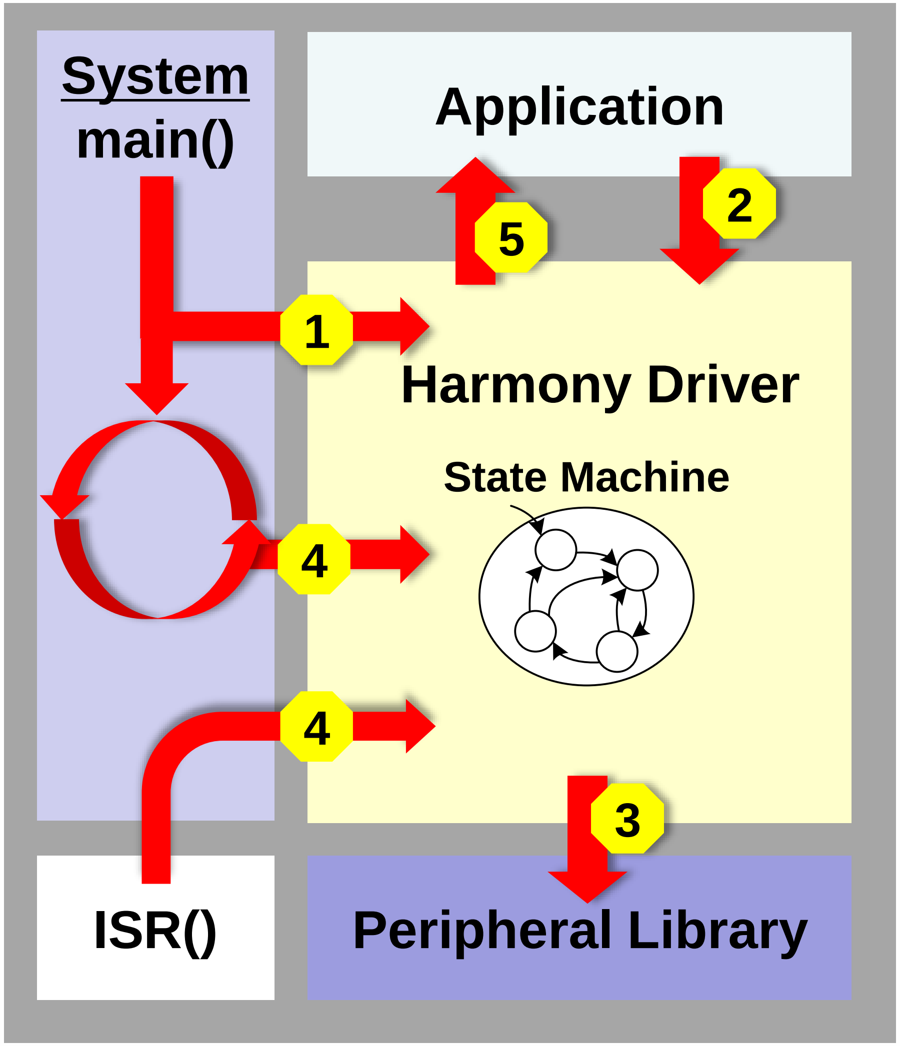


MPLAB® Harmony
Driver Example

1. System Initializes Driver
2. Application Calls Driver
3. Driver Starts Operation
4. Interrupt Occurs and Runs Driver State Machine

   Or

   System Runs the Driver State Machine, if in Polled Mode
5. Driver finishes Operation and Notifies the Application



Harmony Drivers and System Services

Polled vs Interrupt

Callbacks

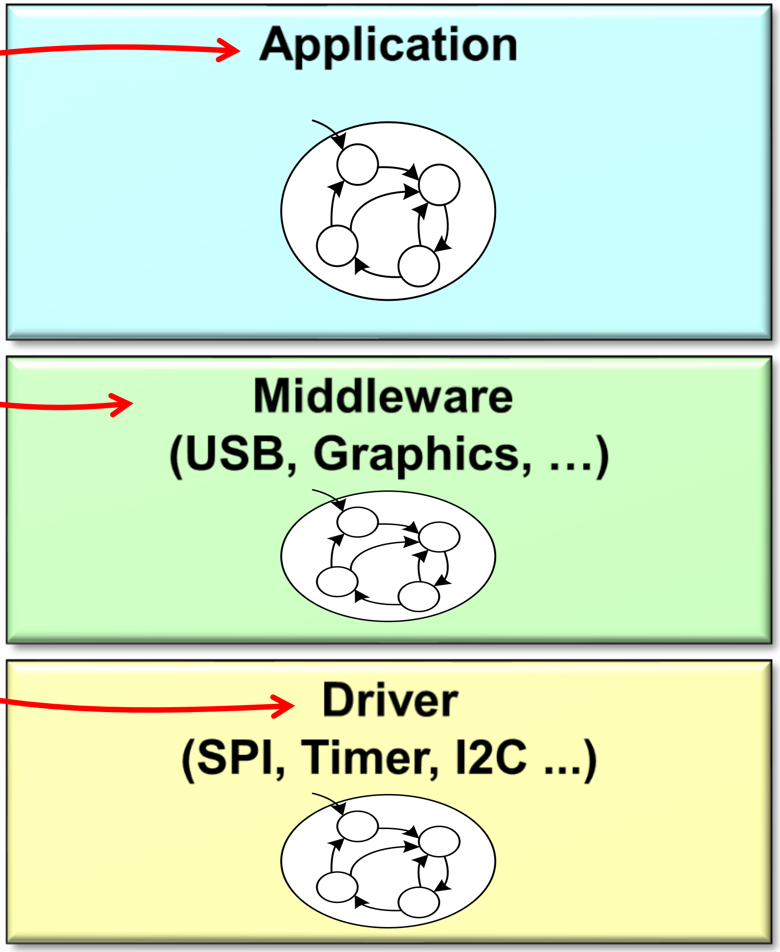
Configuring & Using Driver

Request Queuing and Execution

System Services

Super Loop – Polled

```
system_tasks.c

while(1)
{
    void SYS_Tasks()
    {
            APP_Tasks();

            USB_DEVICE_Tasks();
            GFX_Tasks();

            DRV_SPI_Tasks();
            DRV_TMR_Tasks();
            …
    }
}
```

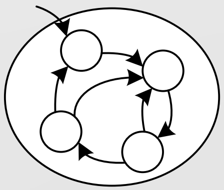


Interrupt Driven

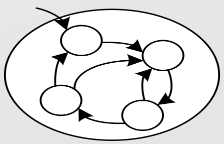
MICROCHIP MASTERs 2017

```
system_tasks.c

while(1)
{
    void SYS_Tasks()
    {
            APP_Tasks();

            USB_DEVICE_Tasks();
            GFX_Tasks();


            …
    }
}
```

Application



Middleware (USB, Graphics, …)

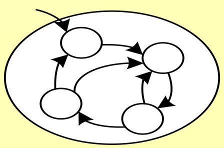


System_interrupt.c

void __ISR(_SPI2_TX_VECTOR, ipl3AUTO) _IntHandlerSPITxInstance0(void)
{
DRV_SPI_Tasks();
}

void __ISR(_TIMER_1_VECTOR, ipl1AUTO) IntHandlerDrvTmrInstance0(void)
{
DRV_TMR_Tasks();
}

Driver (SPI, Timer, I2C …)



Harmony Drivers and System Services

Polled vs Interrupt

Callbacks

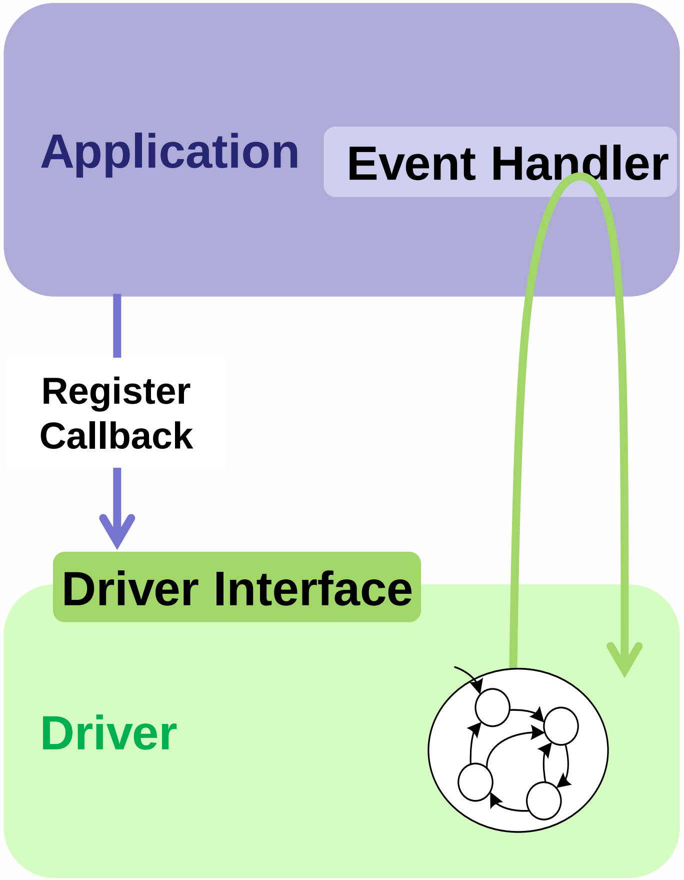
Configuring & Using Driver

Request Queuing and Execution

System Services

Callbacks

- Application functions called by Driver
- Allows applications to be notified, eliminates status polling
- Dynamically registered by the Application
- Driver provides APIs to register Callbacks



MICROCHIP
MASTERs 2017

Callbacks In Polled Mode

Callback executed in
the context of SYS_Tasks

```
void AppAlarmHandler(…)
{
    timerExpired = true;
}
```

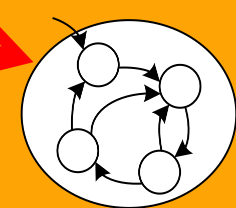
```
void APP_Task()
{
    DRV_TMR_AlarmRegister (tmrHandle,
     AppAlarmHandler);
}
```

Call
Event Handler

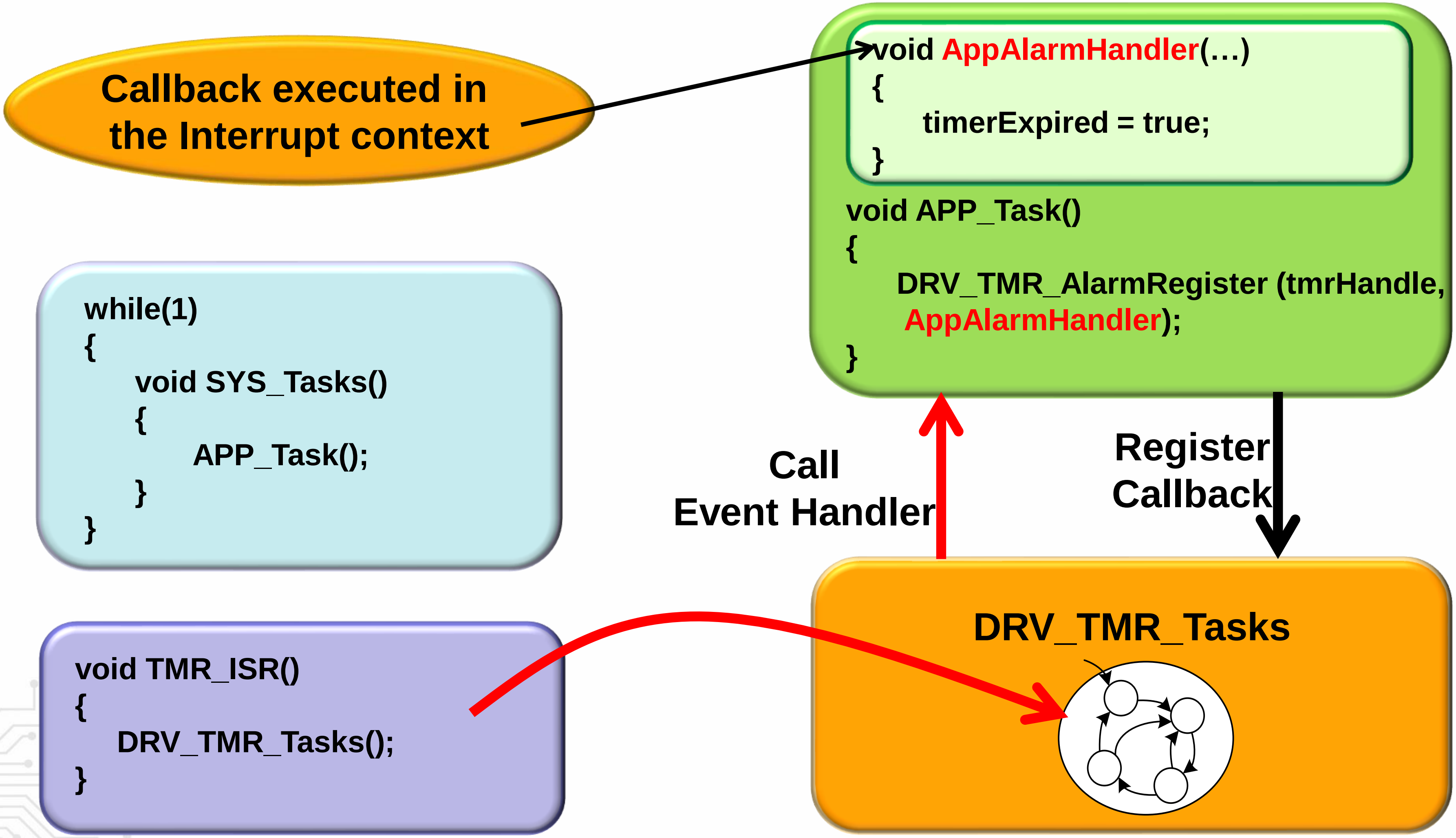
Register
Callback

```
while(1)
{
    void SYS_Tasks()
    {
        DRV_TMR_Tasks();
        APP_Task();
    }
}
```

DRV_TMR_Tasks



Callbacks In Interrupt Mode



Callbacks In Interrupt Mode

- **Must be treated like an ISR**
- **Must be short**
- **Must not call application functions that are not interrupt safe**
- **Must not call *other* driver's interface functions**
- **Use volatile keyword**

