ECEN 5823-001 / -001B

Internet of Things Embedded Firmware

Lecture #10

28 September 2017

Agenda

- Class announcements
- SPI tap sensor assignment questions?
- I2C temp sensor assignment
- I2C peripheral
- Writing your own I2C driver / routine
- Si7021 load power management sequence
- Bluetooth Smart

Class Announcements

- Quiz #5 is due at 11:59pm on Sunday, October 1st, 2017
- SPI tap sensor assignment is due at 11:59pm on Wednesday, September 27th, 2017
- I2C temp sensor assignment is due at 11:59pm on Saturday, October 7th, 2017
    - It is now posted on D2L

Update on the SPI tap sensor assignment

7. BMA280 settings should be initialized to:
   a. Range +/- 4g
   b. Bandwidth 125Hz
   c. Tap quiet 30mS
   d. Tap samples 4
   e. Tap duration 200mS
   f. Tap shock 50mS
   g. Tap threshold 250mg

Update on the SPI tap sensor assignment

5. BMA280 functionality:
   a. Upon power on reset or the Blue Gecko reset, the BMA280 should be in SUSPEND mode
   b. Single tap should turn off LED1
   c. Double tap should turn on LED1

SPI tap sensor assignment questions?

- Does anyone have questions on this assignment that I can address right now?

I2C temp Sensor Assignment

ECEN 5823

I2C temp sensor Assignment

Fall 2017

Objective: Adding the Si7021 accelerator via the I2C bus and enabling / disabling the Si7021 to implement load power management.

Note: This assignment will begin with the completed SPI tap sensor assignment.

Due: Saturday, October 7th, 2017 at 11:59pm

Instructions:

1. Make any changes required to the SPI tap sensor assignment.

2. Connect the STK6101C extension board to the main development kit board.

I2C temp Sensor Assignment

Si7021-A20

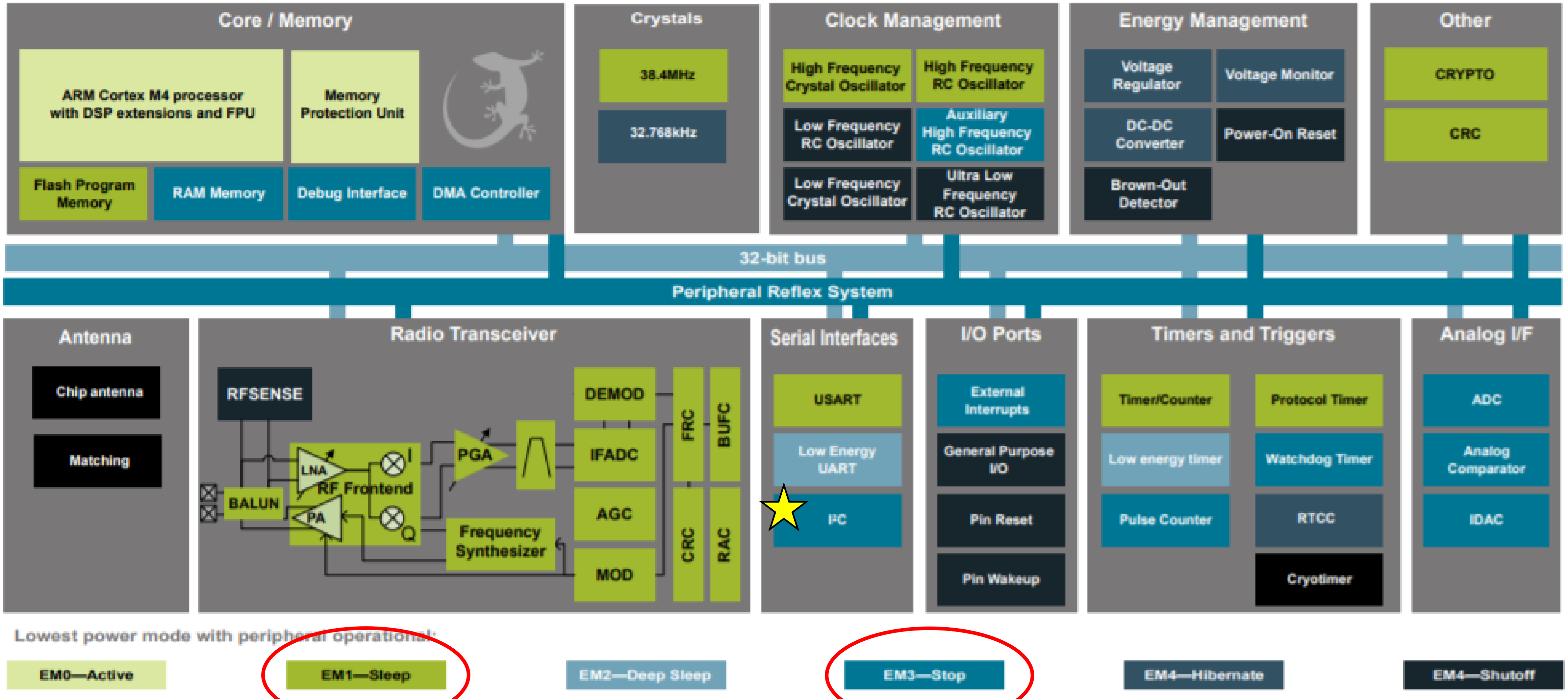
SILICON LABS

I^{2}C HUMIDITY AND TEMPERATURE SENSOR

Features

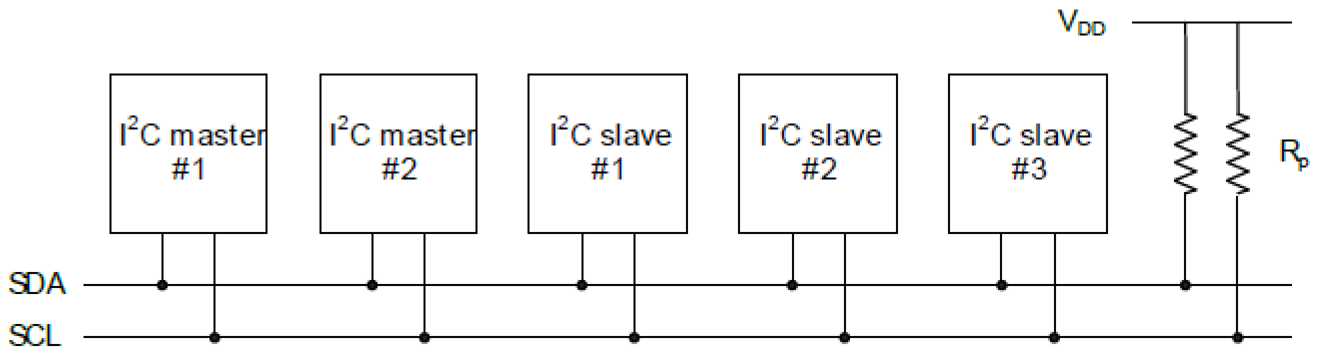
- Precision Relative Humidity Sensor
  - ± 3% RH (max), 0–80% RH
- High Accuracy Temperature Sensor
  - ±0.4 °C (max), –10 to 85 °C
- 0 to 100% RH operating range
- Up to –40 to +125 °C operating range
- Wide operating voltage (1.9 to 3.6 V)
- Low Power Consumption
  - 150 µA active current
  - 60 nA standby current
- Factory-calibrated
- I^{2}C Interface
- Integrated on-chip heater
- 3x3 mm DFN Package
- Excellent long term stability
- Optional factory-installed cover
  - Low-profile
  - Protection during reflow
  - Excludes liquids and particulates

Ordering Information:
See page 29.