Harmony Drivers and System Services

Polled vs Interrupt

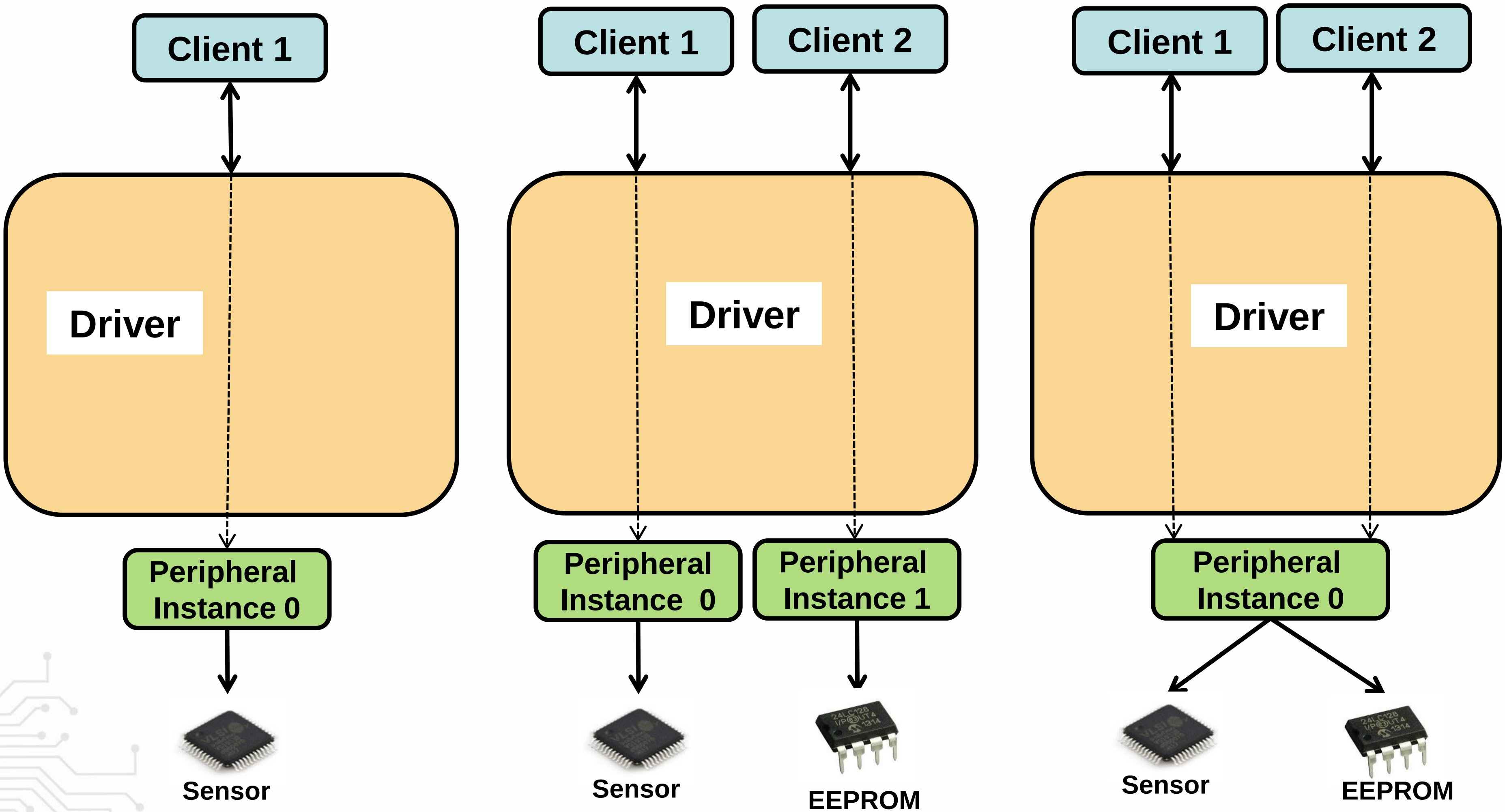
Callbacks

Configuring & Using Driver

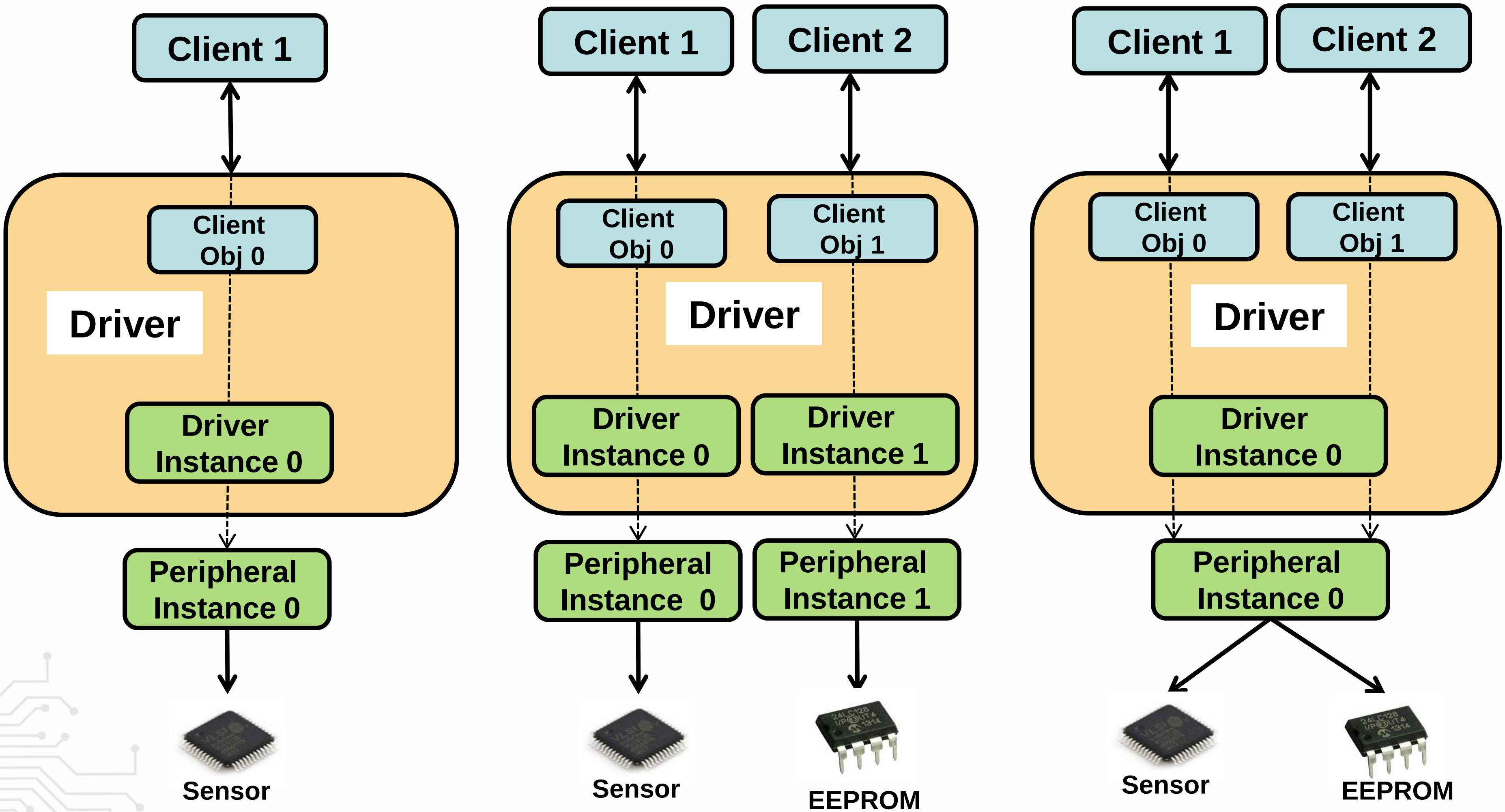
Request Queuing and Execution

System Services

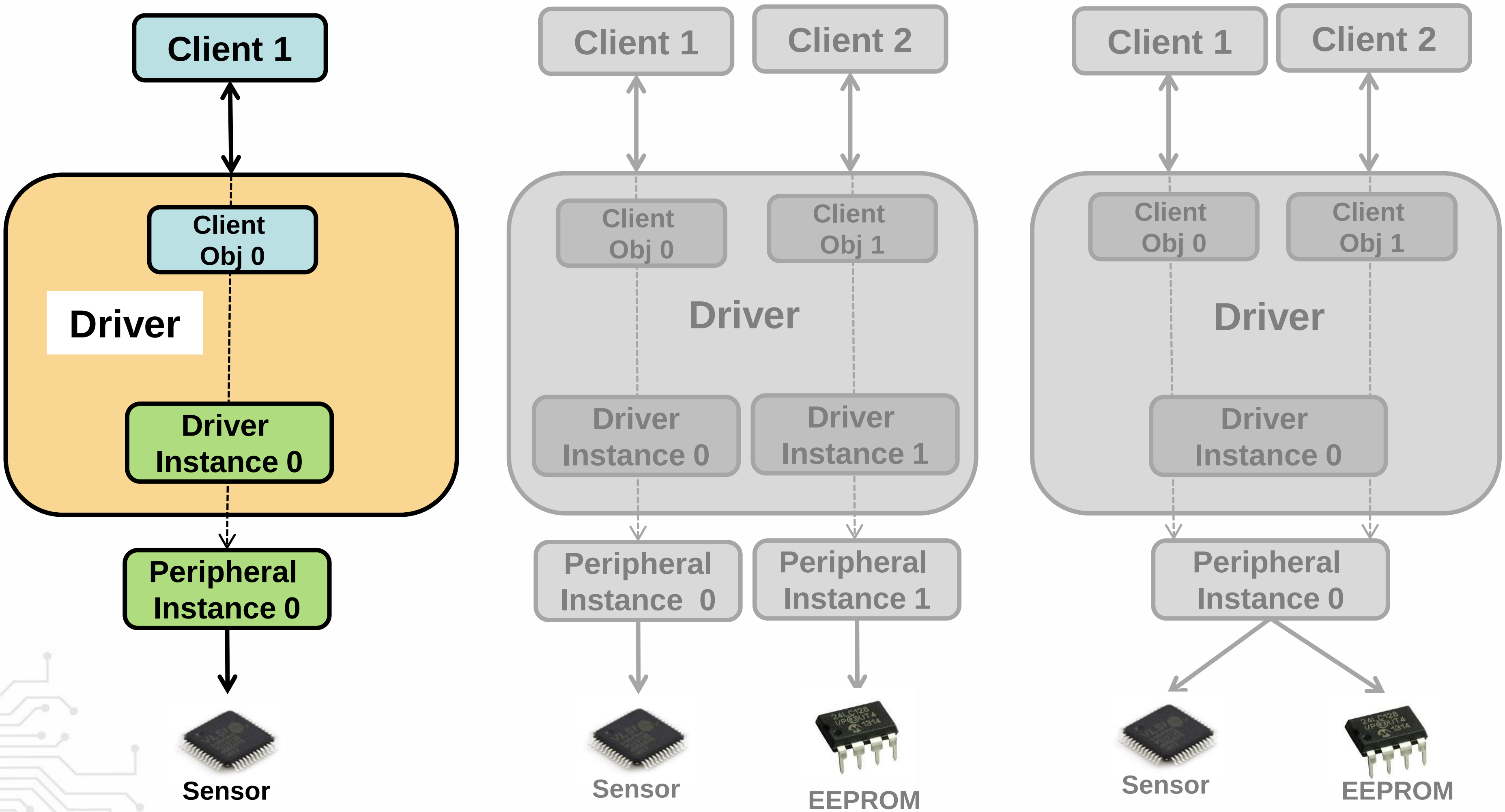
Driver Usage Models



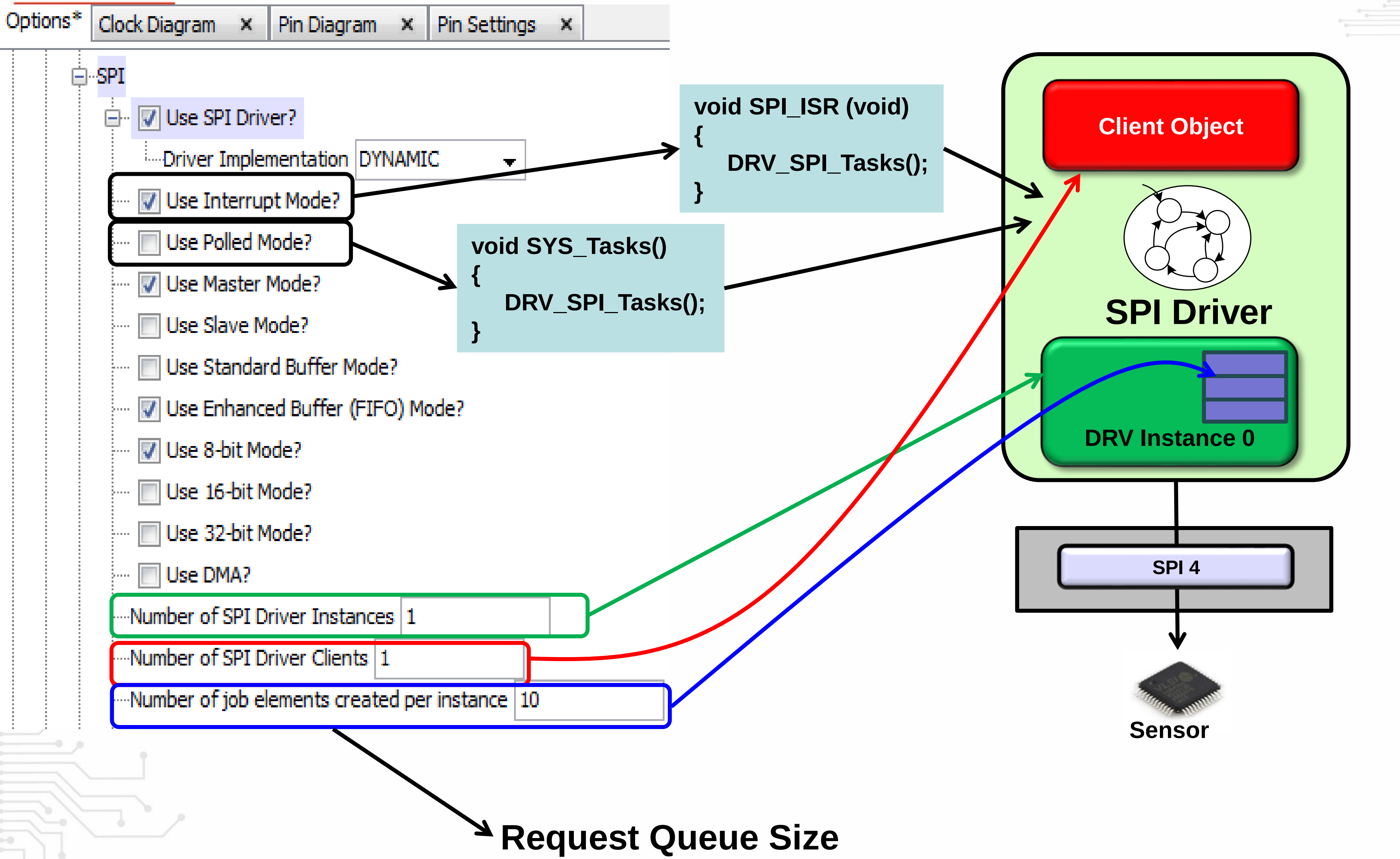
Driver Usage Models



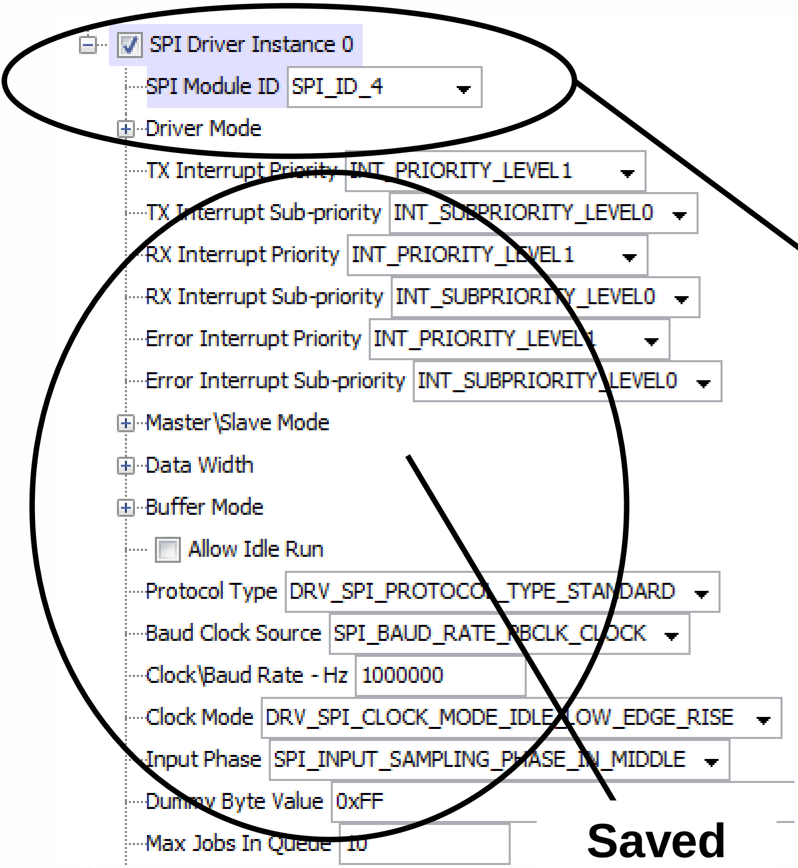
Driver Usage Models



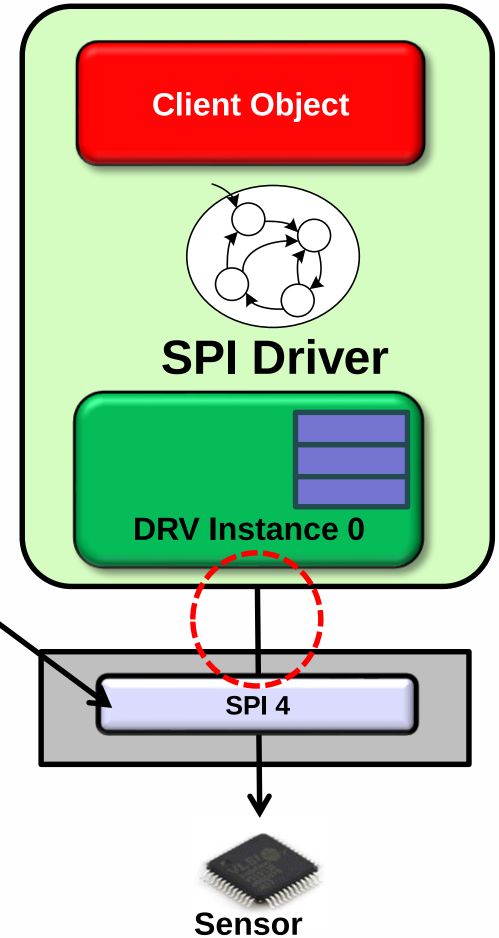
Driver Configuration in MHC



Driver Instance Configuration in MHC




```
void Sys_Initialize( void )
{
    …..
    sysObj.spiObjectIdx0 = DRV_SPI_Initialize(
        DRV_SPI_INDEX_0,
        (const SYS_MODULE_INIT  * const)&drvSpi0InitData
   );
   …..
}
```

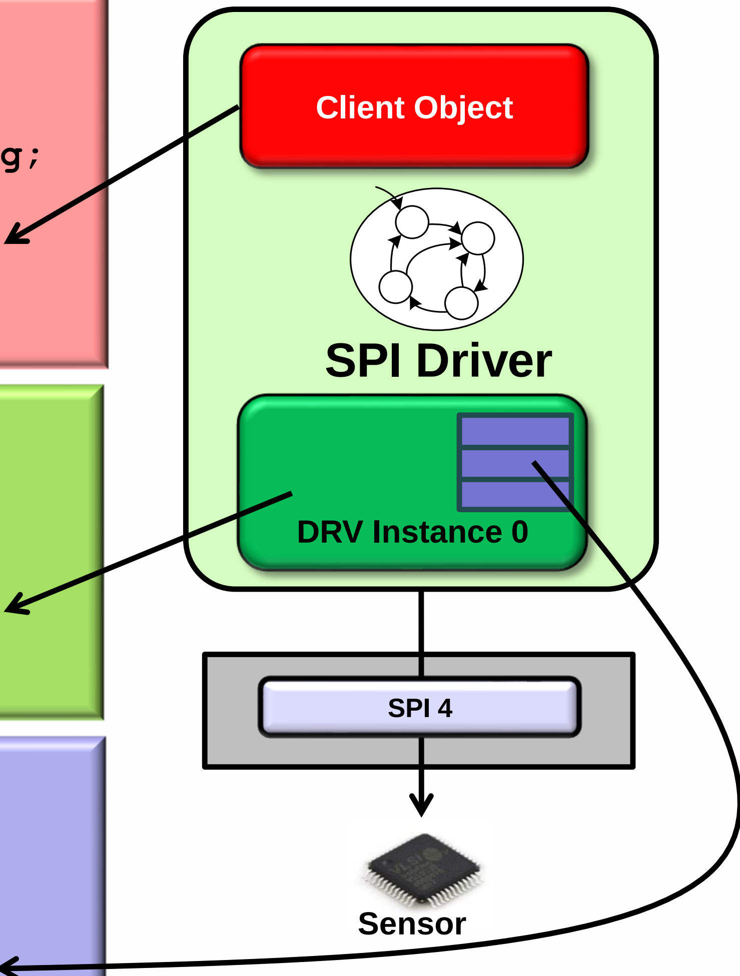


Driver Internal Data structures

```
typedef struct _DRV_SPI_CLIENT_OBJECT
{
    struct DRV_SPI_DRIVER_OBJECT*  driverObject;
    DRV_SPI_BUFFER_EVENT_HANDLER   operationStarting;
    DRV_SPI_BUFFER_EVENT_HANDLER   operationEnded;
    uint32_t                       baudRate;
    .....
}DRV_SPI_CLIENT_OBJECT;
```

```
struct DRV_SPI_DRIVER_OBJECT
{
    SPI_MODULE_ID                  spiId;
    DRV_SPI_MODE                   spiMode;
    SPI_COMMUNICATION_WIDTH        commWidth;
    ....
};
```

```
typedef struct _DRV_SPI_JOB_OBJECT
{
    uint8_t                        *txBuffer;
    uint8_t                        *rxBuffer;
    size_t                         dataLeftToTx;
    size_t                         dataLeftToRx;
    DRV_SPI_BUFFER_EVENT_HANDLER   completeCB;
    .....
}DRV_SPI_JOB_OBJECT;
```



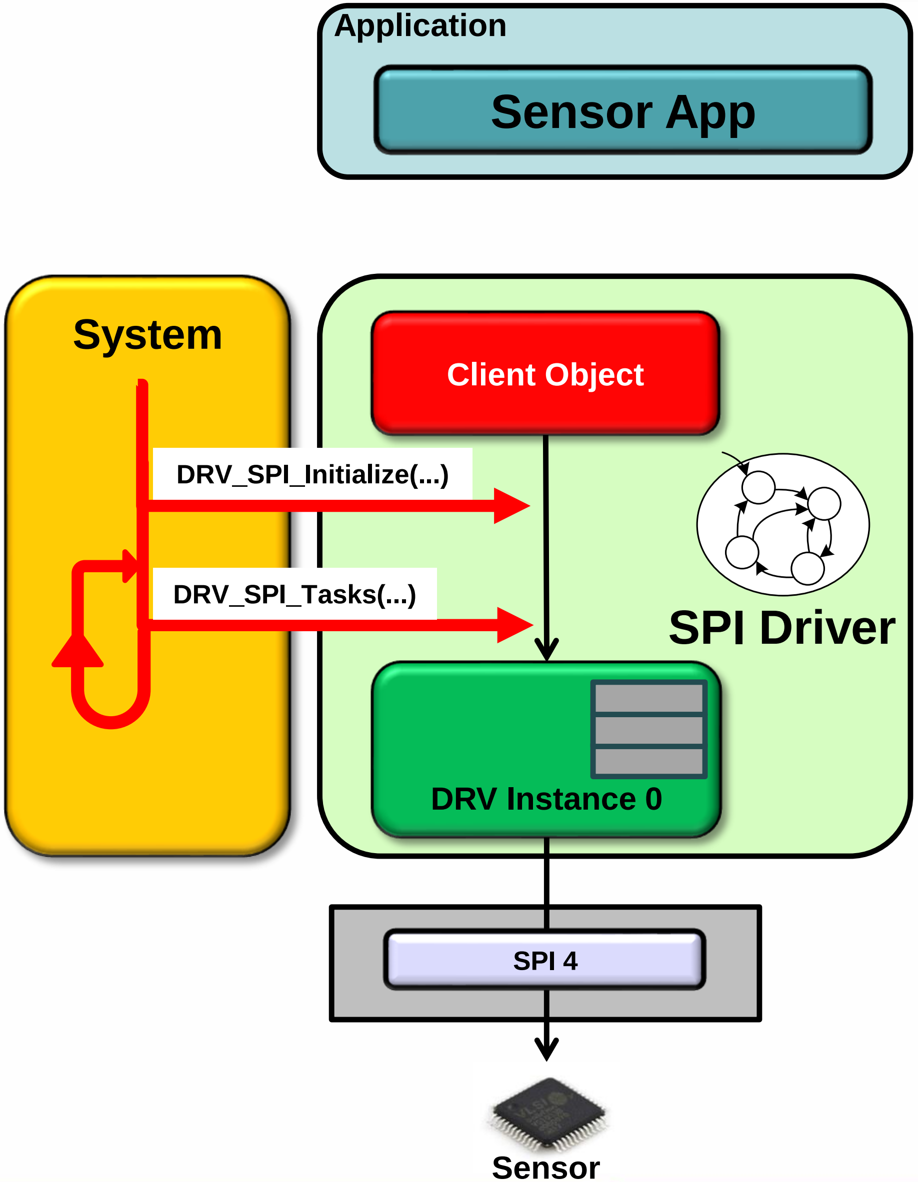
System Interface – Initializing and Running the SPI Driver

- **System Interface**

```
void Sys_Initialize( void )
{
    ....
    sysObj.spiObjectIdx0 = DRV_SPI_Initialize(
        DRV_SPI_INDEX_0,
        (const SYS_MODULE_INIT  * const)&drvSpi0InitData
   );
    ....
}
```

```
// Polled Mode

void SYS_Tasks( void )
{
    ....
        DRV_SPI_Tasks(sysObj. spiObjectIdx0);
    ....
}
```

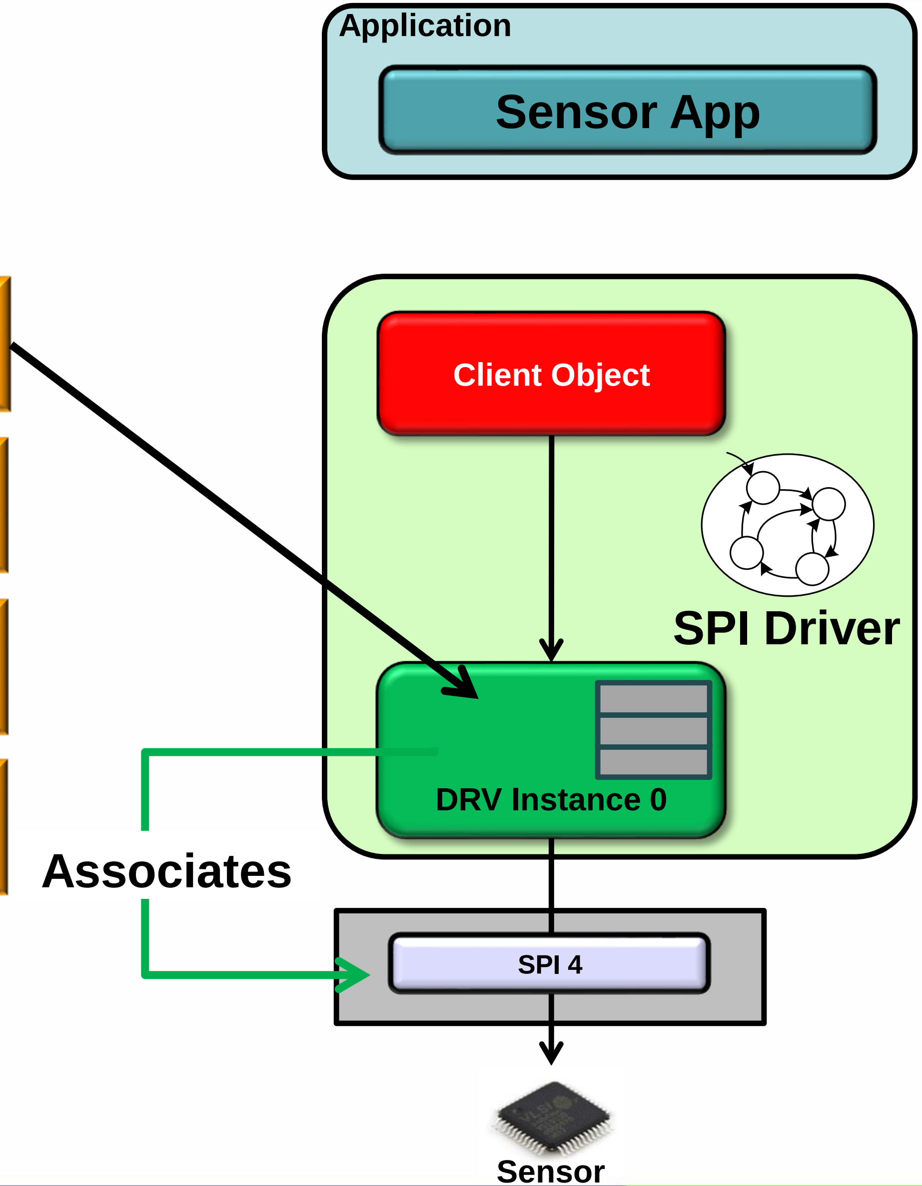
```
// Interrupt Mode

void SPI_ISR( void )
{
        DRV_SPI_Tasks(sysObj.spiObjectIdx0);
}
```



System Interface – Initializing and Running the SPI Driver

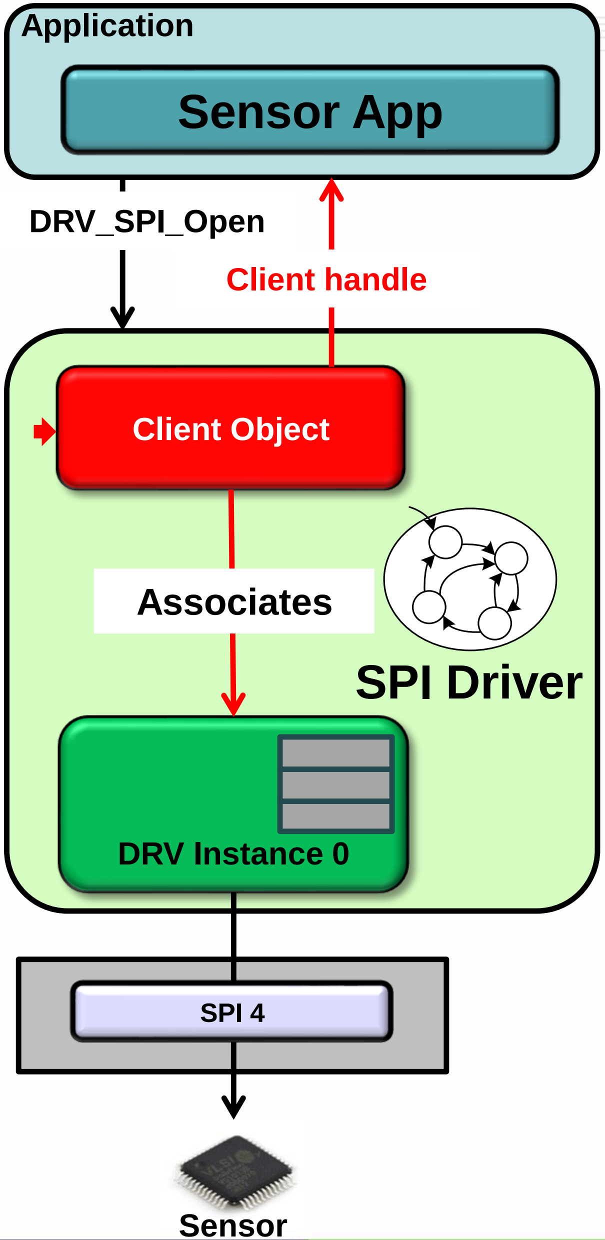
- What it does

- Initializes the SPI Driver Instance
- Runs the SPI Driver state machine
- Associates SPI Driver Instance with the Hardware SPI Instance
- Called as part of the system initialization



Client APIs – Opening the SPI Driver

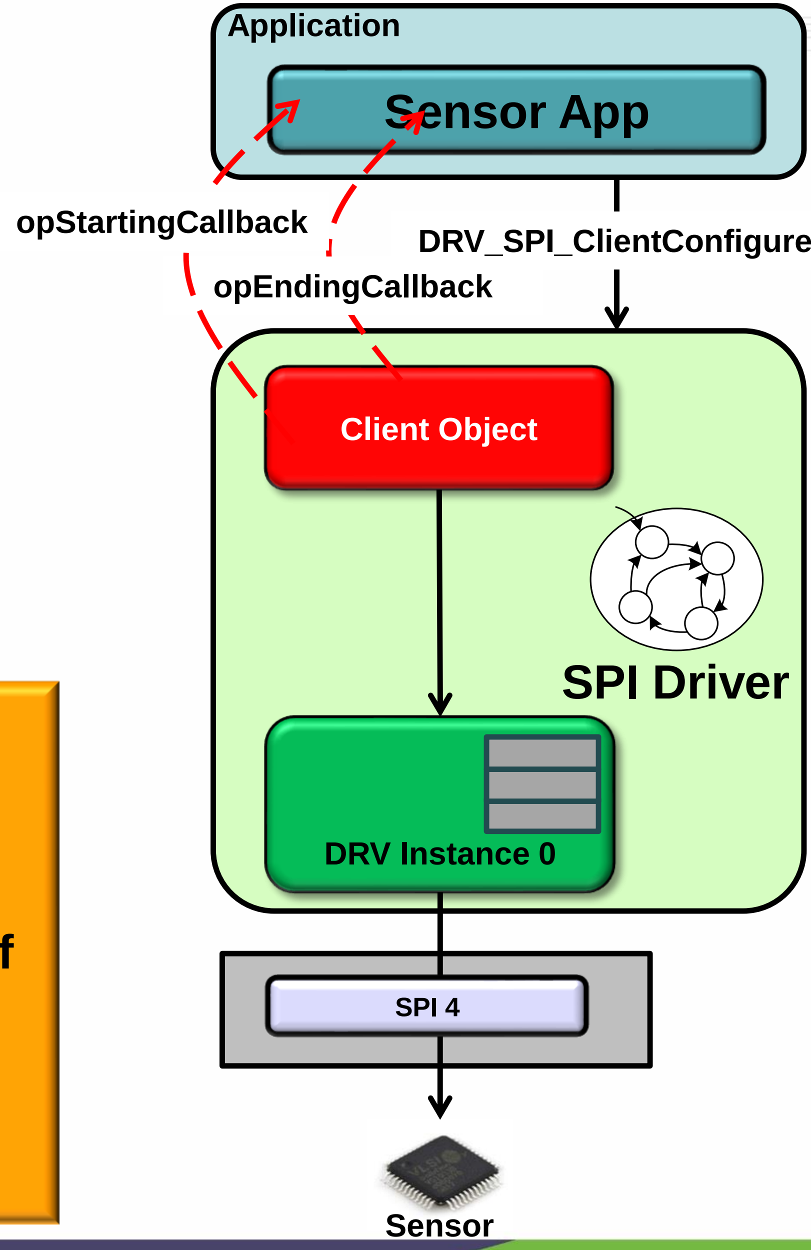
```
sensorClientHandle = DRV_SPI_Open (
        DRV_SPI_INDEX_0,
        DRV_IO_INTENT_READWRITE
    );
```



- What it does
  - Associates a client with the SPI Driver Instance
  - Returns the client handle to the application

Client APIs – Setting up the Driver

```
sensorTask.clientCfg.operationStarting = sensorOpStarting;
sensorTask.clientCfg.operationEnding  = sensorOpEnding;

DRV_SPI_ClientConfigure(
        sensorClientHandle,
        &sensorTask.clientCfg
);
```



- **What it does**

Configures the Client -

- **Registers the application callback functions that will be called by the driver to allow selection/de-selection of SPI Slave**
- **These callbacks will be called by the driver for each SPI request**