Core / Memory
ARM Cortex M4 processor with DSP extensions and FPU
Memory Protection Unit
Flash Program Memory
RAM Memory
Debug Interface
DMA Controller
Crystals
38.4MHz
32.768kHz
Clock Management
High Frequency Crystal Oscillator
High Frequency RC Oscillator
Low Frequency RC Oscillator
Auxiliary High Frequency RC Oscillator
Low Frequency Crystal Oscillator
Ultra Low Frequency RC Oscillator
Energy Management
Voltage Regulator
Voltage Monitor
DC-DC Converter
Power-On Reset
Brown-Out Detector
Other
CRYPTO
CRC
32-bit bus
Peripheral Reflex System
Antenna
Chip antenna
Matching
Radio Transceiver
RFSENSE
BALUN
LNA
PA
RF Frontend
I
Q
PGA
Frequency Synthesizer
DEMOD
IFADC
AGC
MOD
FRC
BUFC
CRC
RAC
Serial Interfaces
USART
Low Energy UART
I²C
I/O Ports
External Interrupts
General Purpose I/O
Pin Reset
Pin Wakeup
Timers and Triggers
Timer/Counter
Protocol Timer
Low energy timer
Watchdog Timer
Pulse Counter
RTCC
Cryotimer
Analog I/F
ADC
Analog Comparator
IDAC
Lowest power mode with peripheral operational:
EM0—Active
EM1—Sleep
EM2—Deep Sleep
EM3—Stop
EM4—Hibernate
EM4—Shutoff

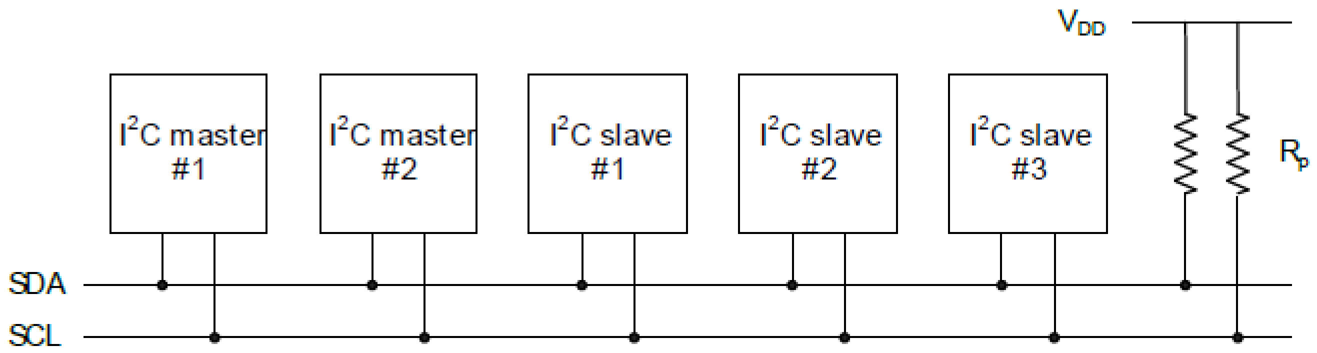
What is I2C?

- The I2C-bus uses two wires for communication
  - A serial data line (SDA)
  - A serial clock line (SCL)
  - It is a true multi-master bus that includes collision detection
  - Arbitration to resolve situations where multiple masters transmit data at the same time without data loss.