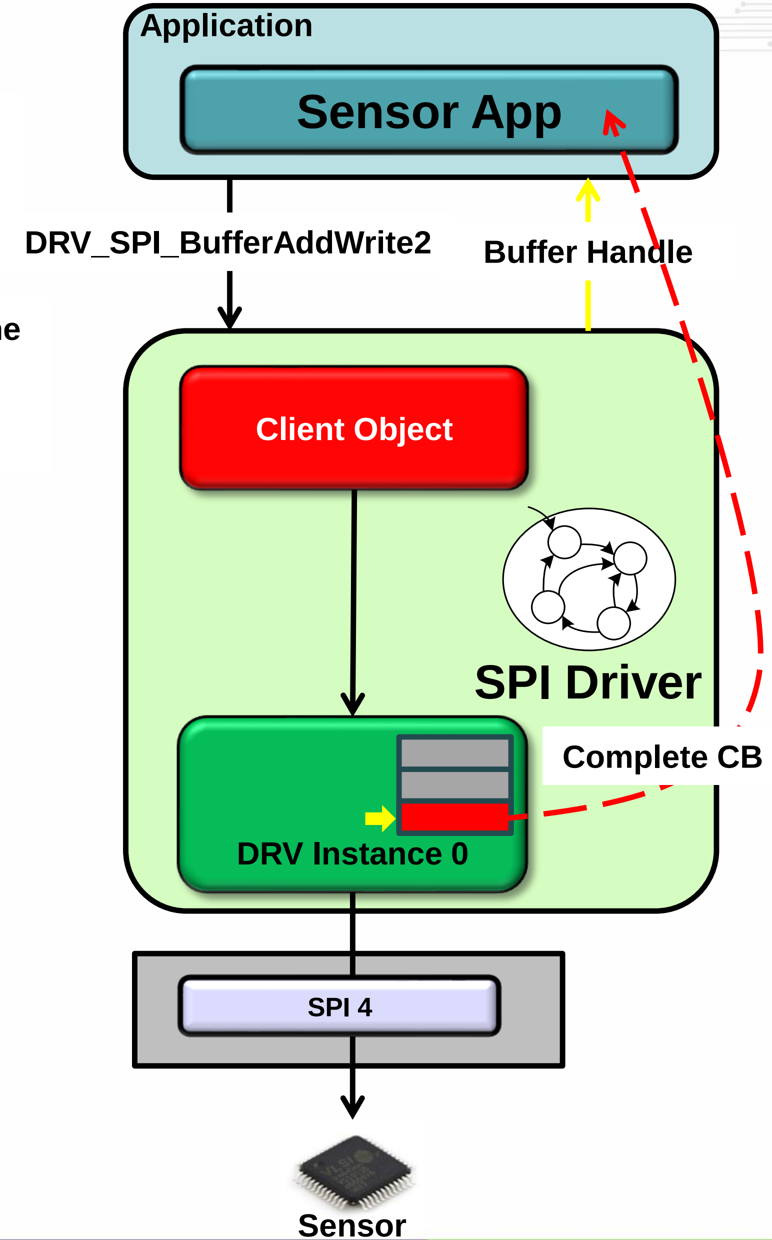
Client APIs - Submitting Requests

```
DRV_SPI_BufferAddWrite2(
        sensorClientHandle,
        (void*)sensorTask.wrBuffer,
        sensorTask.nBytes,
        sensorBufferEventHandler,
        (void*)context,
        &sensorTask.bufferHandle
    );
```

Application callback function. Called by the driver when the request is completed.

context – unused by the driver. Passed back by the driver to the Buffer Event Handler

- What it does
  - Adds the request to the instance specific buffer queue
  - Returns the buffer handle to the application
  - Buffer handle can be used to poll the status of the request by calling *DRV_SPI_BufferStatus(sensorTask.BufferHandle)*
  - Registers an application callback that will be called when the request is completed



Client APIs - Submitting Requests

```
uint8_t request1Status = false;
uint8_t request2Status = false;

case SENSOR_SUBMIT_REQ:
        DRV_SPI_BufferAddWrite2(
                    sensorClientHandle,
                    (void*)sensorTask.wrBuffer1,
                    sensorTask.nBytes1,
                    sensorBufferEventHandler,
                    (void*)&request1Status,
                    &sensorTask.bufferHandle1
                 );

        DRV_SPI_BufferAddWrite2(
                    sensorClientHandle,
                    (void*)sensorTask.wrBuffer2,
                    sensorTask.nBytes2,
                    sensorBufferEventHandler,
                    (void*)&request2Status,
                    &sensorTask.bufferHandle2
                 );
        state = SENSOR_CHECK_REQ_STATUS;
        break;

case SENSOR_CHECK_REQ_STATUS:
        if (request1Status == true && request2Status == true)
        {
                /* move to the next state  */
        }
        break;
```

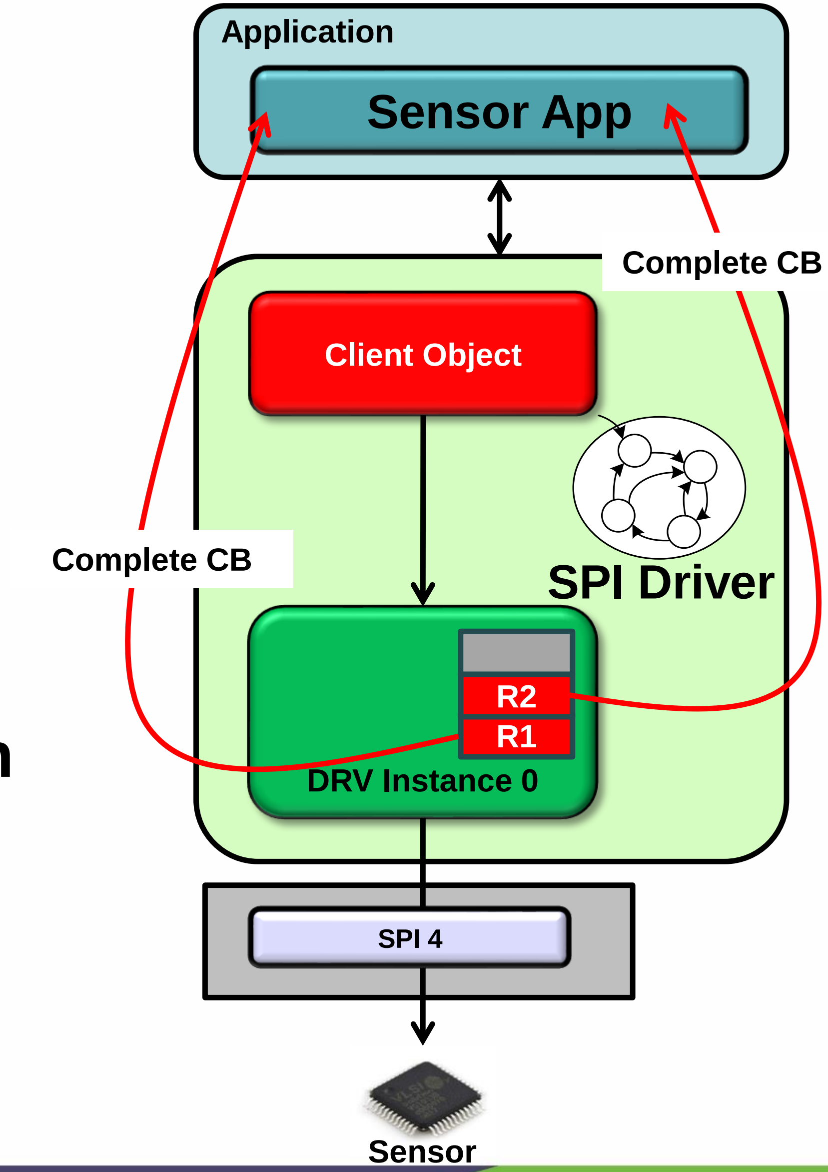
```
static void sensorBufferEventHandler(
    DRV_SPI_BUFFER_EVENT event,
    DRV_SPI_BUFFER_HANDLE bufferHandle,
    void * context
)
{
    switch(event)
    {
        case DRV_SPI_BUFFER_EVENT_COMPLETE:
                if (context)
                {
                        *((uint8_t*)context) = true;
                }
                break;
        case DRV_SPI_BUFFER_EVENT_ERROR:
                if (context)
                {
                        *((uint8_t*)context) = false;
                }
                break;
    }
}
```

Harmony Drivers and System Services

- Polled vs Interrupt
- Callbacks
- Configuring & Using Driver
- **Request Queuing and Execution**
- System Services

Request Queuing

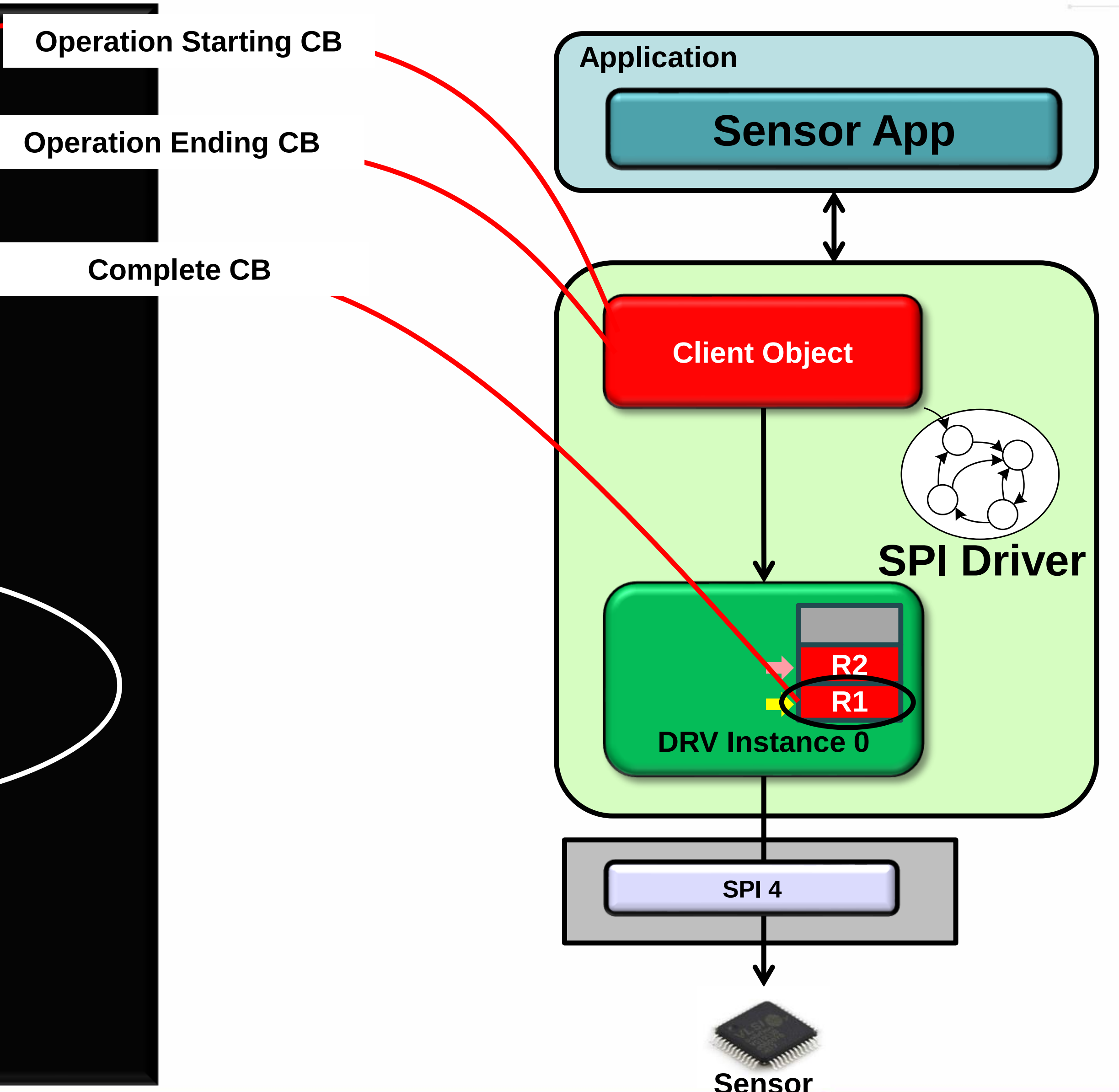
- Queue multiple requests
- Can queue a request before previous request completes
- Application notified through a callback or can poll for status
- A separate buffer queue for each driver instance



SPI Request Execution

MICROCHIP MASTERs 2017

```
void sensorOpStarting(…)
{
   SENSOR_CS_EN();
}
void sensorOpEnding(…)
{
    SENSOR_CS_DIS();
}
void sensorBufferEventHandler(…)
{
    sensorReqCompleted = true;
}

void SensorApp( void )
{
         sensorClientHandle = DRV_SPI_Open(
         DRV_SPI_INDEX_0,
         DRV_IO_INTENT_READWRITE
    );
     …..
         DRV_SPI_BufferAddWrite2(
              sensorClientHandle,
              (void*)sensorTask.wrBuffer1,
              sensorTask.nBytes1,
              sensorBufferEventHandler,
              (void*)NULL,
              &sensorBufferHandle1
     );
        DRV_SPI_BufferAddWrite2(
              sensorClientHandle,
              (void*)sensorTask.wrBuffer2,
              sensorTask.nBytes2,
              sensorBufferEventHandler,
              (void*)NULL,
              &sensorBufferHandle2
     };
```



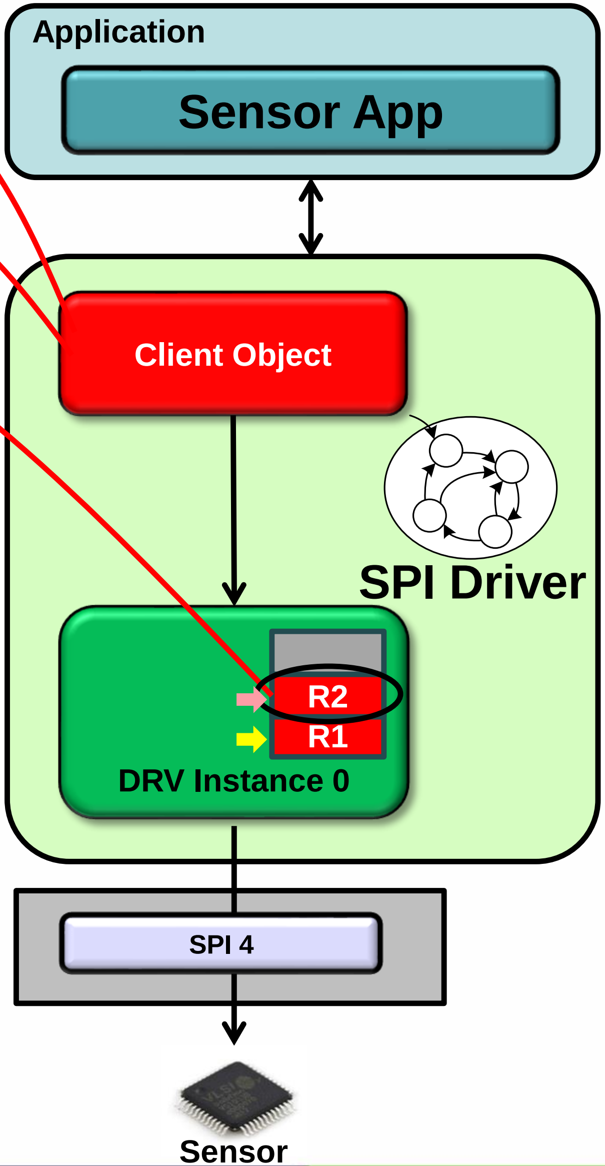
SPI Request Execution

```
void sensorOpStarting(…)
{
   SENSOR_CS_EN();
}
void sensorOpEnding(…)
{
    SENSOR_CS_DIS();
}
void sensorBufferEventHandler(…)
{
    sensorReqCompleted = true;
}

void SensorApp( void )
{
        sensorClientHandle = DRV_SPI_Open(
        DRV_SPI_INDEX_0,
        DRV_IO_INTENT_READWRITE
    );
    ……
        DRV_SPI_BufferAddWrite2(
            sensorClientHandle,
            (void*)sensorTask.wrBuffer1,
            sensorTask.nBytes1,
            sensorBufferEventHandler,
            (void*)NULL
            &sensorBufferHandle1
    );
       DRV_SPI_BufferAddWrite2(
            sensorClientHandle,
            (void*)sensorTask.wrBuffer2,
            sensorTask.nBytes2,
            sensorBufferEventHandler,
            (void*)NULL,
            &sensorBufferHandle2
    };
```

Operation Starting CB

Operation Ending CB

Complete CB



Summary: SPI Driver Client APIs

```
void SensorApp ( void )
{
        DRV_SPI_Open (…);


        DRV_SPI_ClientConfigure (…);


        DRV_SPI_BufferAddWrite2 (….);

}
```

```
void sensorOpStarting(…)
{
    ….
}

void sensorOpEnding(…)
{
    ….
}

void sensorBufferEventHandler(…)
{
    ….
}
```

SPI Driver APIs

DRV_SPI_Open (…);

DRV_SPI_ClientConfigure (…);

DRV_SPI_Close (…);

DRV_SPI_BufferAddWrite2 (….);

DRV_SPI_BufferAddRead2 (….);

DRV_SPI_BufferAddWriteRead2 (….);

DRV_SPI_BufferStatus (…);

Harmony Drivers and System Services

Polled vs Interrupt

Callbacks

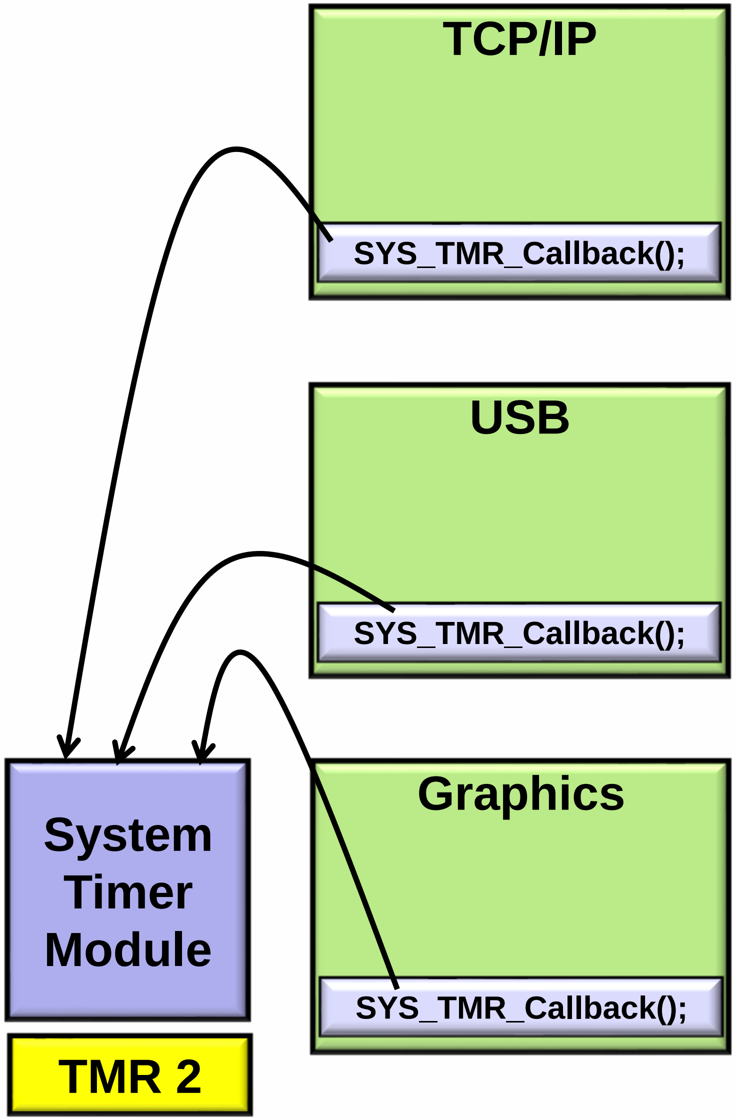
Configuring & Using Driver

Request Queuing and Execution

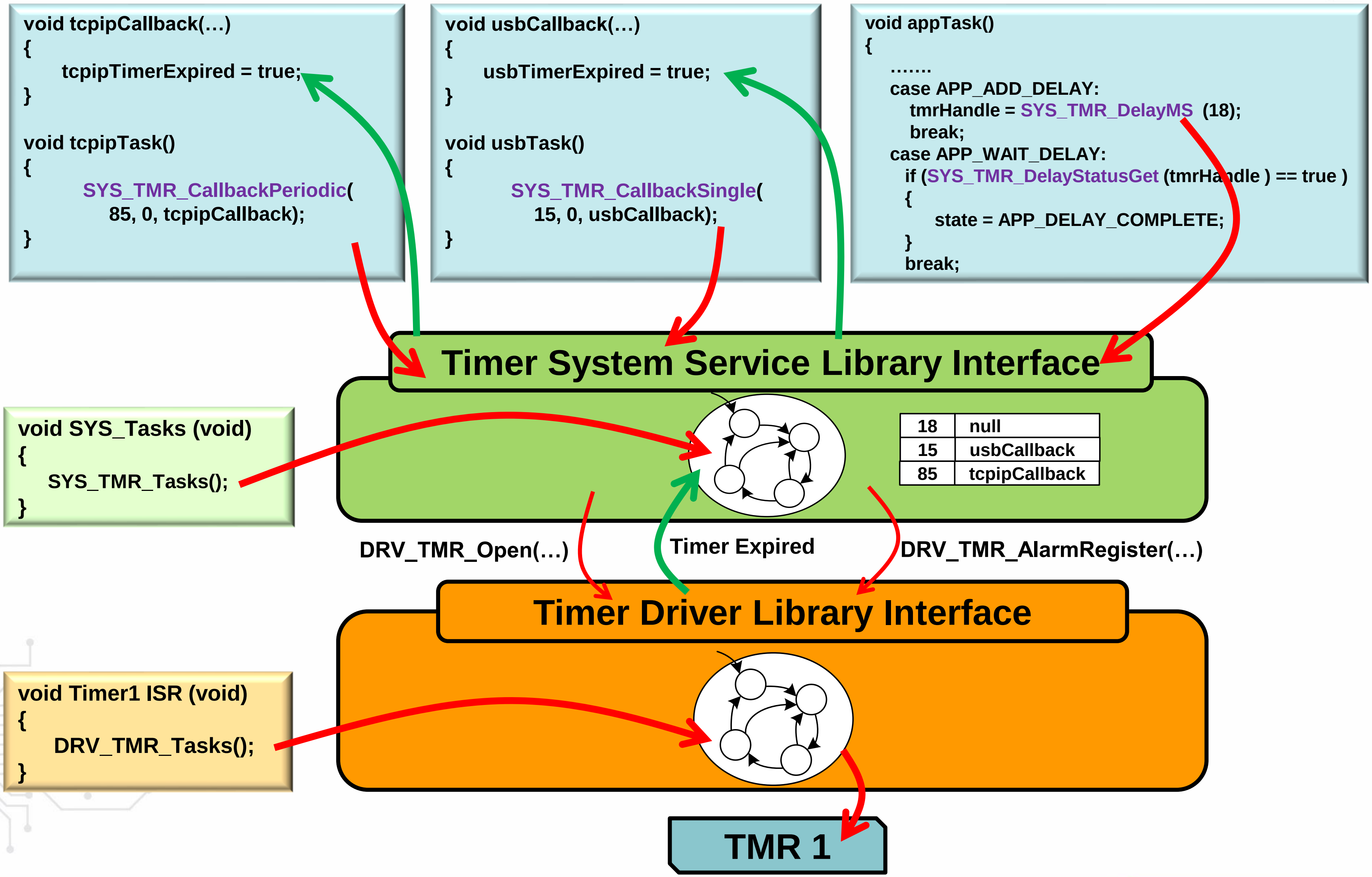
System Services

System Services

- Provides common functionality required by different drivers or modules
- Manages shared resources
- Eliminates potential conflicts
- Keeps the requests separate
- Typically does not provide *open* and *close* functions like a device driver



Timer System Service



Timer System Service APIs

- **Single Shot Timer**

```
SYS_TMR_HANDLE SYS_TMR_CallbackSingle(
    uint32_t periodMs,
    uintptr_t context,
    SYS_TMR_CALLBACK callback
);
```

- **Periodic Timer**

```
SYS_TMR_HANDLE SYS_TMR_CallbackPeriodic(
    uint32_t periodMs,
    uintptr_t context,
    SYS_TMR_CALLBACK callback
);
```

- **Delay Timer**

```
SYS_TMR_HANDLE SYS_TMR_DelayMS(
    uint32_t delayMs
);
```

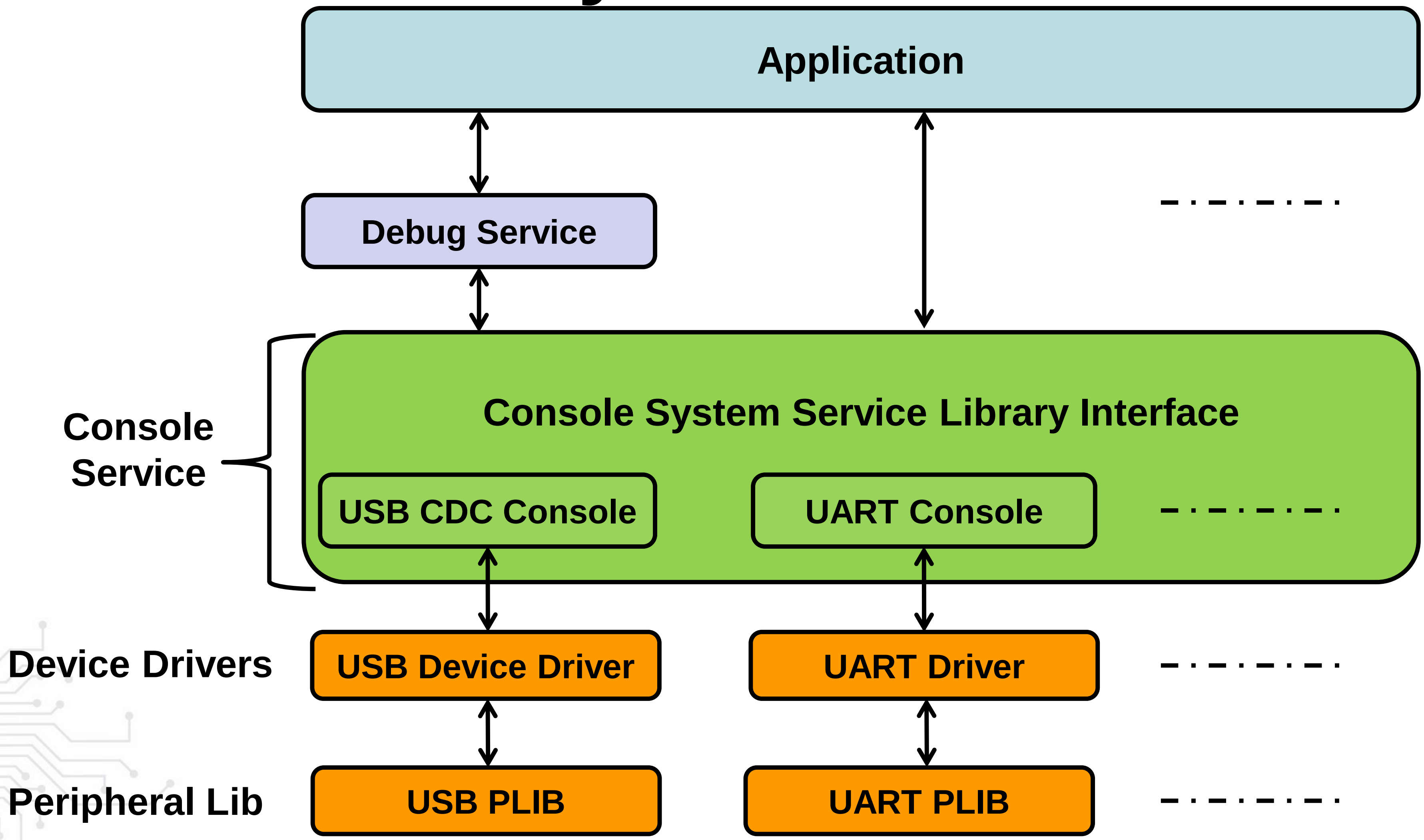
- **Check Delay Status**

```
bool SYS_TMR_DelayStatusGet(
    SYS_TMR_HANDLE handle
);
```

- **Delete Timer Object**

```
void SYS_TMR_CallbackStop(
    SYS_TMR_HANDLE handle
);
```

Console & Debug System Service



Console & Debug System Service APIs

❑Debug System Service

- SYS_DEBUG_MESSAGE(level, message)
- SYS_DEBUG_PRINT(level, fmt, ...)
- SYS_PRINT(fmt, ...)
- SYS_DEBUG_ErrorLevelSet (….)

❑Console System Service

- SYS_CONSOLE_Read(….)
- SYS_CONSOLE_Write(….)
- SYS_CONSOLE_RegisterCallback(….)

Quiz

1. If multiple applications need a delay, they will make use of …

   a) Timer Driver
   b) Timer System Service

   Answer: b) Timer System Service

2. Application event handlers (callback functions) are called by a driver from…

   a) Interrupt Context
   b) System Tasks

   Answer: Either a) or b)

3. Driver initialization is performed by ..

   a) System, using the System Interface provided by the driver
   b) Application, using the Client Interface provided by the driver

   Answer: a) System, using the System Interface provided by the driver

Lab 1

Create a MPLAB® Harmony Application using MPLAB Harmony Drivers and System Services

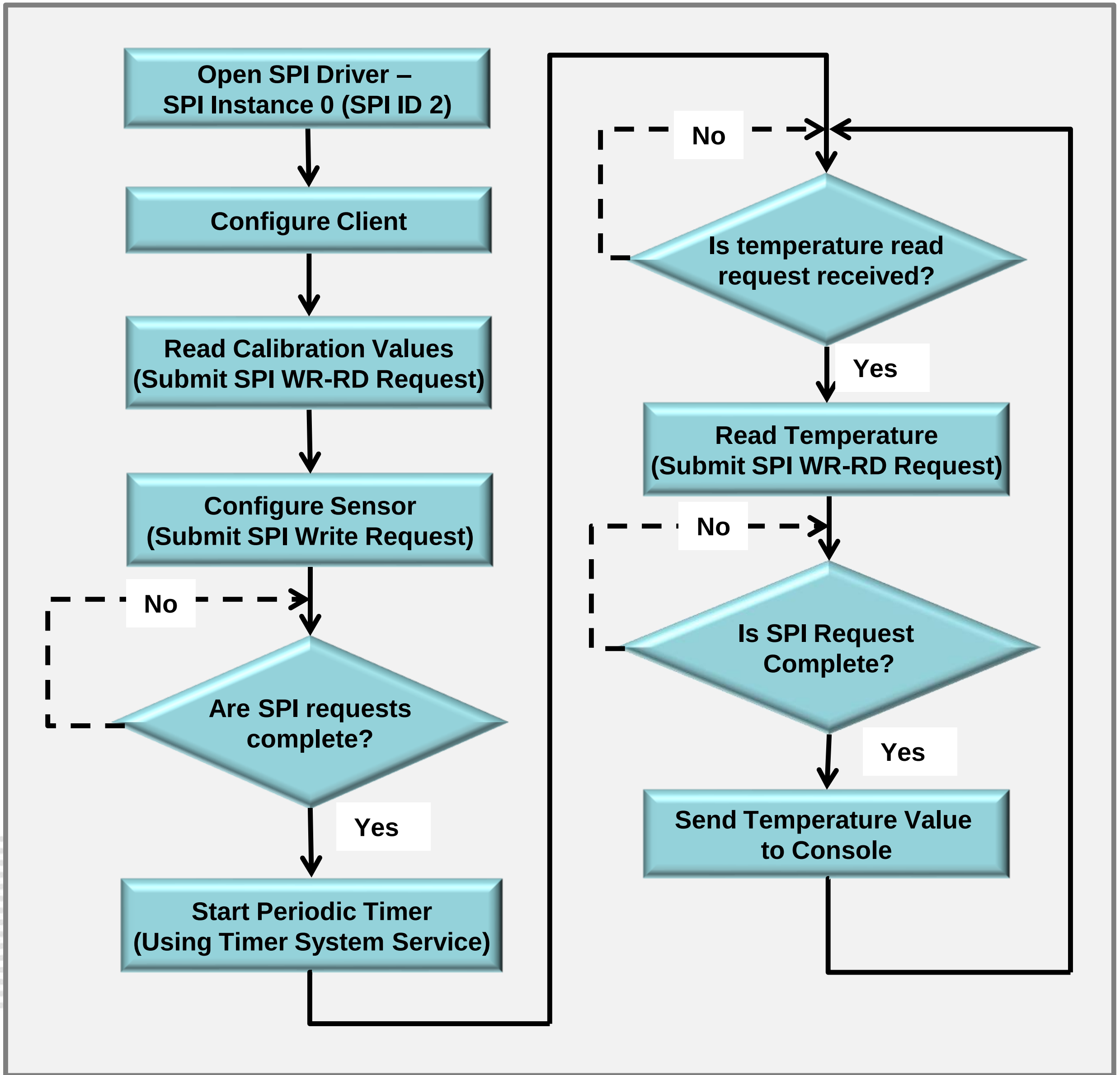
Lab 1: Objectives

Be able to create a MPLAB® Harmony application using the SPI driver configured in Interrupt mode

Be able to use the Timer and the Console System Service

Be able to use the Driver Callback and Status Polling mechanisms

Lab1: Logical Flowchart



Callbacks:

SENSOR_APP_PeriodicTimer_EventHandler:

readTemperatureReq = true;

SENSOR_APP_BufferEventHandler:

bufferStatus = event;

(event = SPI_BUFFER_STATUS_COMPLETE
OR
event = SPI_BUFFER_STATUS_ERROR)

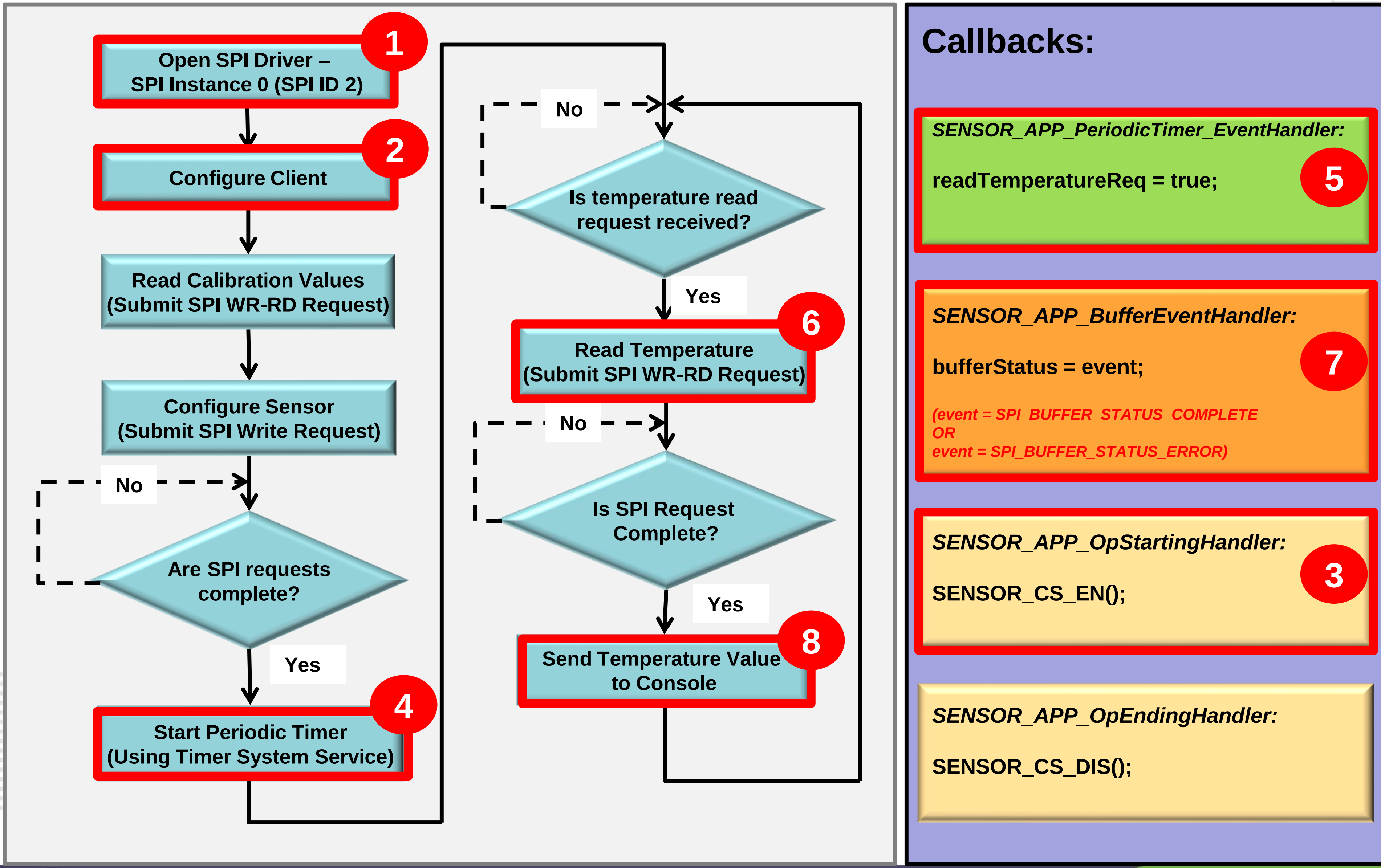
SENSOR_APP_OpStartingHandler:

SENSOR_CS_EN();

SENSOR_APP_OpEndingHandler:

SENSOR_CS_DIS();

Lab1: Logical Flowchart



Lab 1: Summary

In this lab we have...

- Created a MPLAB® Harmony application to read sensor data using the SPI driver running in Interrupt mode
- Used the Timer System Service to read the sensor data at periodic intervals.
- Used the (USB CDC) Console System Service to send the sensor data to PC
- Used the driver callback and status polling mechanisms

Class Agenda

MPLAB® Harmony Key Concepts

Harmony Drivers and System Services

- Lab1: Create a MPLAB Harmony Application using MPLAB Harmony Drivers and System Services.