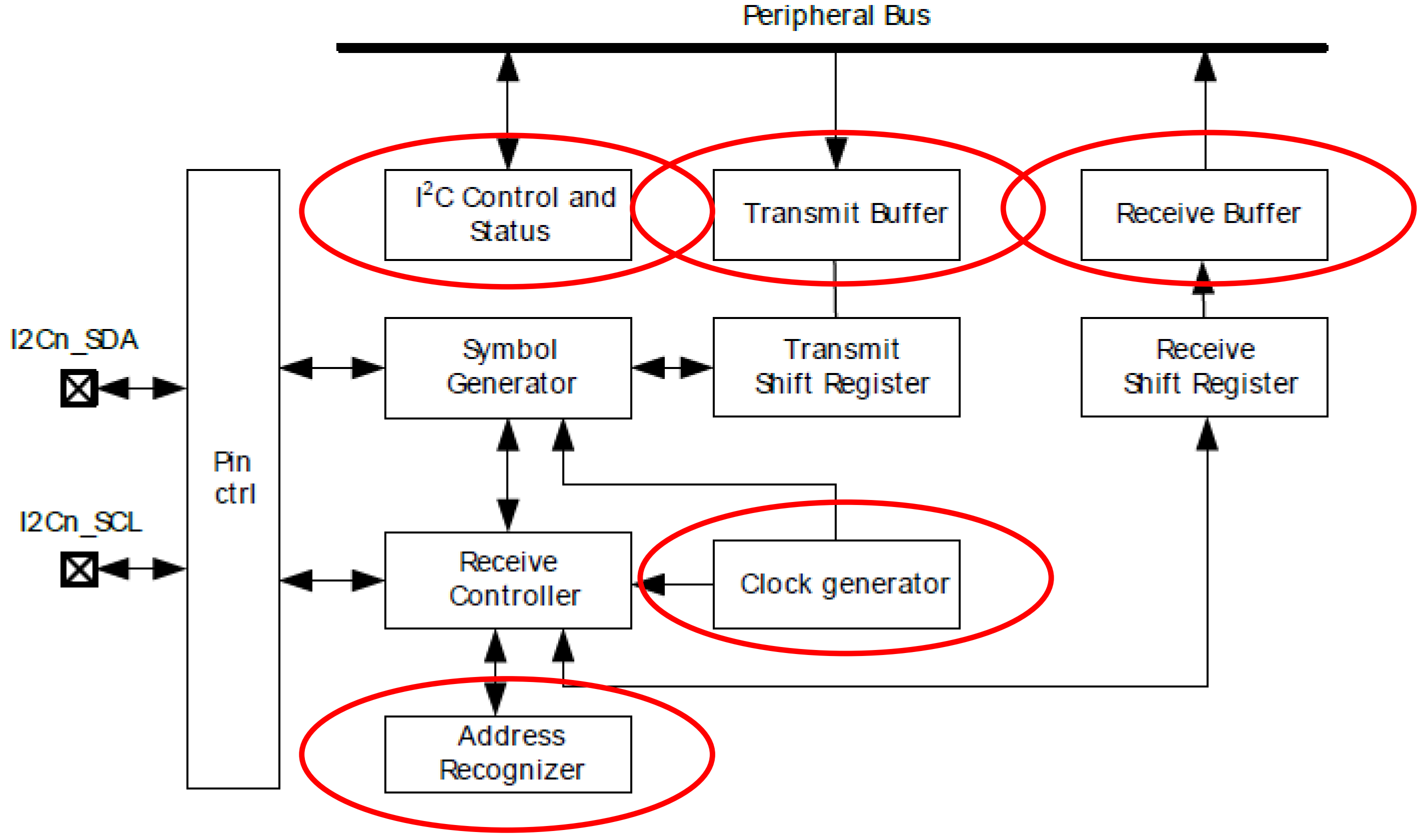


What is I2C?

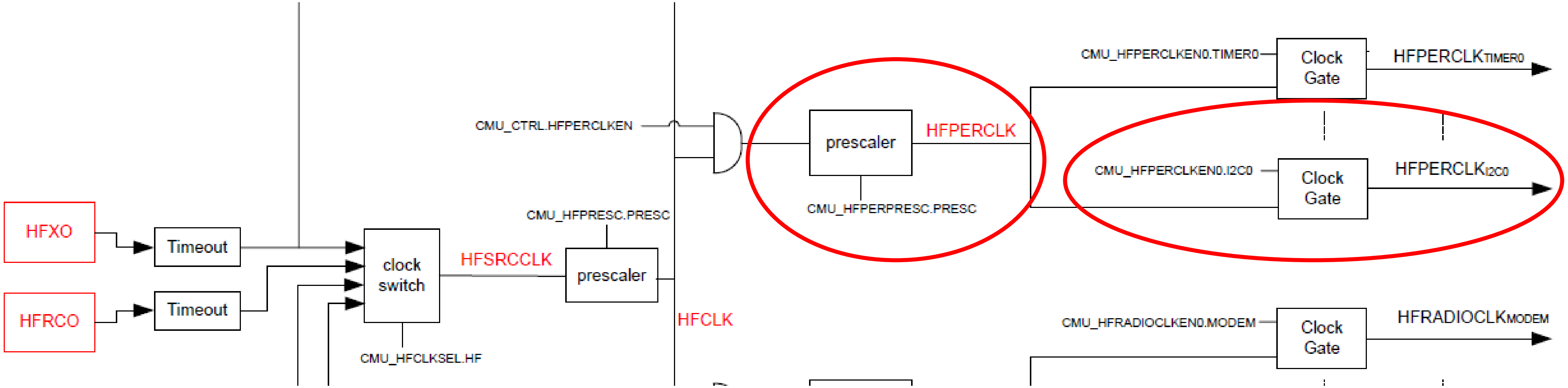
- Each device on the bus is addressable by a unique address
- The I2C master can address all the devices on the bus, including other masters
- Both the bus lines are open-drain. The maximum value of the pull-up resistor can be calculated as a function of the maximal rise-time tr for the given bus speed
- The maximal rise times for 100 kHz, 400 kHz and 1 MHz I2C are 1 µs, 300 ns and 120 ns respectively.