Harmony Drivers Advanced Usage

- Lab2: Use Harmony Driver in Multi Instance Configuration

Using MPLAB Harmony in an RTOS environment

- Lab3: Add RTOS to the Application

Summary

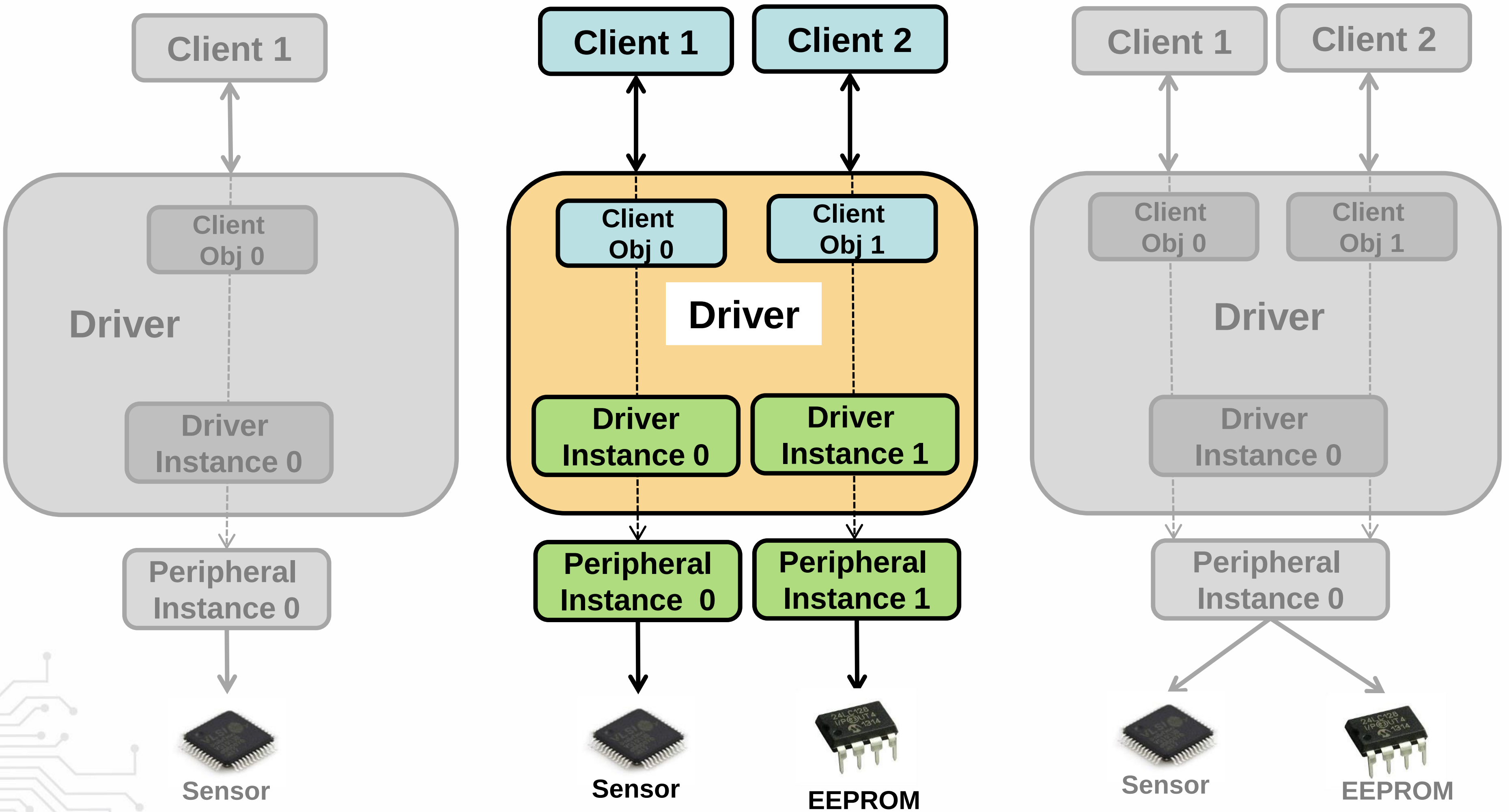
MPLAB® Harmony Drivers Advanced Usage

Multiple Instances

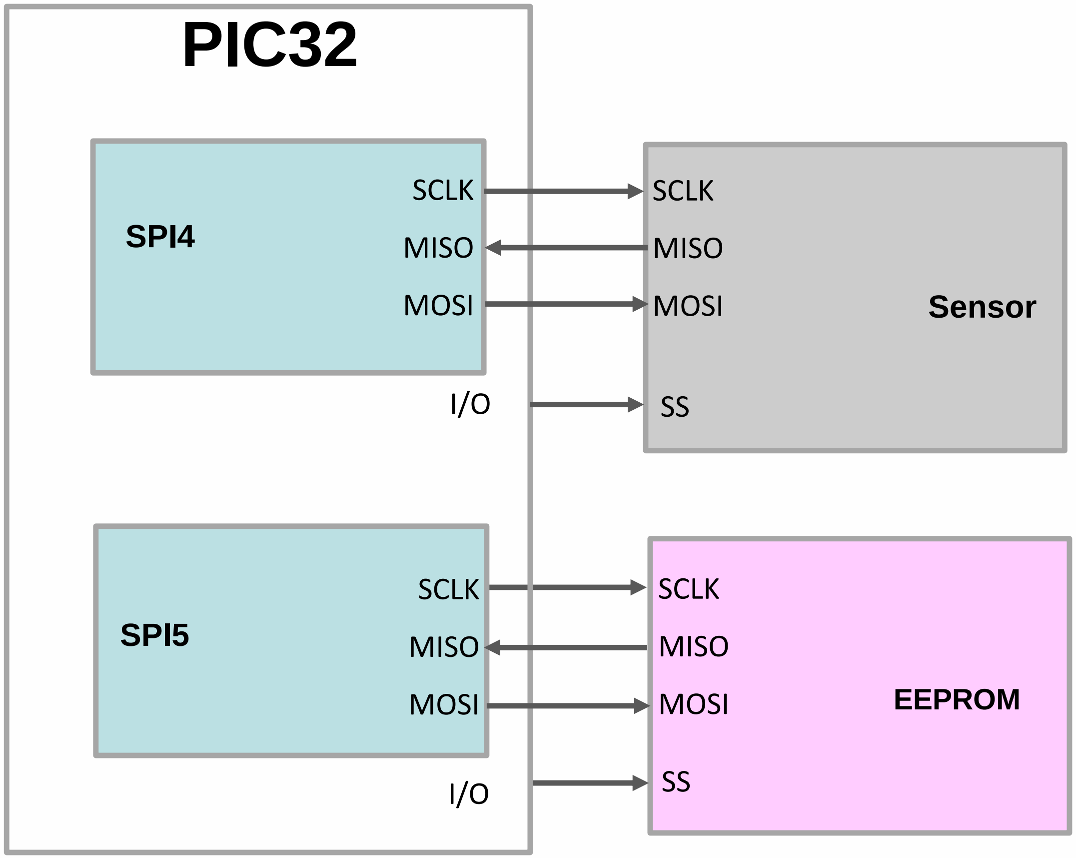
Multiple Clients

Interrupt Safety

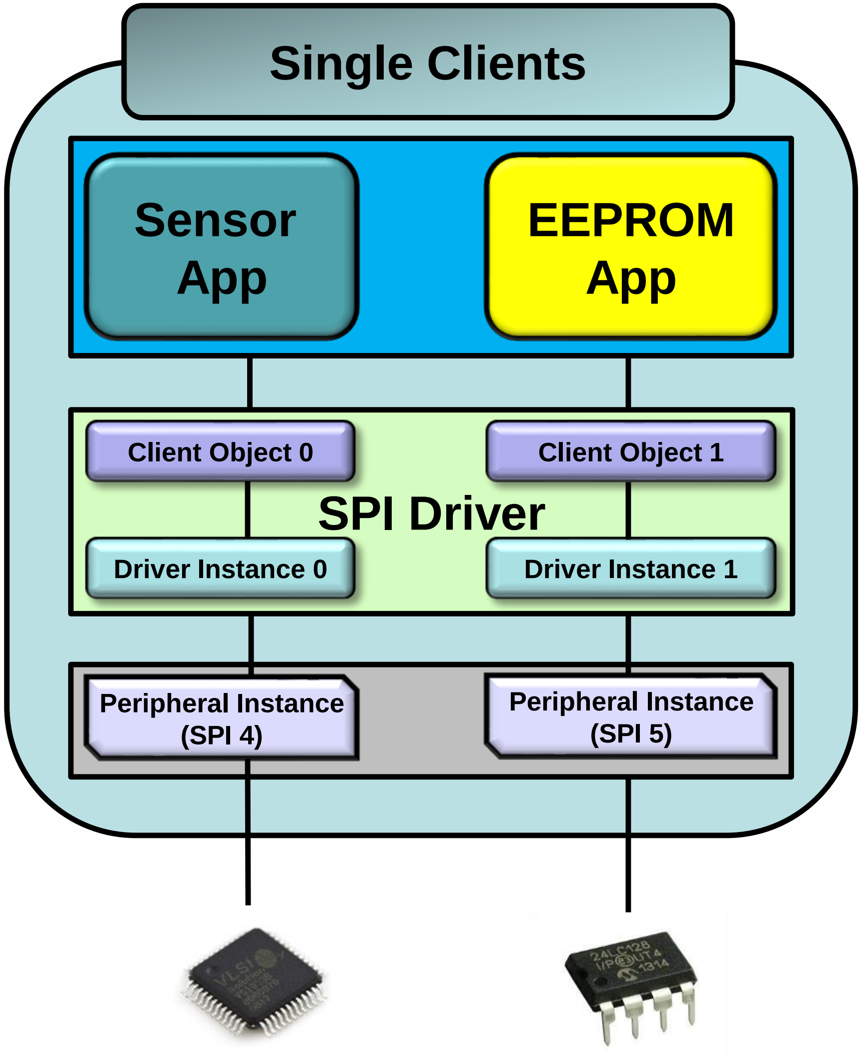
Driver Usage Models



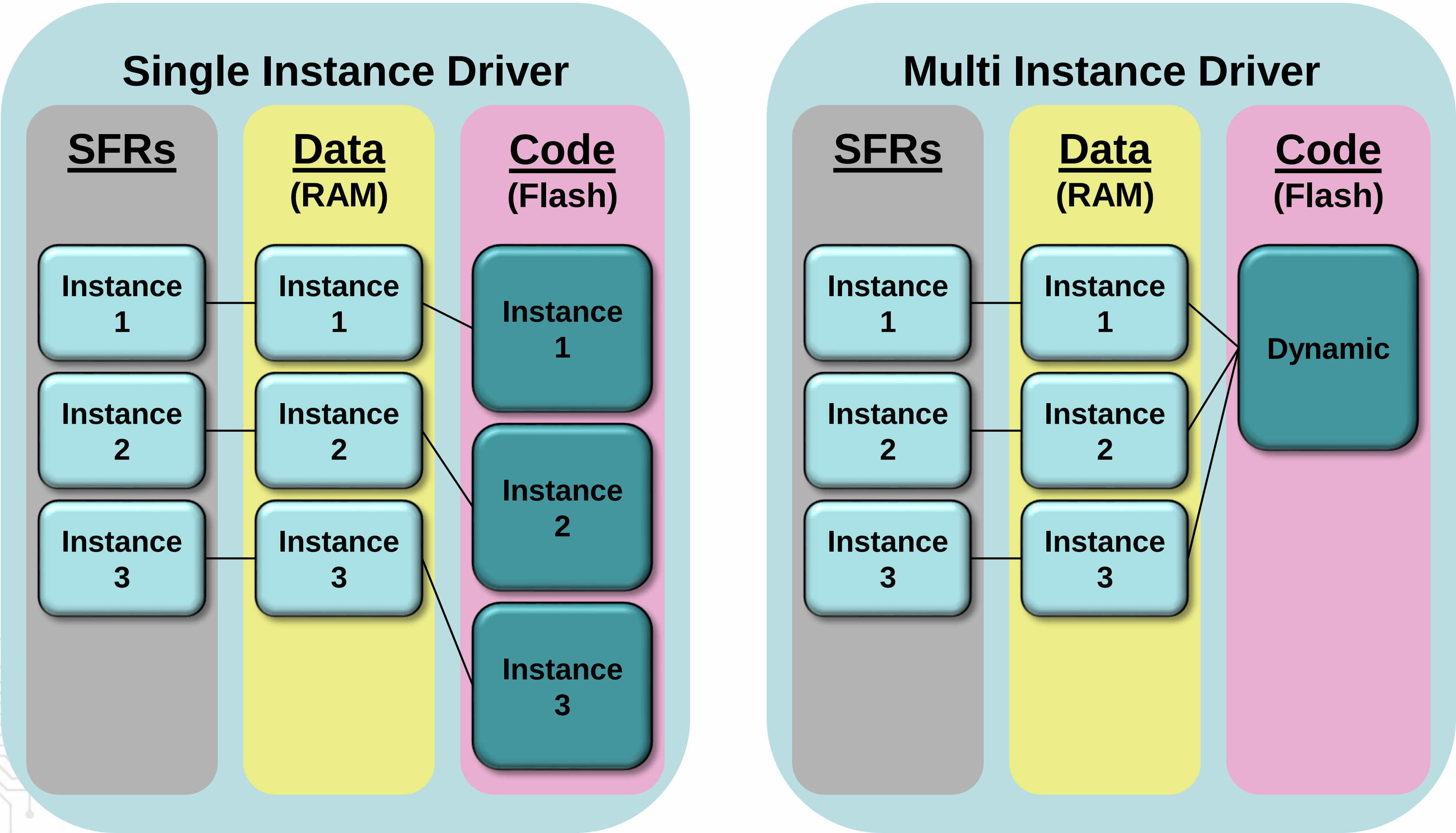
Interfacing with Multiple Devices



Multiple Instances of a Driver



Single Instance vs. Multi Instance Driver

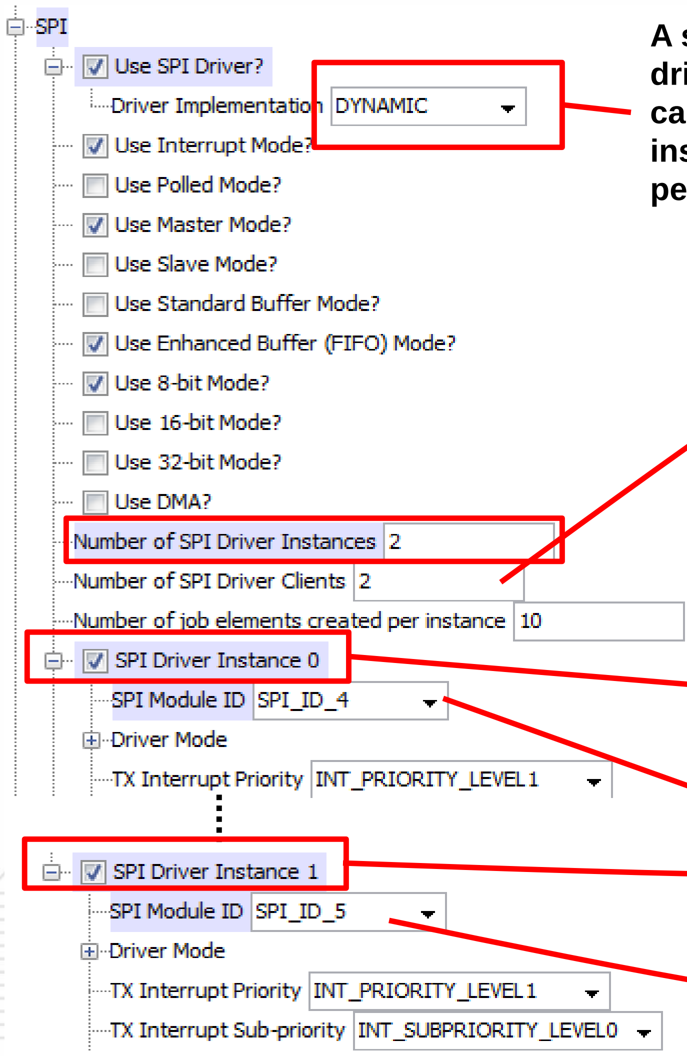


Single Instance vs. Multi Instance Driver

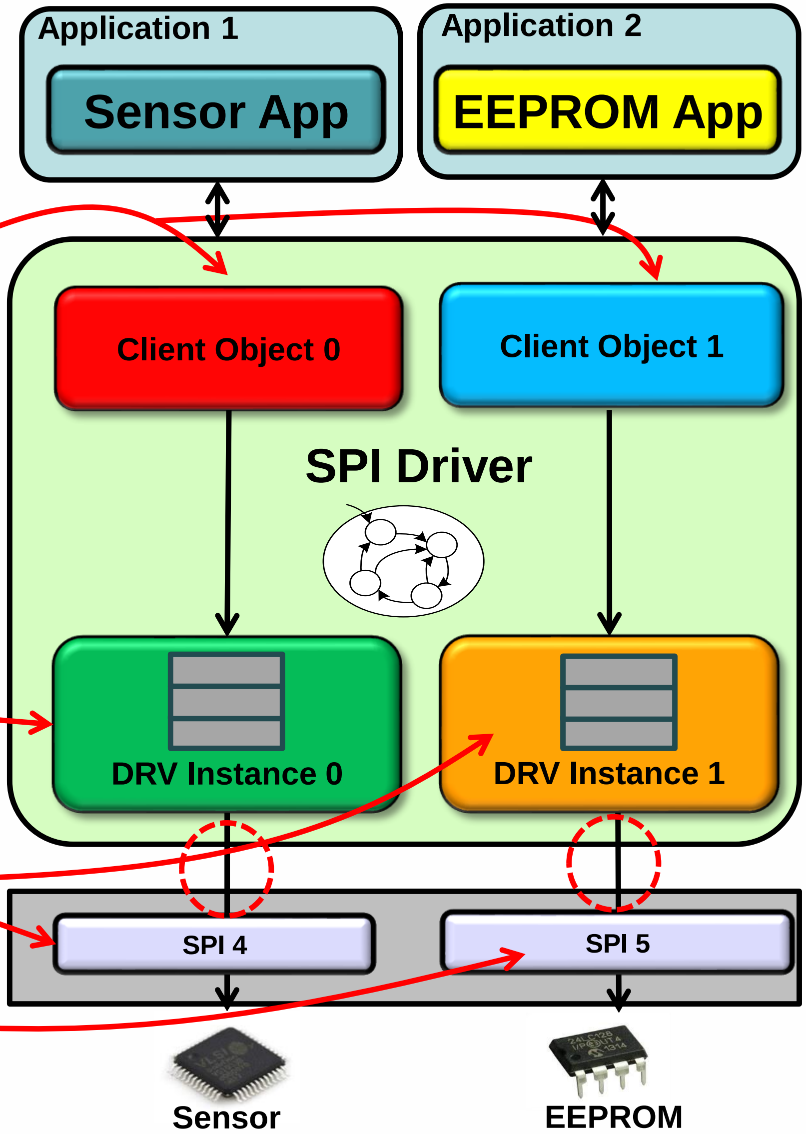
- ❑Single Instance Driver
  - ✓ Suitable for single instance of a peripheral
  - ✓ Driver source code duplicated for every instance
  - ✓ Results in larger driver code size
  - ✓ Peripheral registers of the instance are hard-coded in the driver source code

- ❑ Multi Instance Driver
  - ✓ Suitable for multiple instances of a peripheral
  - ✓ Single copy of driver source code
  - ✓ Results in reduced driver code size
  - ✓ Peripheral registers of the instance are accessed using an index or a pointer

Enabling Two Instances of SPI Peripheral in MHC



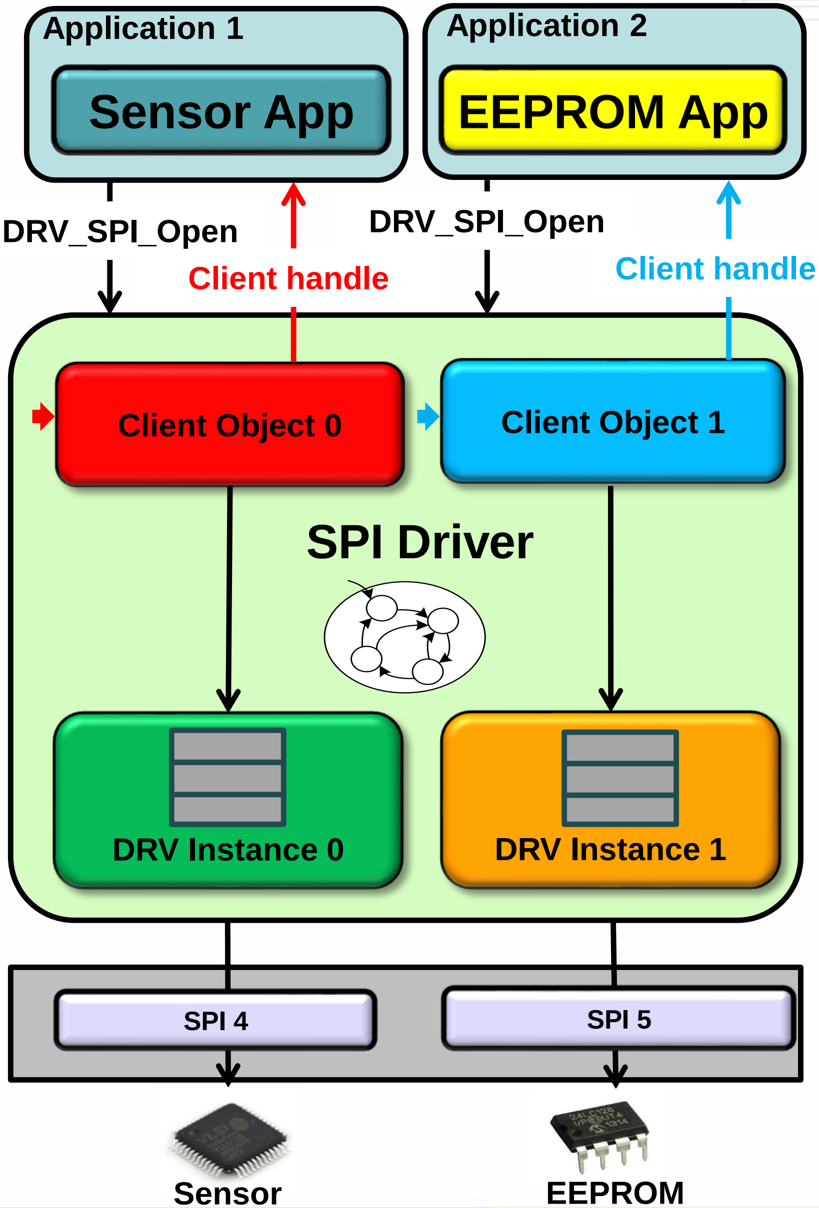
A single copy of the driver implementation can support multiple instances of the peripheral



Two SPI : Opening the Driver

```
void SensorApp( void )
{
    sensorClientHandle = DRV_SPI_Open (
        DRV_SPI_INDEX_0,
        DRV_IO_INTENT_READWRITE
    );

    DRV_SPI_ClientConfigure (
        sensorClientHandle,
        &sensorClientData
    );

}
```

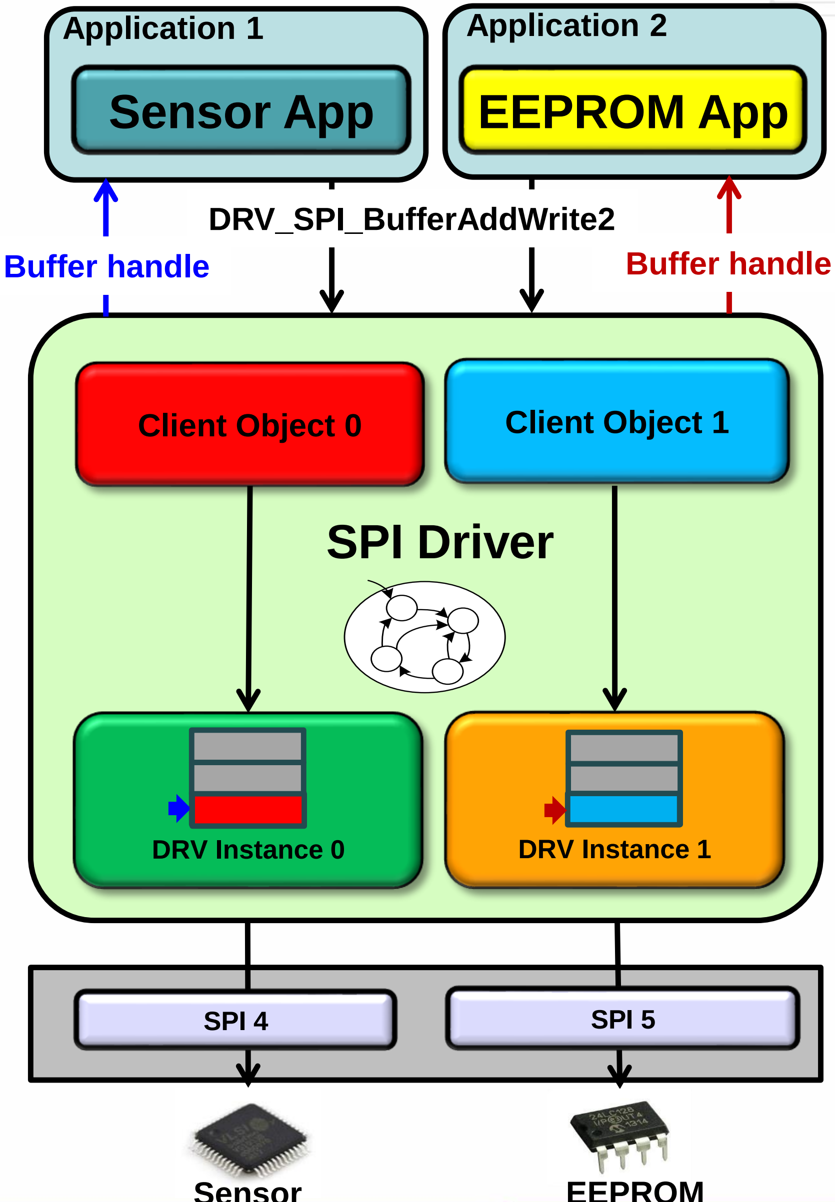
```
void EEPROMApp( void )
{
    eepromClientHandle = DRV_SPI_Open (
        DRV_SPI_INDEX_1,
        DRV_IO_INTENT_READWRITE
    );

    DRV_SPI_ClientConfigure (
        eepromClientHandle,
        &eepromClientData
    );
}
```



Two SPI : Submitting Requests

```
DRV_SPI_BufferAddWrite2 (
        sensorClientHandle,
        (void*)sensorTask.wrBuffer,
        sensorTask.nBytes,
        sensorBufferEventHandler,
        NULL,
        &sensorBufferHandle
    );
```

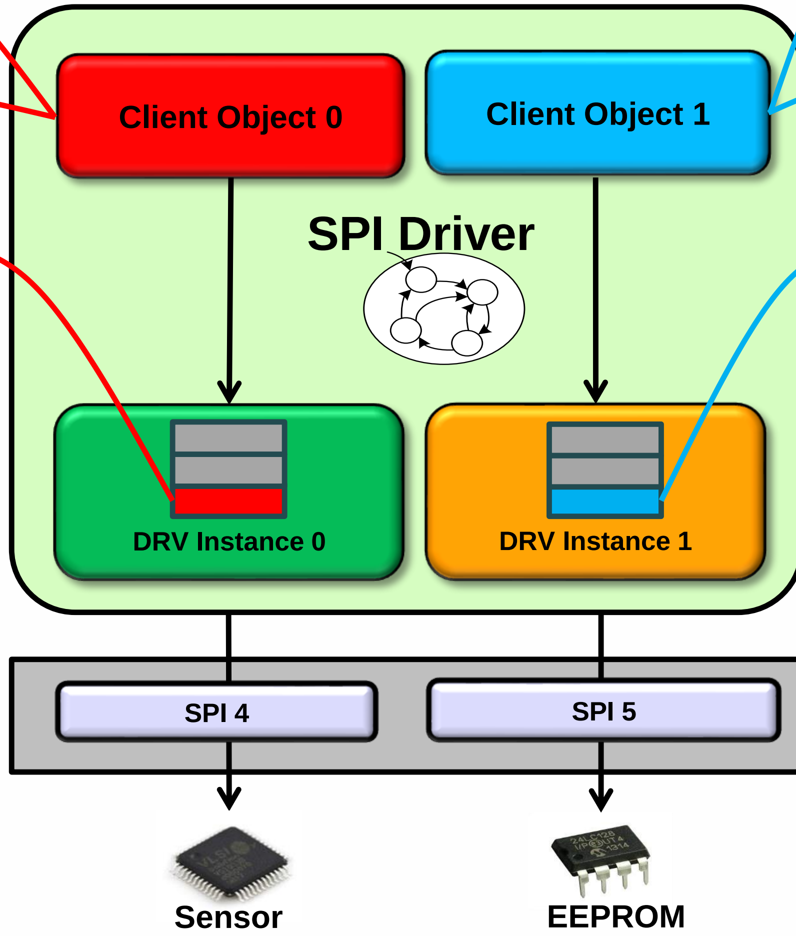
```
DRV_SPI_BufferAddWrite2 (
        eepromClientHandle,
        (void*)eepromTask.wrBuffer,
        eepromTask.nBytes,
        eepromBufferEventHandler,
        NULL,
        &eepromBufferHandle
    );
```



Two SPI : Callbacks

```
void sensorOpStarting(…)
{
   SENSOR_EN_CS();
}

void sensorOpEnding(…)
{
   SENSOR_DIS_CS();
}

void sensorBufferEventHandler(…)
{
   sensorReqCompleted = true;
}
```

Sensor App




```
void eepromOpStarting(…)
{
   EEPROM_EN_CS();
}

void eepromOpEnding(…)
{
   EEPROM_DIS_CS();
}

void eepromBufferEventHandler(…)
{
   eepromReqCompleted = true;
}
```

EEPROM App

Lab 2

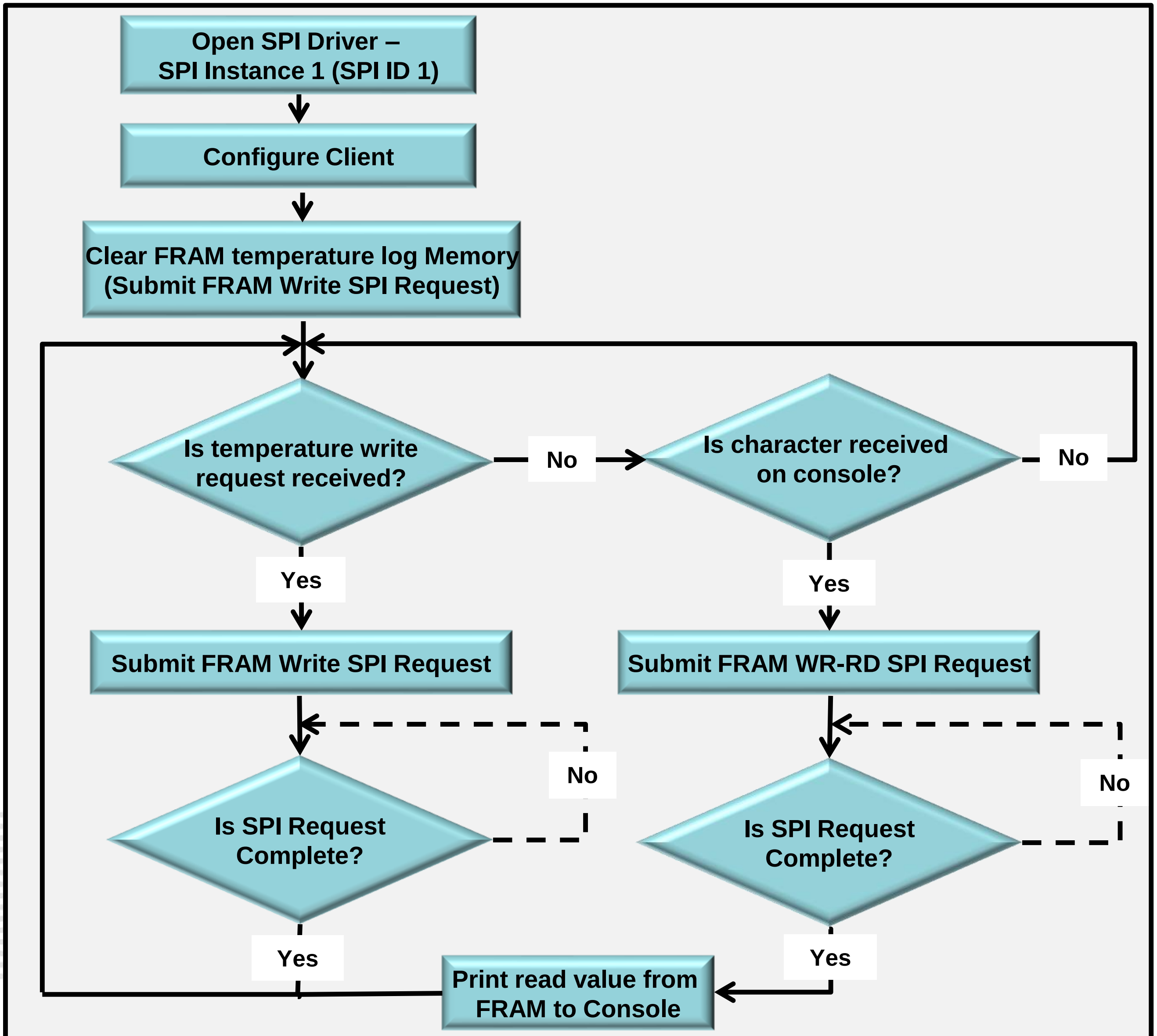
Use MPLAB® Harmony Driver in Multi Instance Configuration

Lab 2: Objectives

Be able to add a second application to write data to FRAM

Be able to add a second instance of the SPI Driver and demonstrate multiple instances of the Driver

Lab2: Logical Flowchart



Callbacks:

FRAM_APP_BufferEventHandler:

bufferStatus = event;

(event = SPI_BUFFER_STATUS_COMPLETE
OR
event = SPI_BUFFER_STATUS_ERROR)

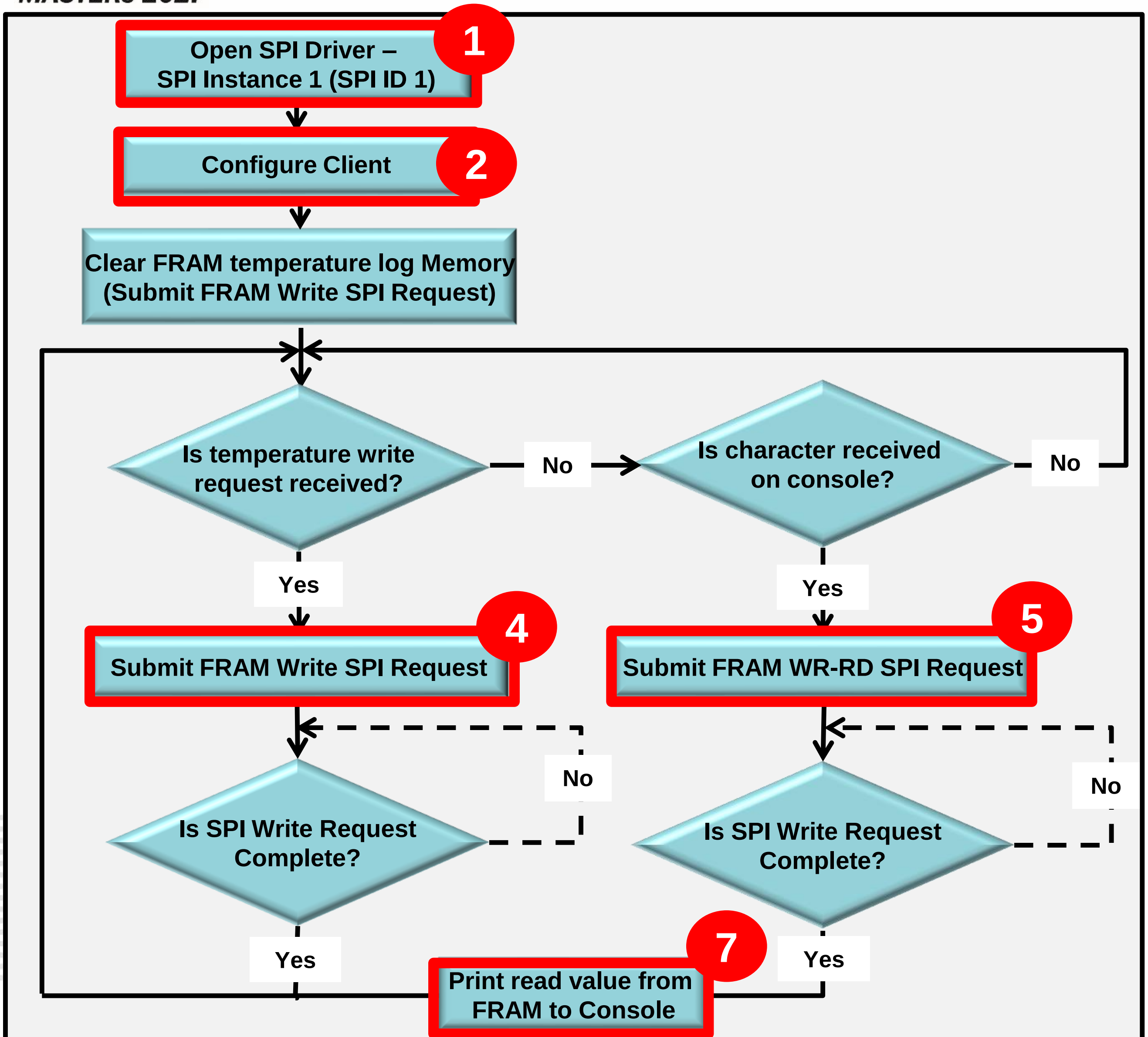
FRAM_APP_OpStartingHandler:

FRAM_CS_EN();

FRAM_APP_OpEndingHandler:

FRAM_CS_DIS();

Lab2: Logical Flowchart



Callbacks:

FRAM_APP_BufferEventHandler:

bufferStatus = event; 6

(event = SPI_BUFFER_STATUS_COMPLETE OR event = SPI_BUFFER_STATUS_ERROR)

FRAM_APP_OpStartingHandler:

FRAM_CS_EN(); 3

FRAM_APP_OpEndingHandler:

FRAM_CS_DIS();

Lab 2: Summary

In this lab we have...