I2Cn peripheral block diagram



I2C



If the I2C clock input is the HFPEFCLK, how can it work done into EM3?

Setting up the I2C

- First, the clock tree to the I2C must be established
  - Without establishing the clock tree, all writes to the I2Cn registers will not occur
  - I2Cn clock source is the HFPERCLK, so no oscillator enable is required, but the HFPERCLK needs to be enabled using CMU_ClockEnable
  - Pseudo code in the CMU setup routine to enable the I2Cn clock tree:
    - Lastly, enable the I2C clocking using the CMU_ClockEnable for the I2Cn

Setting up the I2C

- Second, the I2C must be set up
  - Specify SCL and SDA pins of the I2Cn peripheral
    - Recommend using the GPIO pin mode set up emlib routines
    - Silicon Labs' I2C application note, AN0011, software examples is available resource to insure the GPIO pins are set up correctly

I2Cn routing information

Table 6.1. CSP43 2.4 GHz Device Pinout

CSP Pin# and Name		Pin Alternate Functionality / Description				
Pin #	**Pin Name**	**Analog**	**Timers**	**Communication**	**Radio**	**Other**
A1	VREGSW	DCDC regulator switching node				
A2	VREGVDD	Voltage regulator VDD input				
A3	DECOUPLE	Decouple output for on-chip voltage regulator. An external decoupling capacitor is required at this pin.				
A4	IOVDD	Digital IO power supply.				
A6	PF0	BUSAX BUSBY	TIM0_CC0 #24 TIM0_CC1 #23 TIM0_CC2 #22 TIM0_CDTI0 #21 TIM0_CDTI1 #20 TIM0_CDTI2 #19 TIM1_CC0 #24 TIM1_CC1 #23 TIM1_CC2 #22 TIM1_CC3 #21 LE- TIM0_OUT0 #24 LETIM0_OUT1 #23 PCNT0_S0IN #24 PCNT0_S1IN #23	US0_TX #24 US0_RX #23 US0_CLK #22 US0_CS #21 US0_CTS #20 US0_RTS #19 US1_TX #24 US1_RX #23 US1_CLK #22 US1_CS #21 US1_CTS #20 US1_RTS #19 LEU0_TX #24 LEU0_RX #23 I2C0_SDA #24 I2C0_SCL #23	FRC_DCLK #24 FRC_DOUT #23 FRC_DFRAME #22 MODEM_DCLK #24 MODEM_DIN #23 MODEM_DOUT #22 MODEM_ANT0 #21 MODEM_ANT1 #20	PRS_CH0 #0 PRS_CH1 #7 PRS_CH2 #6 PRS_CH3 #5 ACMP0_O #24 ACMP1_O #24 DBG_SWCLKTCK #0
A7	PF1	BUSAY BUSBX	TIM0_CC0 #25 TIM0_CC1 #24 TIM0_CC2 #23 TIM0_CDTI0 #22 TIM0_CDTI1 #21 TIM0_CDTI2 #20 TIM1_CC0 #25 TIM1_CC1 #24 TIM1_CC2 #23 TIM1_CC3 #22 LE- TIM0_OUT0 #25	US0_TX #25 US0_RX #24 US0_CLK #23 US0_CS #22 US0_CTS #21 US0_RTS #20 US1_TX #25 US1_RX #24 US1_CLK #23 US1_CS #22 US1_CTS #21 US1_RTS #20	FRC_DCLK #25 FRC_DOUT #24 FRC_DFRAME #23 MODEM_DCLK #25 MODEM_DIN #24 MODEM_DOUT #23 MODEM_ANT0 #22 MODEM_ANT1 #21	PRS_CH0 #1 PRS_CH1 #0 PRS_CH2 #7 PRS_CH3 #6 ACMP0_O #25 ACMP1_O #25 DBG_SWDIOTMS #0