- **Added a new application to write temperature data to FRAM using second instance of SPI**

- **Demonstrated usage of Harmony drivers with multiple peripheral instances enabled**

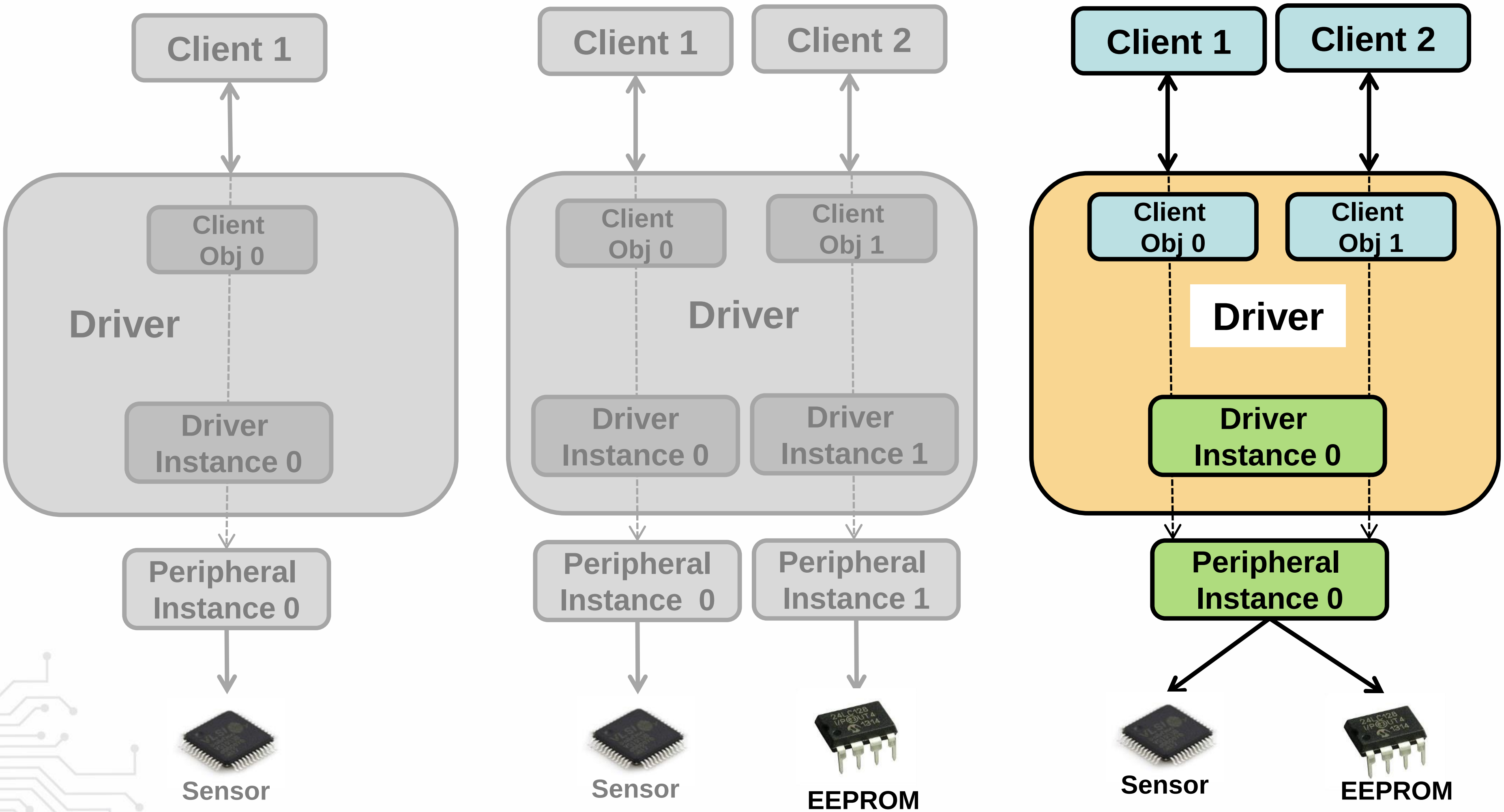
MPLAB® Harmony Drivers Advanced Usage

Multiple Instances

Multiple Clients

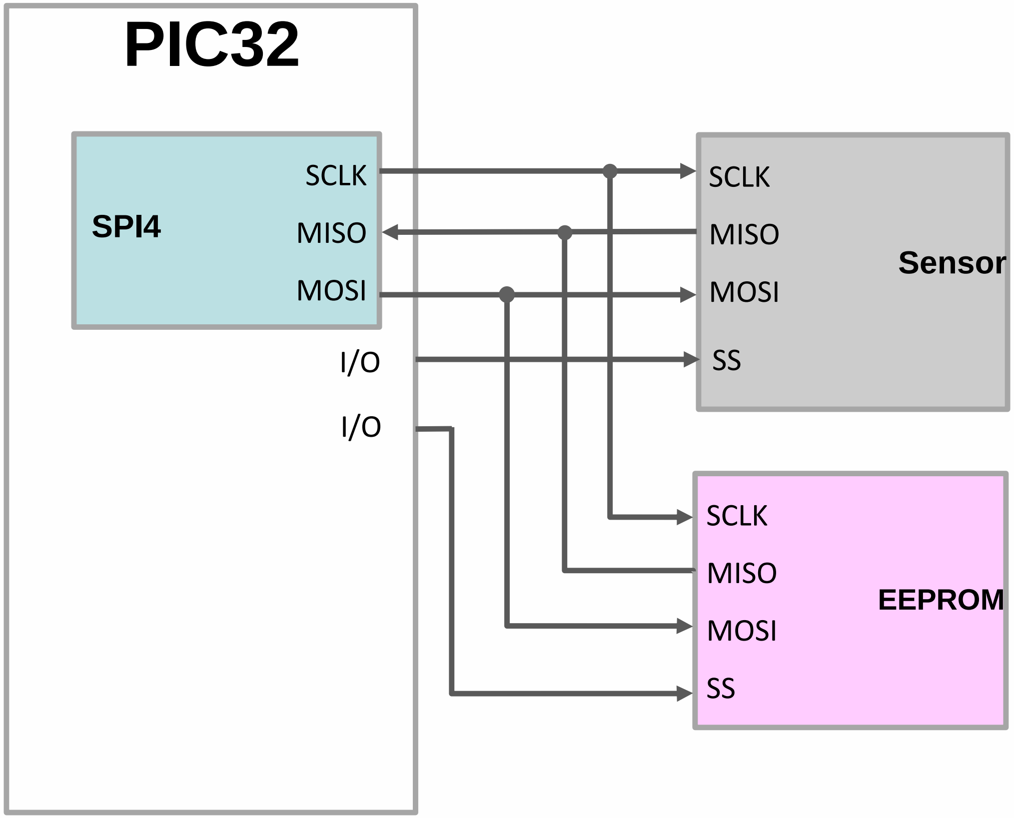
Interrupt Safety

Driver Usage Models

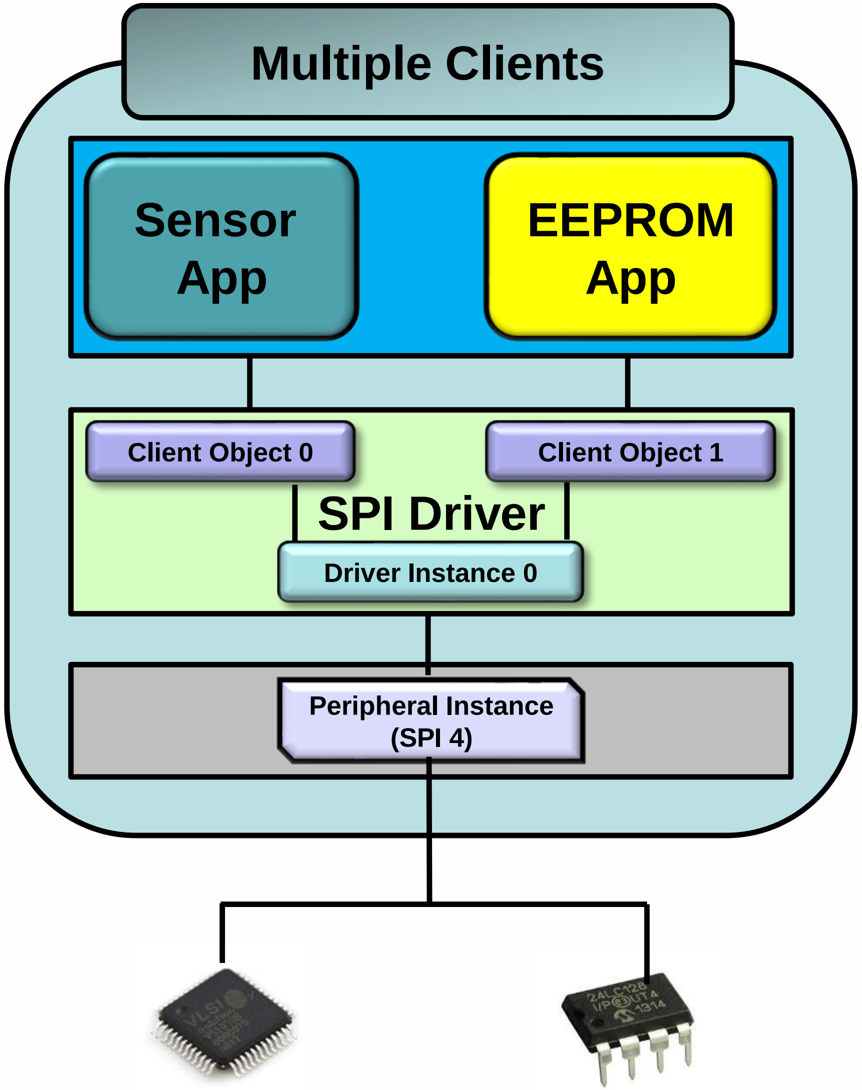


Interfacing with Multiple devices

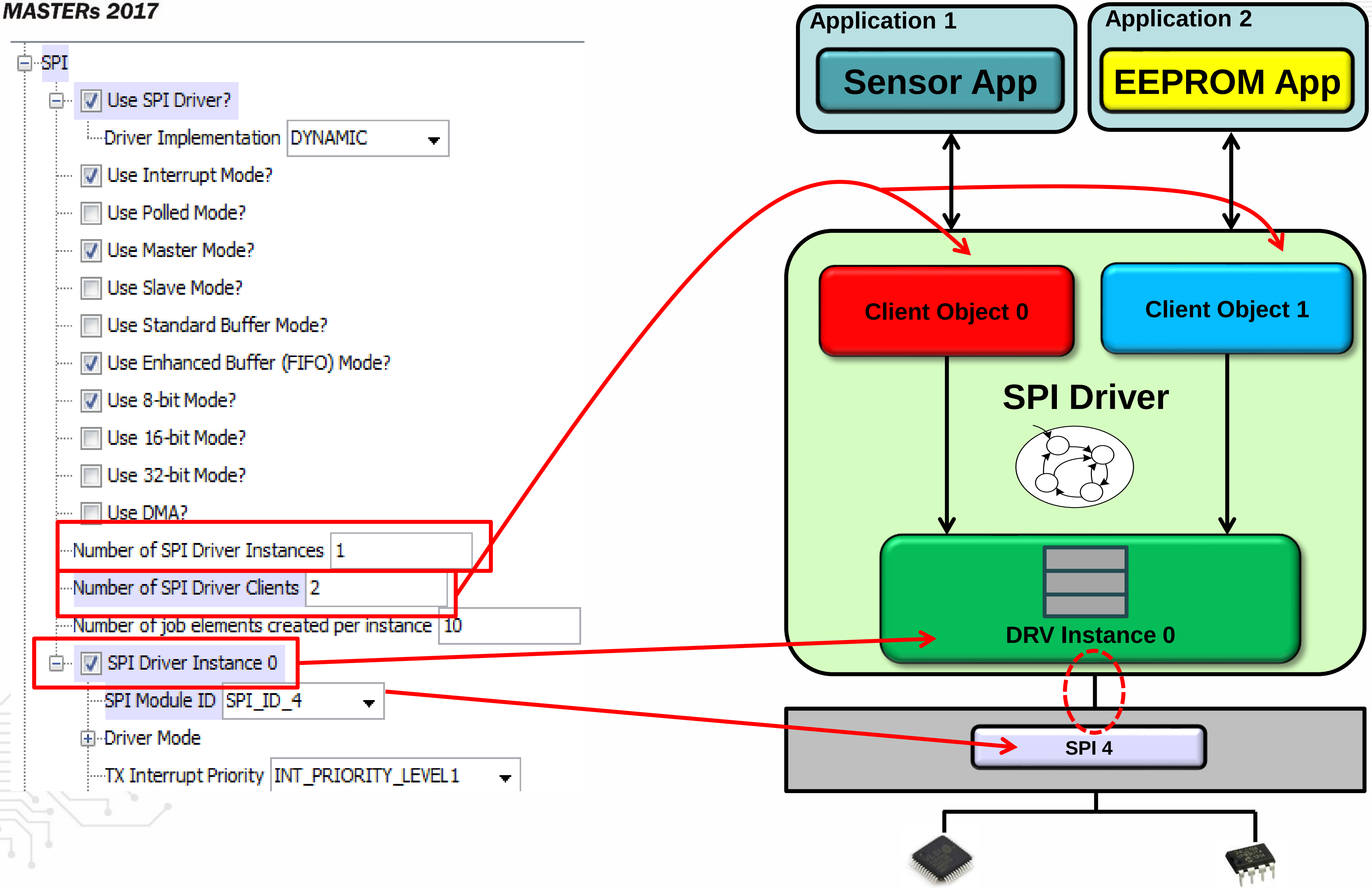
Reduced I/O usage



One Driver Instance, Multiple Clients



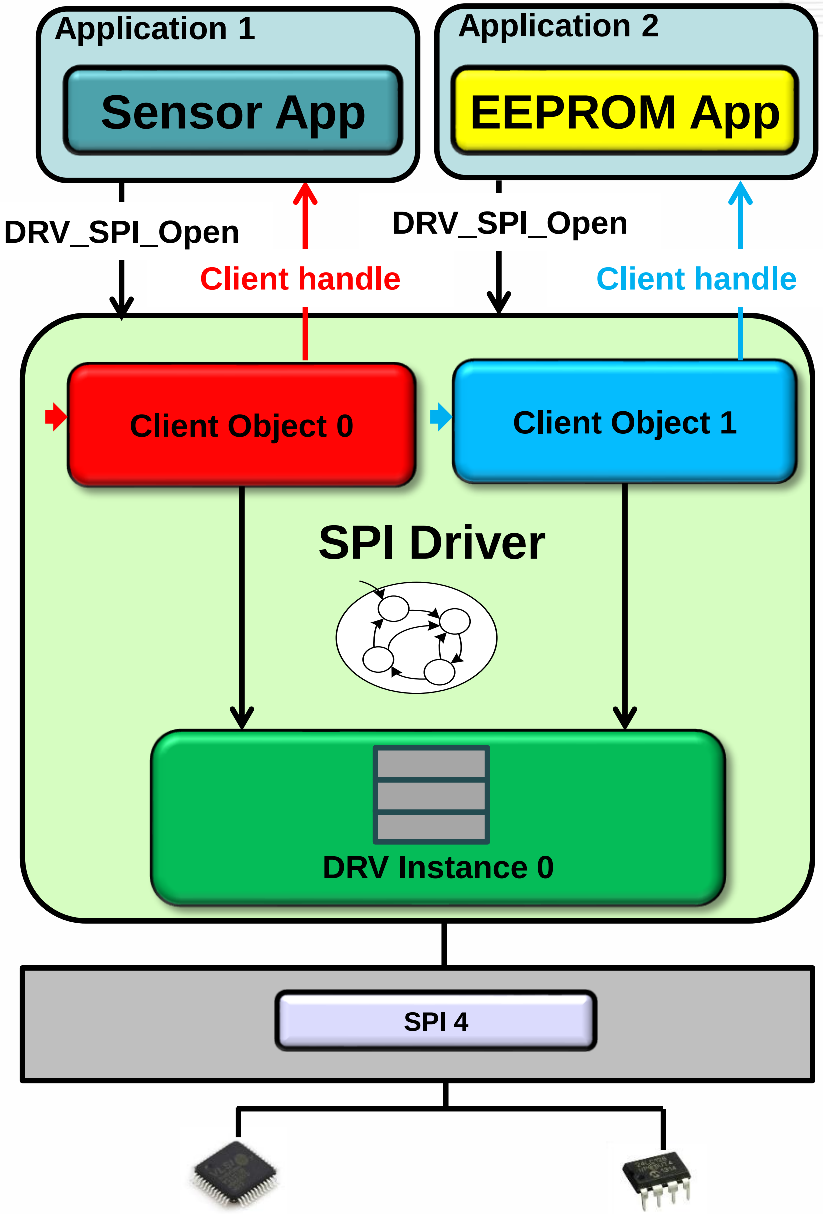
Enabling Two Clients for a Driver Instance in MHC



One SPI, Two Clients: Opening driver

```
void SensorApp( void )
{
    sensorClientHandle = DRV_SPI_Open (
        DRV_SPI_INDEX_0,
        DRV_IO_INTENT_READWRITE
    );

    DRV_SPI_ClientConfigure (
        sensorClientHandle,
        &sensorClientData
    );

}
```

```
void EEPROMApp( void )
{
    eepromClientHandle = DRV_SPI_Open (
        DRV_SPI_INDEX_0,
        DRV_IO_INTENT_READWRITE
    );

    DRV_SPI_ClientConfigure (
        eepromClientHandle,
        &eepromClientData
    );
}
```

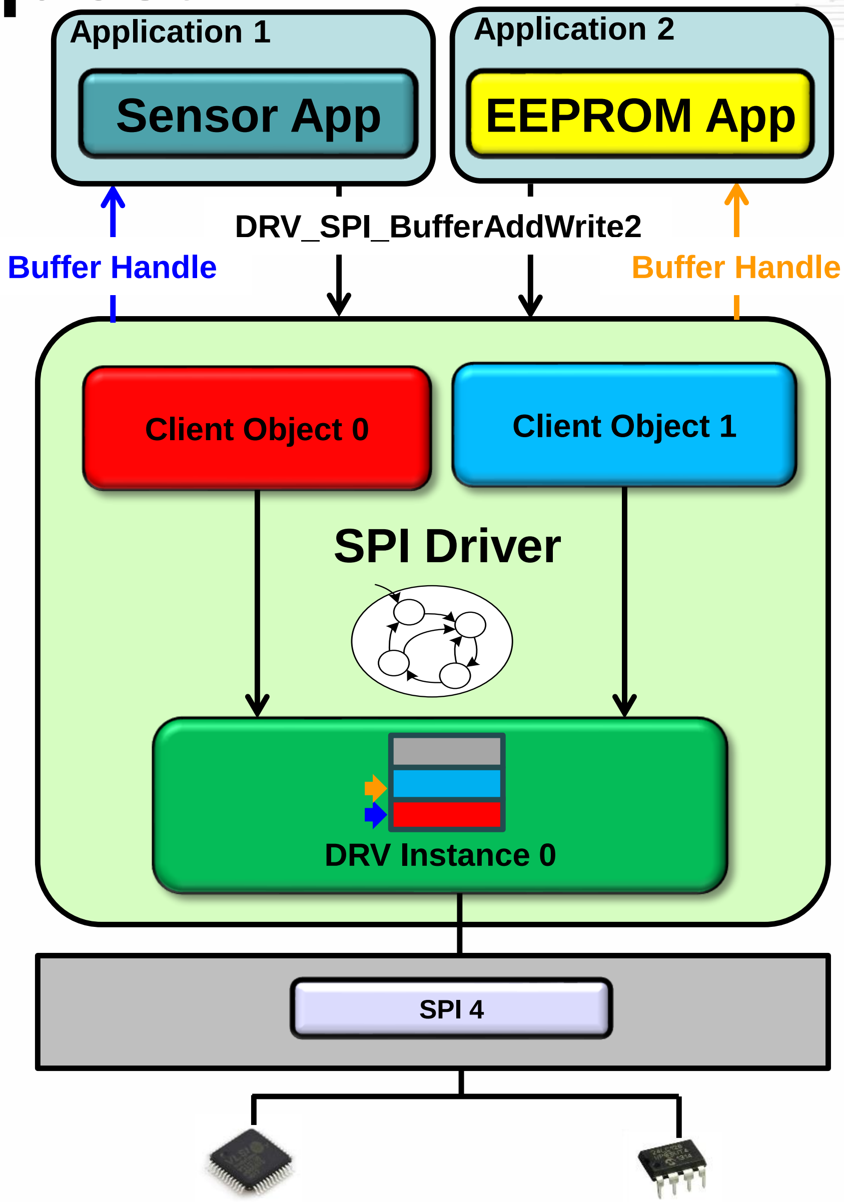


One SPI, Two Clients: Submitting Request

MICROCHIP MASTERs 2017

```
DRV_SPI_BufferAddWrite2 (
        sensorClientHandle,
        (void*)sensorTask.wrBuffer,
        sensorTask.nBytes,
        snsorBufferEventHandler,
        NULL,
        &sensorBufferHandle
 );
```

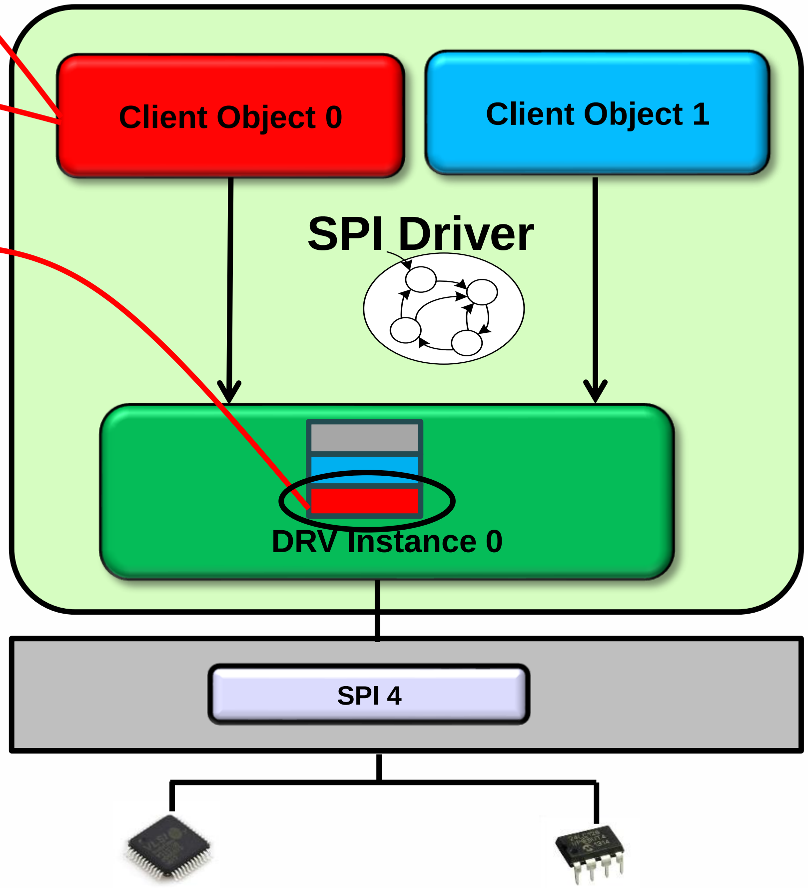
```
DRV_SPI_BufferAddWrite2 (
        eepromClientHandle,
        (void*)eepromTask.wrBuffer,
        eepromTask.nBytes,
        eepromBufferEventHandler,
        NULL,
        &eepromBufferHandle
 );
```



One SPI, Two Clients: Callback

```
void sensorOpStarting(…)
{
   SENSOR_EN_CS();
}

void sensorOpEnding(…)
{
    SENSOR_DIS_CS();
}

void sensorBufferEventHandler(…)
{
    sensorReqCompleted = true;
}
```

Sensor App



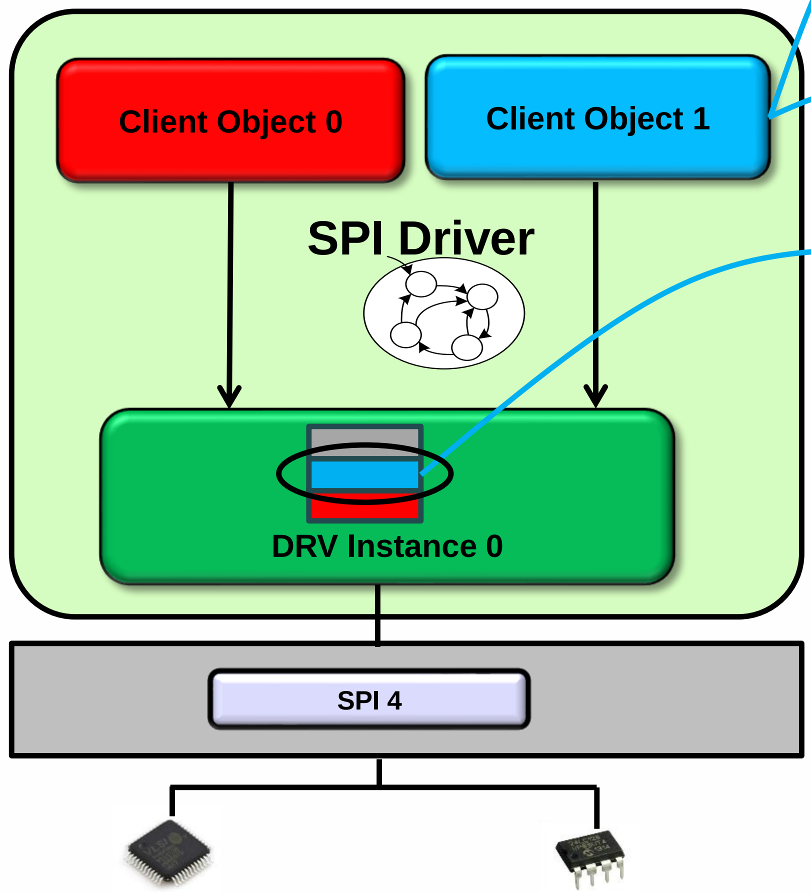

```
void eepromOpStarting(…)
{
   EEPROM_EN_CS();
}

void eepromOpEnding(…)
{
    EEPROM_DIS_CS();
}

void eepromBufferEventHandler(…)
{
    eepromReqCompleted = true;
}
```

EEPROM App

One SPI, Two Clients: Callback

```
void sensorOpStarting(…)
{
   SENSOR_EN_CS();
}

void sensorOpEnding(…)
{
    SENSOR_DIS_CS();
}

void sensorBufferEventHandler(…)
{
    sensorReqCompleted = true;
}
```

Sensor App




```
void eepromOpStarting(…)
{
   EEPROM_EN_CS();
}

void eepromOpEnding(…)
{
    EEPROM_DIS_CS();
}

void eepromBufferEventHandler(…)
{
    eepromReqCompleted = true;
}
```

EEPROM App

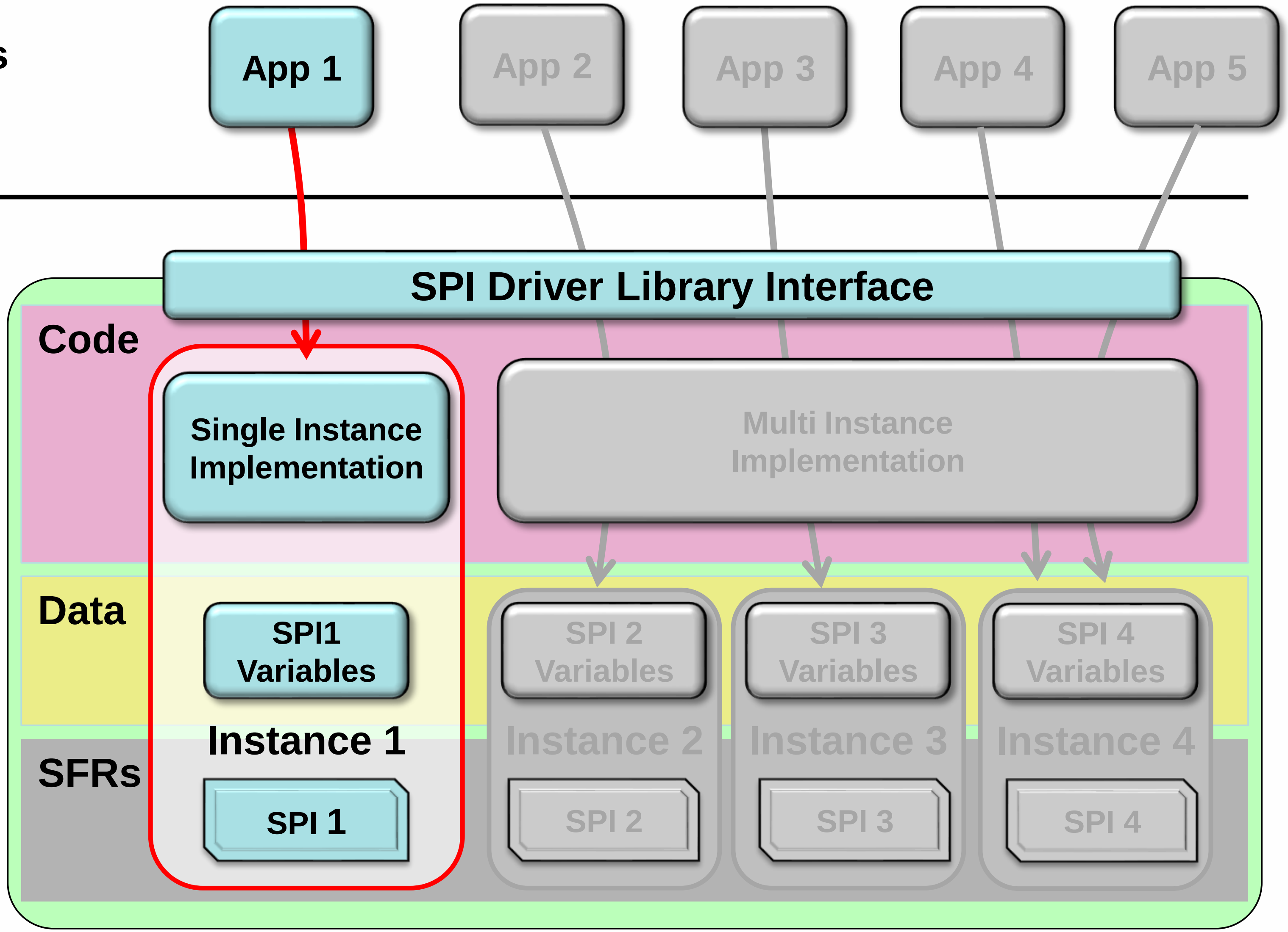
Key Relationships

Multiple Clients Possible

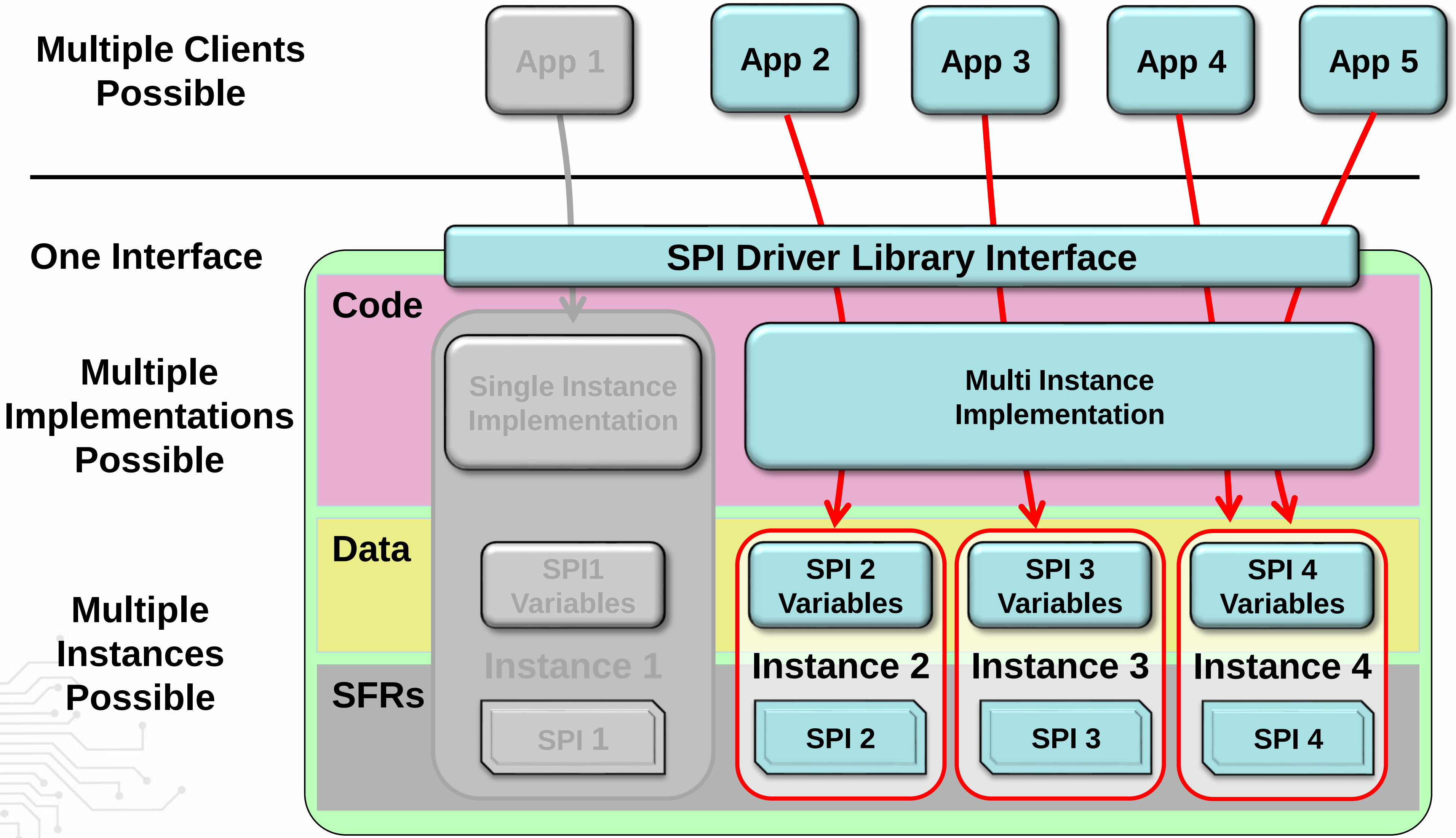
One Interface

Multiple Implementations Possible

Multiple Instances Possible



Key Relationships



MPLAB® Harmony Drivers Advanced Usage

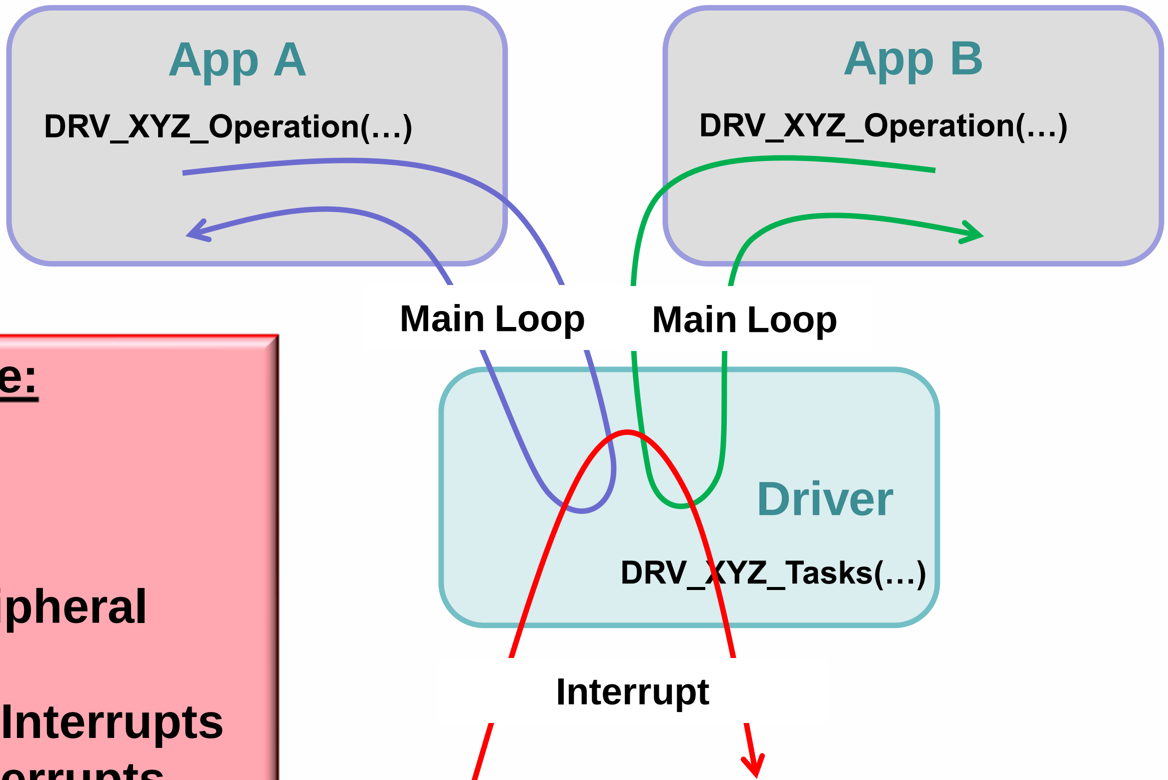
Multiple Instances

Multiple Clients

Interrupt Safety

Interrupt Safety: Driver In Interrupt mode

• Driver Running in Interrupt Mode



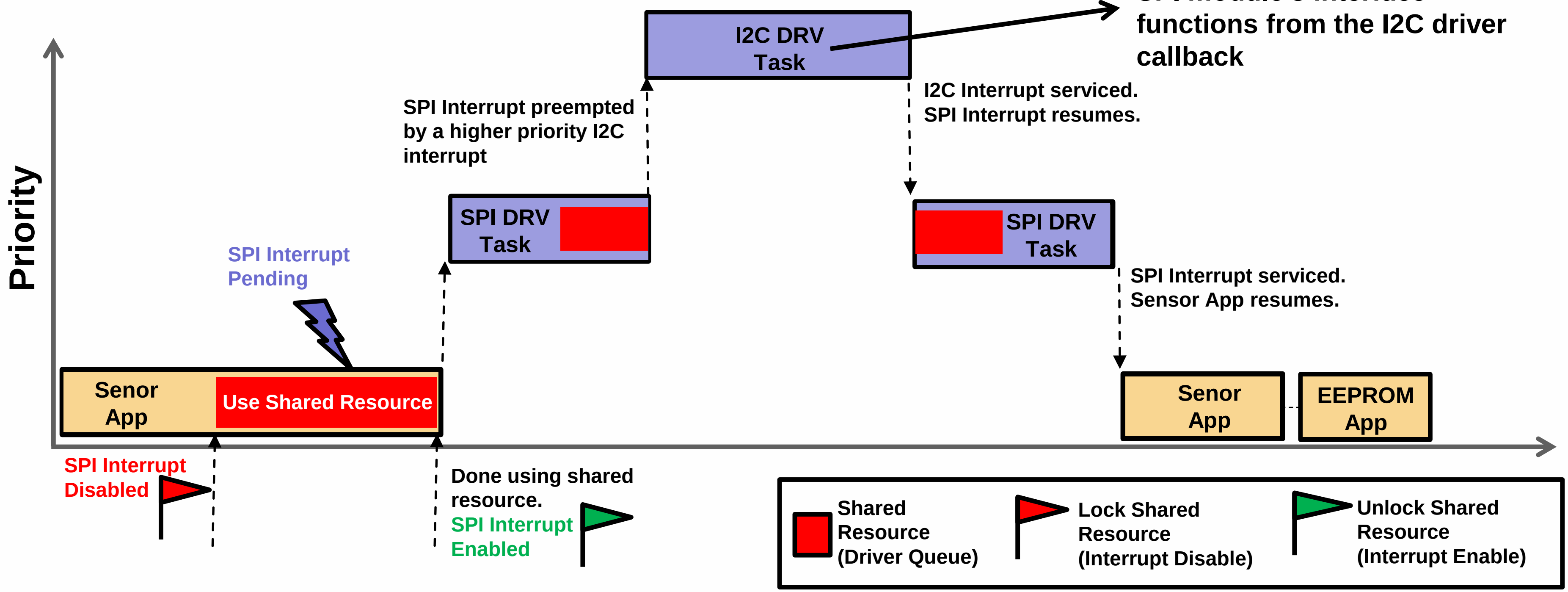
Accessing Shared Resource:

Harmony Drivers…

- **Disables Associated Peripheral Interrupt (briefly)**
- **Does not Disable Global Interrupts**
- **Allows higher priority Interrupts, protecting their response time latency**

Interrupt Safety: Driver In Interrupt mode

Application must not call the SPI module's interface functions from the I2C driver callback




```
while(1)
{
      void SYS_Tasks ( void )
      {
            SensorApp();

            EEPROMApp();
      }
}
```

```
void __ISR(_SPI1_TX_VECTOR, ipl3AUTO)
_IntHandlerSPITxInstance0(void)
{

    DRV_SPI_Tasks(sysObj.spiObjectIdx0);

}
```

Quiz

1. A Harmony Driver can have multiple clients to a single peripheral instance

   a) True
   b) False

   Answer: a) True

2. If an application has 3 slave devices interfaced to a single SPI peripheral instance, it…

   a) Sets the Number of SPI Driver Instances to 1
   b) Sets the Number of SPI Driver Instances to 3

   Answer: a) Sets the Number of SPI Driver Instances to 1

3. Harmony drivers implement interrupt safety by..

   a) Disabling the Global Interrupts
   b) Disabling the associated peripheral interrupt

   Answer: b) Disabling the associated peripheral interrupt

Class Agenda

MPLAB® Harmony Key Concepts

Harmony Drivers and System Services

- Lab1: Create a MPLAB Harmony Application using MPLAB Harmony Drivers and System Services.

Harmony Drivers Advanced Usage

- Lab2: Use Harmony Driver in Multi Instance Configuration

Using MPLAB Harmony in an RTOS environment

- Lab3: Add RTOS to the Application

Summary

Using MPLAB® Harmony in an RTOS environment

Why use RTOS?

- Removes dependency of tasks on the execution time of other functions running in the “super loop”
- Ensures better responsiveness to events
- Allows hard real time tasks to meet its deadlines
- Does not waste CPU cycles
- Idle task allows putting CPU in low power modes

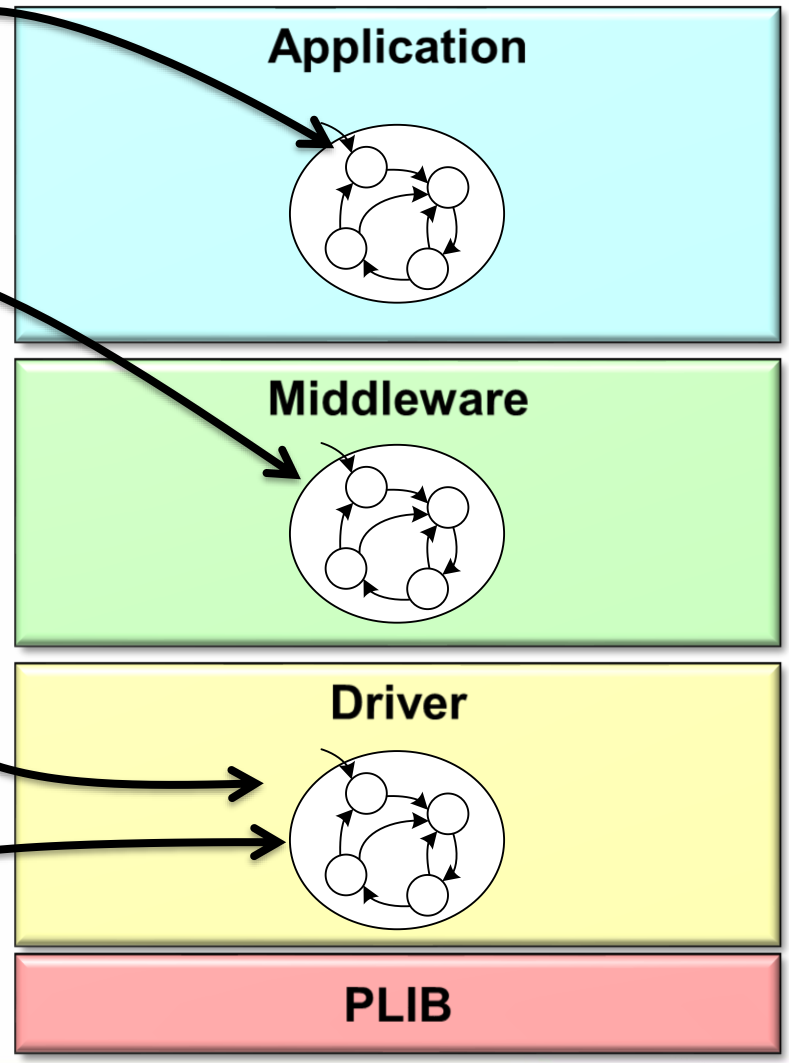
RTOS Driven

```
static void _SENSORTASK_Tasks(void)
{
    while(1)
    {
        SENSORTASK_Tasks();
        vTaskDelay(10); /*Give time back*/
    }
}
```

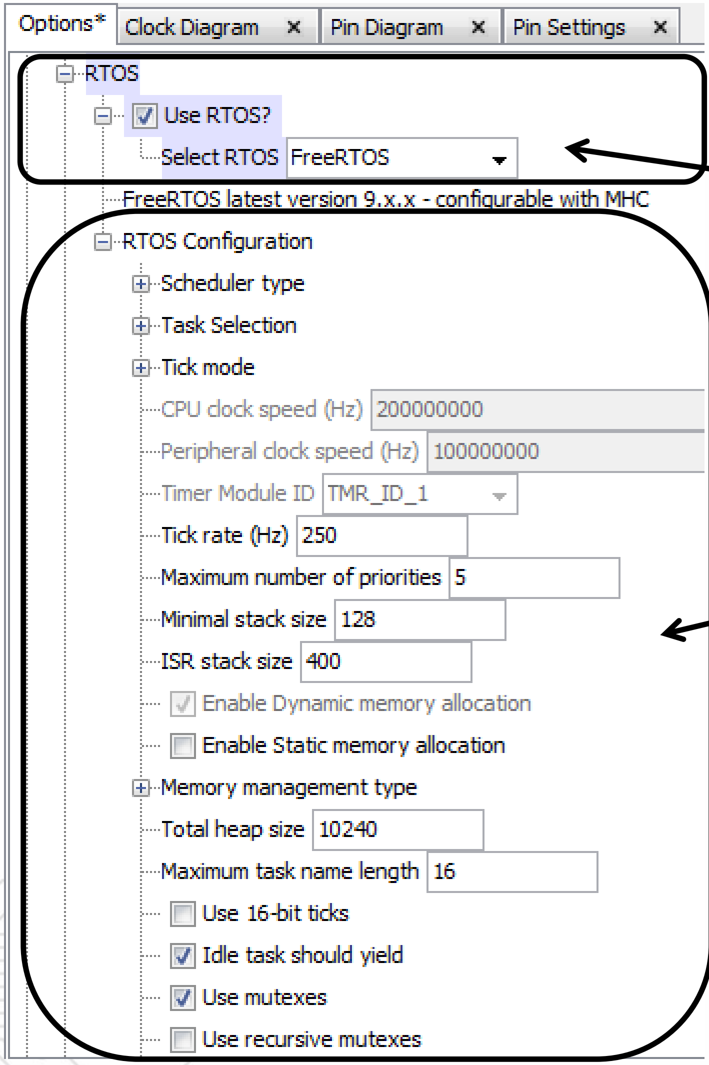
```
static void _GFX_Tasks(void)
{
    while(1)
    {
        GFX_Update();
        vTaskDelay(10); /*Give time back*/
    }
}
```

```
void _SYS_Tasks (void)
{
  while(1)
  {
    DRV_SPI_Tasks(sysObj.spiObjectIdx0);
    vTaskDelay(10);/*Give time back*/
  }
}
```

```
void I2C_ISR(void)
{
    DRV_I2C_Tasks(sysObj.i2cObjectIdx0);
}
```



RTOS – MHC Configuration

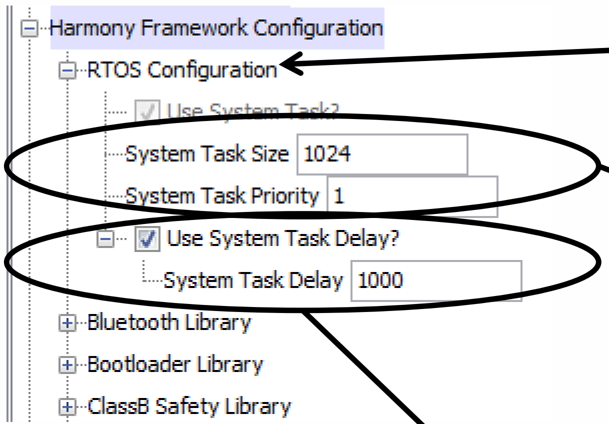


Enable and Select desired RTOS

- ❑ freeRTOS
- ❑ openRTOS
- ❑ uc/os-II
- ❑ uc/os-III
- ❑ Thread X
- ❑ embOS

Enable and Configure RTOS specific options

RTOS – MHC Configuration



RTOS configuration for the SYS_Tasks thread.

```
void SYS_Tasks ( void )
{
   /* Create OS Thread for Sys Tasks. */
   xTaskCreate((TaskFunction_t) _SYS_Tasks,
            "Sys Tasks",
            1024, NULL, 1, NULL);
    .....
}
```

```
static void _SYS_Tasks ( void)
{
   while(1)
   {
      /* Maintain system services */
      SYS_CONSOLE_Tasks(sysObj.sysConsole0);

      /* Maintain Device Drivers */

      /* Maintain Middleware */


      /* Task Delay */
      vTaskDelay(1000 / portTICK_PERIOD_MS);
   }
}
```

Move the task to blocked state for 1000 milliseconds. Allows other tasks to run.

RTOS – MHC Configuration

SPI Driver Instance 0
RTOS Configuration (Instance 0)
Run This Driver Instance As: Standalone
Task Size: 1024
Task Priority: 1
Use Task Delay?
Task Delay: 1000
SPI Module ID: SPI_ID_2
Driver Mode

RTOS configuration for a driver instance configured for polled mode

Standalone – Run the driver task as a separate RTOS thread

```
void SYS_Tasks ( void )
{
   /* Create OS Thread for DRV_SPI Instance 0 Tasks. */

   xTaskCreate((TaskFunction_t) _DRV_SPI_IDX0_Tasks,
            "DRV_SPI Instance 0 Tasks",
            1024, NULL, 1, NULL);
   ....
}
```

```
void _DRV_SPI_IDX0_Tasks(void)
{
   while(1)
   {
      DRV_SPI_Tasks(sysObj.spiObjectIdx0);

      vTaskDelay(1000 / portTICK_PERIOD_MS);
   }
}
```

SPI Driver Instance 0
RTOS Configuration (Instance 0)
Run This Driver Instance As: Combined with System Tasks
SPI Module ID: SPI_ID_2
Driver Mode

```
static void _SYS_Tasks ( void)
{
   while(1)
   {
      /* Maintain system services */
      SYS_CONSOLE_Tasks(sysObj.sysConsole0);

      /* Maintain Device Drivers */
      DRV_SPI_Tasks(sysObj.spiObjectIdx0);

      /* Maintain Middleware */

      /* Task Delay */
      vTaskDelay(1000 / portTICK_PERIOD_MS);
   }
}
```

Combined with System Tasks – Run the driver task as part of the _SYS_Tasks thread

System Initialization & Tasks

```
int main(void)
{
    SYS_Initialize(NULL);

    while(true)
    {
        SYS_Tasks();
    }

    return(EXIT_VALUE);
}
```

```
void SYS_Initialize( void* data )
{
    SYS_CLK_Initialize( &clkInit );
    BSP_Initialize();
    sysObj.spiObjectIdx0 = DRV_SPI_Initialize (DRV_SPI_INDEX_0,
&drvSpi0InitData);
    sysObj.drvTmr0     = DRV_TMR_Initialize(DRV_TMR_INDEX_0,
&drvTmr0InitData);
     ……

   /*create Tasks*/
    APP_Initialize();
```

```
void SYS_Tasks ( void )
{
    /* Create OS Thread for Sys Tasks. */
    xTaskCreate((TaskFunction_t) _SYS_Tasks, "Sys Tasks",1024, NULL, 3, NULL);
    xTaskCreate((TaskFunction_t) _USB_Tasks, "USB Tasks",1024, NULL, 3, NULL);

    /* Create OS Thread for SENSORTASK Tasks. */
    xTaskCreate((TaskFunction_t) _SENSORTASK_Tasks, "SENSORTASK Tasks", 1024, NULL, 2,
NULL);

    /* Create OS Thread for EEPROMTASK Tasks. */
    xTaskCreate((TaskFunction_t) _EEPROM_Tasks, "EEPROM Tasks", 1024, NULL, 1, NULL);
    …….

     /**************
     * Start RTOS *
     **************/
    vTaskStartScheduler(); /* This function never returns. */
```

OS Abstraction Layer

The Operating System Abstraction Layer (OSAL) provides a consistent interface to MPLAB® Harmony Framework components (drivers, middleware, etc.)

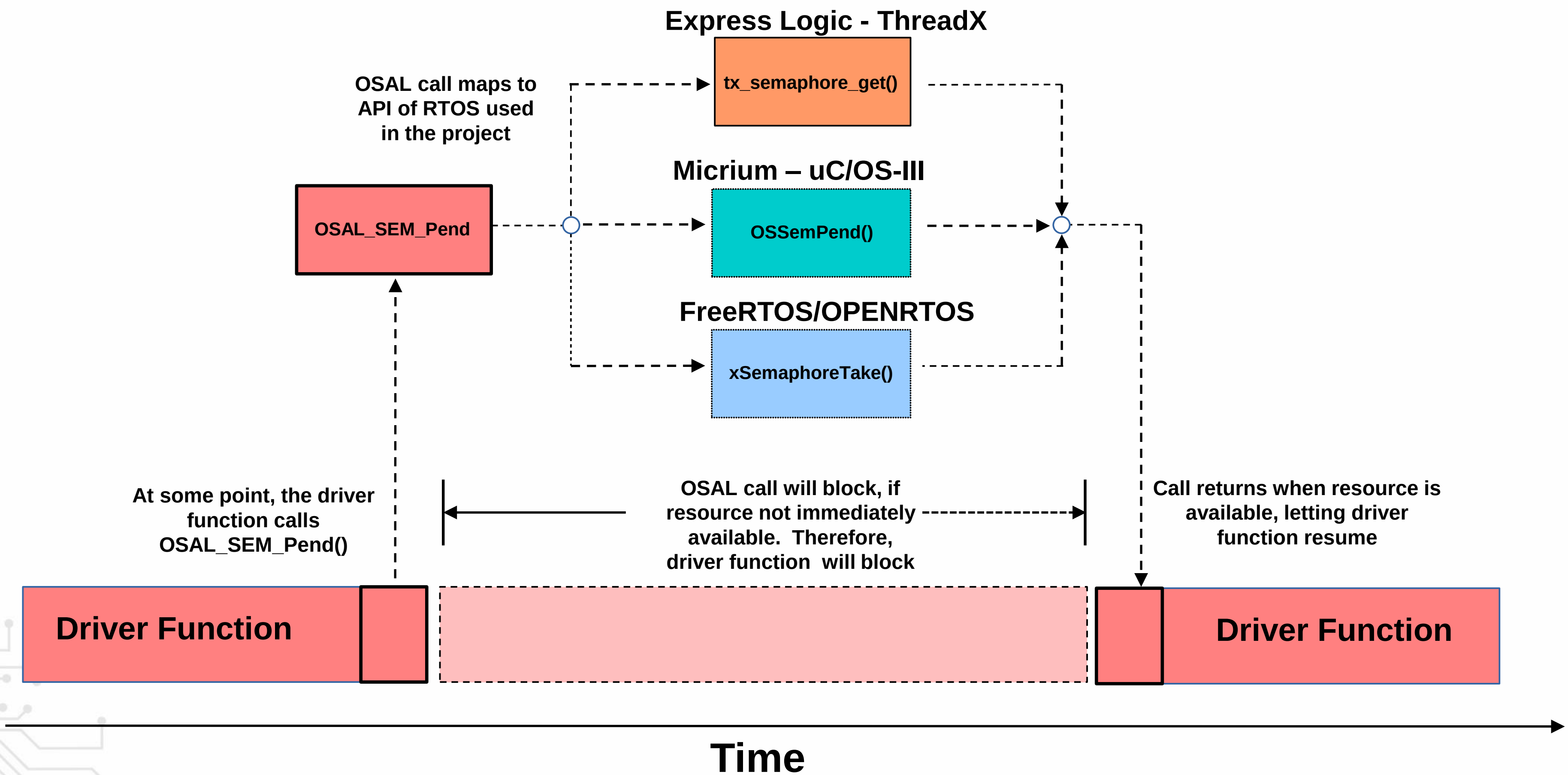
It is designed to allow correct operation of Harmony in an RTOS environment.

Takes care of the underlying differences between the available/supported RTOS Kernels.

OS Abstraction Layer

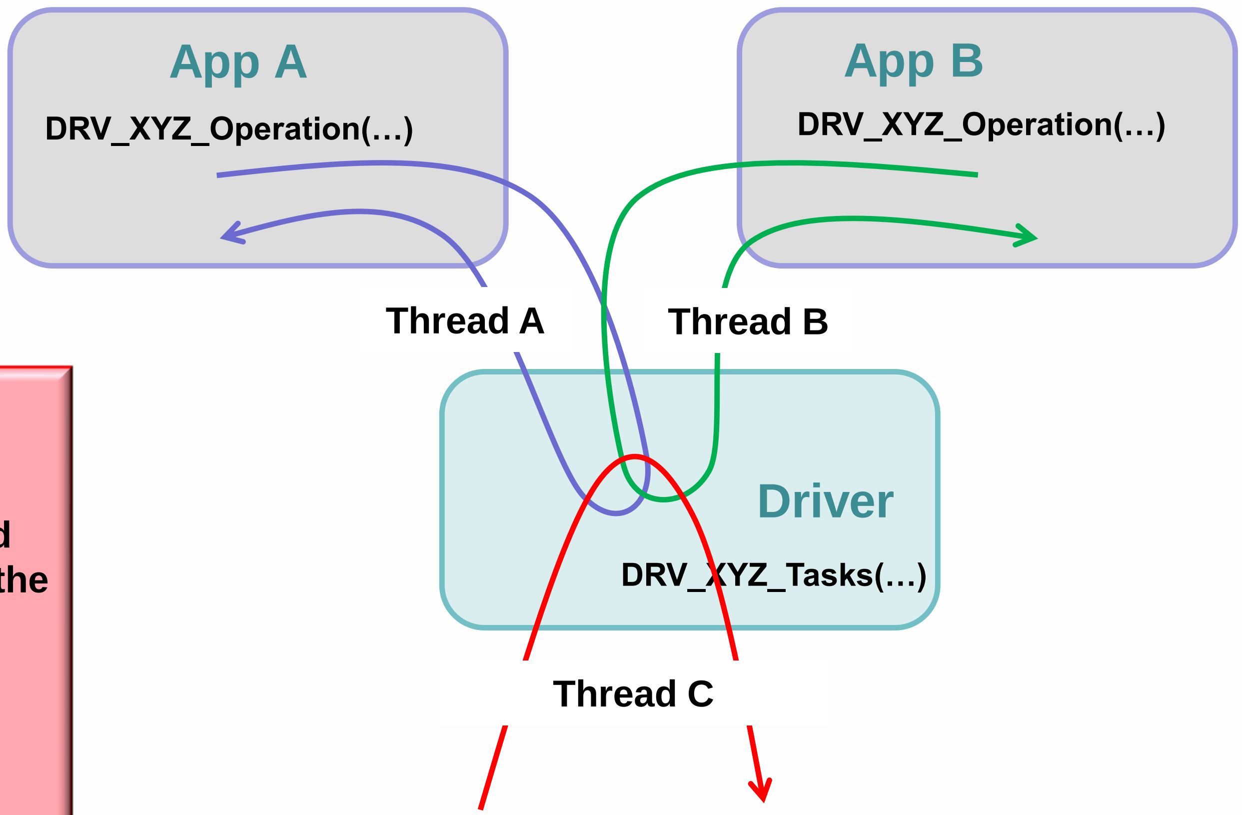
- The design intention is that drivers will use the minimal set of OSAL features necessary to ensure that they can safely operate in a multi-threaded environment yet can also compile correctly when no underlying RTOS is present
- The interface to the OSAL Library is defined in the "osal.h" header file

OSAL Mapping to RTOS



RTOS Thread Safety: Driver in Polled mode

• Driver Running from an RTOS Thread

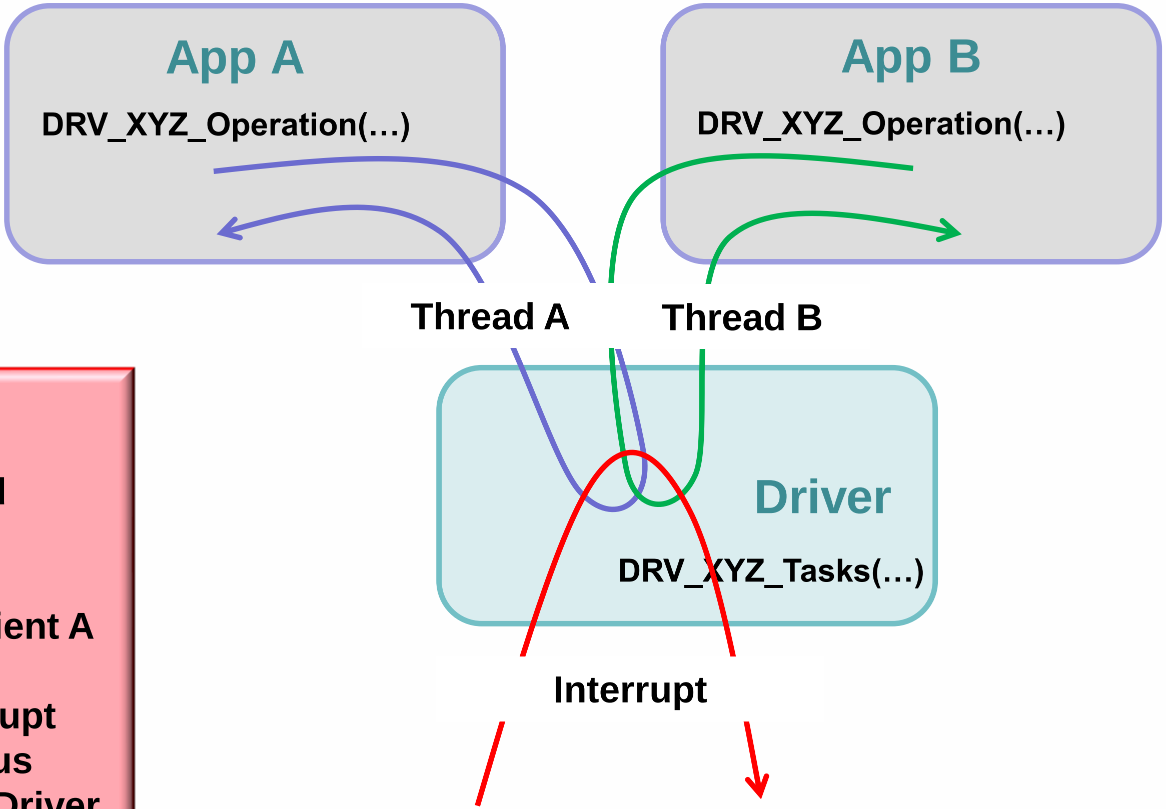


<u>Accessing Shared Resource:</u>

- Mutex
- RTOS allows single thread with the mutex to access the resource
- All other threads are blocked waiting for the mutex to be released

RTOS Thread Safety: Driver in Interrupt mode

• Driver Running from Interrupt



Accessing Shared Resource:

- **Mutex + Disable Associated Interrupt (briefly)**
- **Mutex guards against simultaneous access by Client A and Client B**
- **Disabling Associated Interrupt guards against simultaneous access by a Client and the Driver Task running from ISR**

Lab 3

Add RTOS to the Application

Lab 3: Objectives

Be able to convert the existing non-RTOS based application to run with FreeRTOS using MHC

Be able to configure Application tasks, Driver tasks and System Services to run with FreeRTOS

Lab 3: Summary

In this lab we have...

- Configured and added FreeRTOS to the existing application using MHC
- Configured the Application tasks, Driver tasks and System Services to run with FreeRTOS
- Realized how easy it is to add RTOS support to an existing application, using MHC

Key benefits of MPLAB® Harmony

Improved Cross-Micro Compatibility
- Common and Consistent APIs
- Easier to grow/shrink into larger/smaller parts

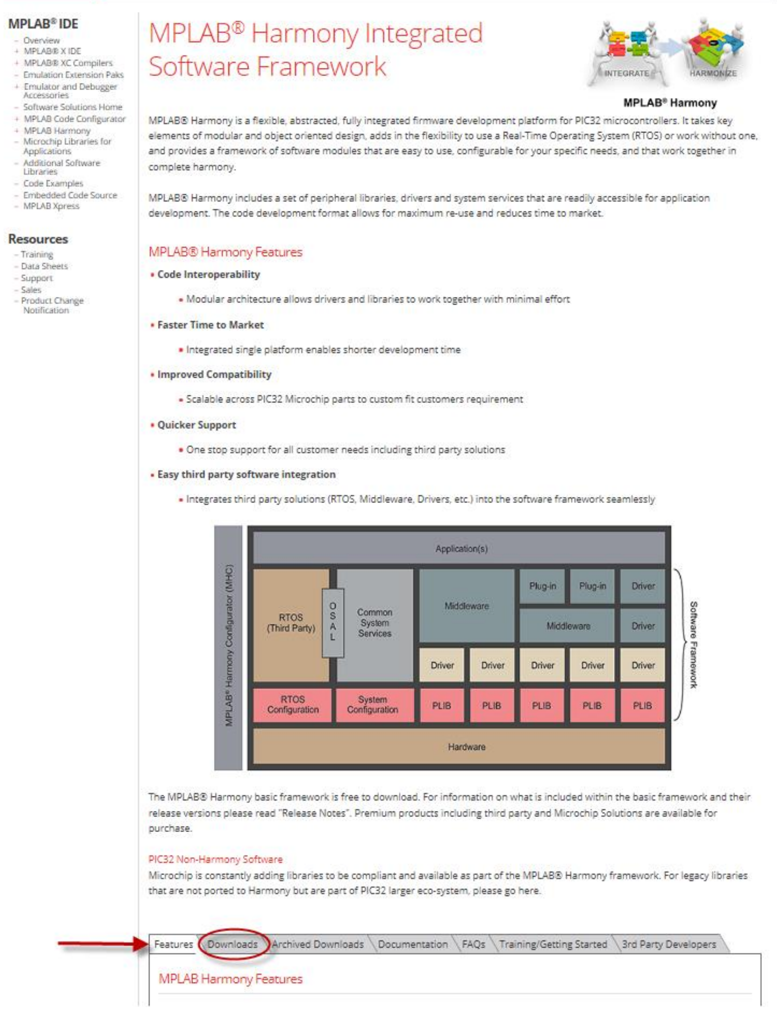
Improved Code Interoperability
- Modules work together with minimal effort

Faster Time to Market
- Able to develop applications and add features quickly

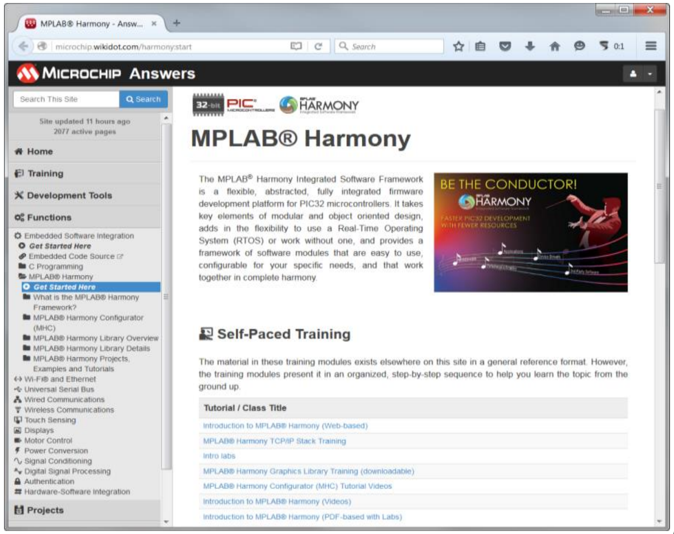
MICROCHIP MASTERs 2017

Web Resources

Download: www.microchip.com/mplab/mplab-harmony

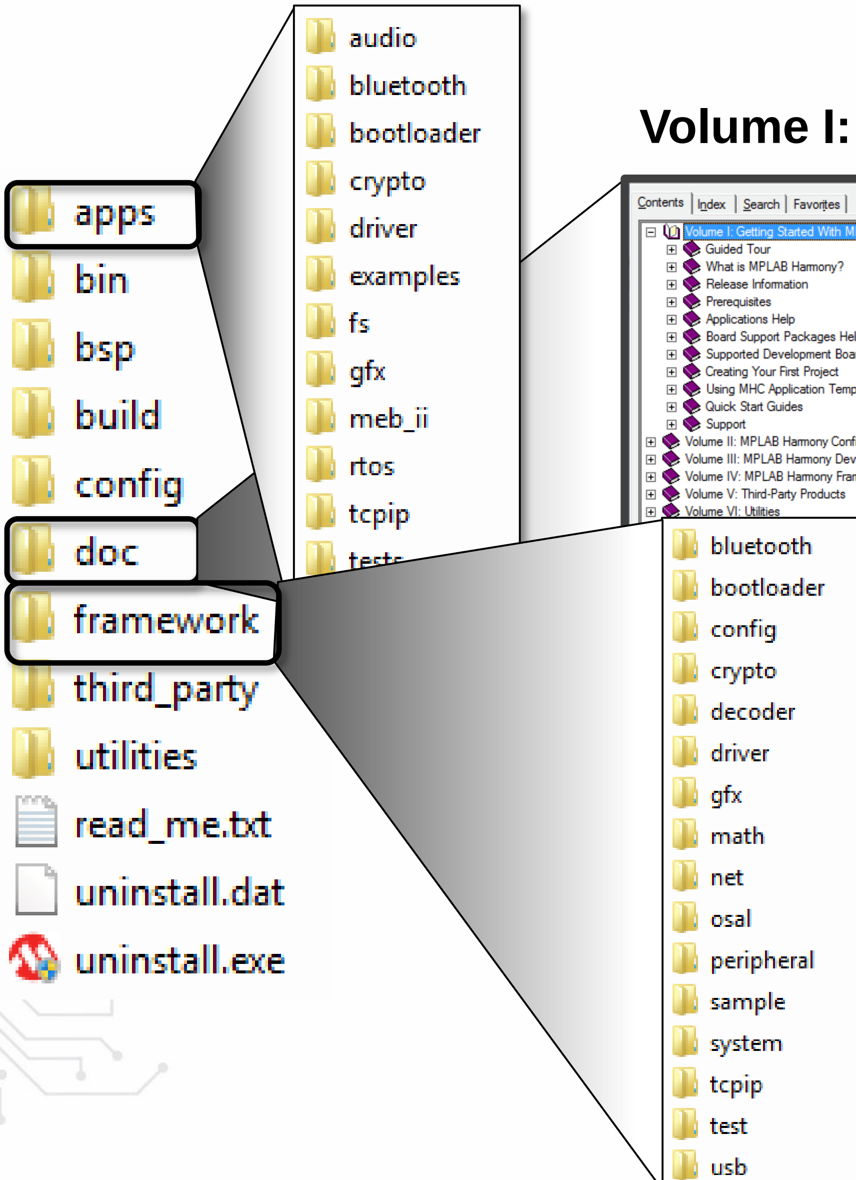


Training: microchip.wikidot.com/harmony:start

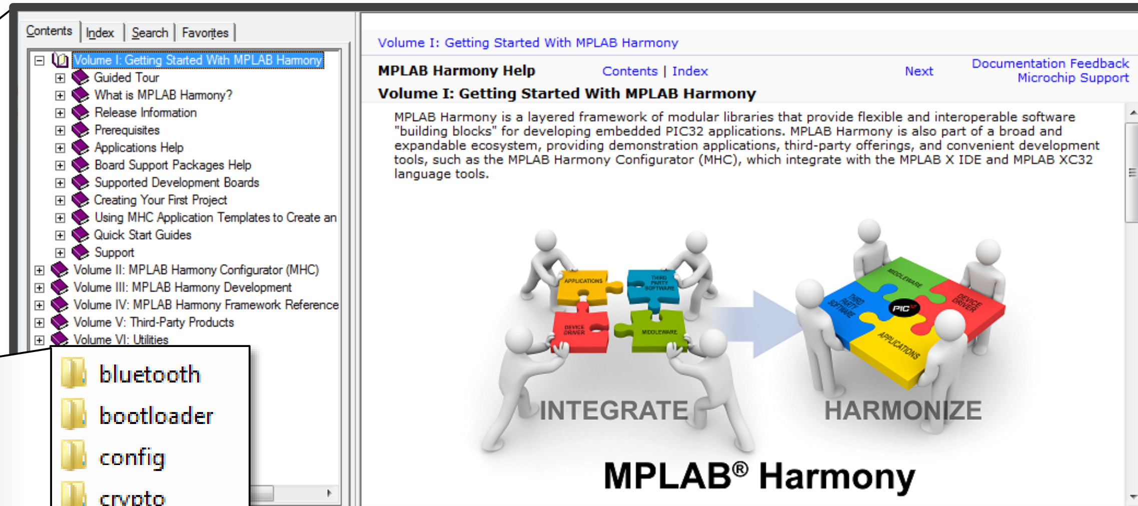


Forum: www.microchip.com/forums

Installed Resources



Volume I: Getting Started With MPLAB® Harmony



Default Installation Directories

Windows: `C:\Microchip\harmony\<version>`

Linux/Mac: `~/microchip/harmony/<version>`

Summary

Class Summary

Today we have covered:

- MPLAB® Harmony key concepts and drivers.
- We looked at the various features supported by Harmony drivers.
- We also learnt how to enable and configure applications to run in an RTOS environment.
- We went through several labs to experience Harmony ourselves.
- Harmony is much more than this!

MPLAB® Harmony Classes

Class Number	Description
21016 DEV5	Creating Simple PIC32 Embedded Applications using MPLAB® Harmony
21017 DEV6	Creating Advanced PIC32 Embedded Applications using MPLAB® Harmony
21032 FRM10	Understanding and Meeting real-time constraints in a MPLAB® Harmony RTOS application
21048 GFX1	PIC32 Graphics Development with MPLAB® Harmony Graphics Composer Suite
21049 GFX2	Developing accelerated graphics applications with Next-Generation High-Performance PIC32MZ Graphics (DA) Family and MPLAB® Harmony
21061 USB6	Developing USB Host and Device Applications with MPLAB® Harmony USB Stack
21070 NET1	Introduction to the MPLAB® Harmony TCP/IP Stack
21036 AD1	5x5x5 RGB LED Cube Design using Harmony with a PIC32, Bluetooth and USB
21067 BLU6	Developing PIC32 Bluetooth Audio and BLE applications with the BM64 Driver in MPLAB® Harmony

Dev Tools For This Class

Hardware Tools:

DM320104 **:**Curiosity PIC32MZEF Development Board

MIKROE-1486 : FRAM Click Board

MIKROE-1978 : Weather Click Board

CAB0028 : Cable, USB A to Micro-B, 6' (2 Nos.)

Software Tools:

MPLAB® X v3.61

MPLAB® XC32 v1.43

MPLAB® Harmony v2.03b

Thank you!

LEGAL NOTICE

SOFTWARE:

You may use Microchip software exclusively with Microchip products. Further, use of Microchip software is subject to the copyright notices, disclaimers, and any license terms accompanying such software, whether set forth at the install of each program or posted in a header or text file.

Notwithstanding the above, certain components of software offered by Microchip and 3rd parties may be covered by "open source" software licenses – which include licenses that require that the distributor make the software available in source code format. To the extent required by such open source software licenses, the terms of such license will govern.

NOTICE & DISCLAIMER:

These materials and accompanying information (including, for example, any software, and references to 3rd party companies and 3rd party websites) are for informational purposes only and provided "AS IS." Microchip assumes no responsibility for statements made by 3rd party companies, or materials or information that such 3rd parties may provide.

MICROCHIP DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, OR STATUTORY, INCLUDING ANY IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT WILL MICROCHIP BE LIABLE FOR ANY DIRECT OR INDIRECT, SPECIAL, PUNITIVE, INCIDENTAL, OR CONSEQUENTIAL LOSS, DAMAGE, COST, OR EXPENSE OF ANY KIND RELATED TO THESE MATERIALS OR ACCOMPANYING INFORMATION PROVIDED TO YOU BY MICROCHIP OR OTHER THIRD PARTIES, EVEN IF MICROCHIP HAS BEEN ADVISED OF THE POSSIBLITY OF SUCH DAMAGES OR THE DAMAGES ARE FORESEEABLE. PLEASE BE AWARE THAT IMPLEMENTATION OF INTELLECTUAL PROPERTY PRESENTED HERE MAY REQUIRE A LICENSE FROM THIRD PARTIES.

TRADEMARKS:

The Microchip name and logo, the Microchip logo, AnyRate, AVR, AVR logo, AVR Freaks, BeaconThings, BitCloud, CryptoMemory, CryptoRF, dsPIC, FlashFlex, flexPWR, Heldo, JukeBlox, KEELOQ, KEELOQ logo, Kleer, LANCheck, LINK MD, maXStylus, maXTouch, MediaLB, megaAVR, MOST, MOST logo, MPLAB, OptoLyzer, PIC, picoPower, PICSTART, PIC32 logo, Prochip Designer, QTouch, RightTouch, SAM-BA, SpyNIC, SST, SST Logo, SuperFlash, tinyAVR, UNI/O, and XMEGA are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

ClockWorks, The Embedded Control Solutions Company, EtherSynch, Hyper Speed Control, HyperLight Load, IntelliMOS, mTouch, Precision Edge, and Quiet-Wire are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Adjacent Key Suppression, AKS, Analog-for-the-Digital Age, Any Capacitor, AnyIn, AnyOut, BodyCom, chipKIT, chipKIT logo, CodeGuard, CryptoAuthentication, CryptoCompanion, CryptoController, dsPICDEM, dsPICDEM.net, Dynamic Average Matching, DAM, ECAN, EtherGREEN, In-Circuit Serial Programming, ICSP, Inter-Chip Connectivity, JitterBlocker, KleerNet, KleerNet logo, Mindi, MiWi, motorBench, MPASM, MPF, MPLAB Certified logo, MPLIB, MPLINK, MultiTRAK, NetDetach, Omniscient Code Generation, PICDEM, PICDEM.net, PICkit, PICtail, PureSilicon, QMatrix, RightTouch logo, REAL ICE, Ripple Blocker, SAM-ICE, Serial Quad I/O, SMART-I.S., SQI, SuperSwitcher, SuperSwitcher II, Total Endurance, TSHARC, USBCheck, VariSense, ViewSpan, WiperLock, Wireless DNA, and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

Silicon Storage Technology is a registered trademark of Microchip Technology Inc. in other countries.

GestIC is a registered trademark of Microchip Technology Germany II GmbH & Co. KG, a subsidiary of Microchip Technology Inc., in other countries.

All other trademarks mentioned herein are property of their respective companies.