Helpful I2C hints from AN0011SW application note

```
/* Initializing I2Cn */
/* Output value must be set to 1 to not drive lines low... We set
*/
/* SCL first, to ensure it is high before changing SDA. */
GPIO_PinModeSet(I2Cn_SCL_Port, I2Cn_SCL_Pin, gpioModeWiredAnd, 1);
GPIO_PinModeSet(I2Cn_SDA_Port, I2Cn_SDA_Pin, gpioModeWiredAnd, 1);
```

Why are the pins set to WiredAnd and not PushPull?

Helpful I2C hints from AN0011SW application note

```
/* Initializing I2Cn */
/* Output value must be set to 1 to not drive lines low... We set */
/* SCL first, to ensure it is high before changing SDA. */
GPIO_PinModeSet(I2Cn_SCL_Port, I2Cn_SCL_Pin, gpioModeWiredAnd, 1);
GPIO_PinModeSet(I2Cn_SDA_Port, I2Cn_SDA_Pin, gpioModeWiredAnd, 1);

/* Toggle I2C SCL 9 times to reset any I2C slave that may require it */
for (int i=0;i<9;i++) {
   GPIO_PinOutClear(I2C1_SCL_Port, I2C1_SCL_Pin);
   GPIO_PinOutSet(I2C1_SCL_Port, I2C1_SCL_Pin);
   }
```

Setting up the I2C

- Second, the I2C must be set up
  - Specify SCL and SDA pins of the I2Cn peripheral
    - Recommend using the GPIO pin mode set up emlib routines
    - Silicon Labs' I2C application note, AN0011, software examples is available resource to insure the GPIO pins are set up correctly
  - Must route the I2C pins to the I2Cn peripheral
    - This can be accomplished by writing the correct location register into I2Cn->ROUTE
  - Need to specify the I2C init Type_Def
    - I2C_Init(I2Cn, &init_Type_Def);

Setting up the I2C

- Third, I2Cn bus must be reset
  - Upon setting up the I2C bus, the bus and its peripherals may be out of synch
  - To reset the I2C bus, the following procedure should be executed:

```
/* Exit the busy state.  The I2Cn will be in this state out of RESET */
if (I2Cn->STATE & I2C_STATE_BUSY){
   I2Cn->CMD = I2C_CMD_ABORT;
}
```

Setting up the I2C

- Forth, the I2C interrupts must be enabled if needed
  - Clear all interrupts from the I2C to remove any interrupts that may have been set up inadvertently by accessing the I2Cn->IFC register or the emlib routine
  - Enable the desired interrupts by setting the appropriate bits in I2Cn->IEN
  - Set BlockSleep mode to the desired Energy Mode
    - Call BlockSleep mode right before accessing the I2C bus
    - The Blue Gecko can be an I2C Master in EM0 & EM1
    - The Blue Gecko can detect its I2C Slave address down into EM3 since the clock is generated from the I2C bus clock SCL
  - Enable interrupts to the CPU by enabling the I2Cn in the Nested Vector Interrupt Control register using NVIC_EnableIRQ(I2Cn_IRQn);

Setting up the I2C

- Fifth, the I2Cn interrupt handler must be included
  - Routine name must match the vector table name:

```
Void I2Cn_IRQHandler(void) {
}
```

  - Inside this routine, you add the functionality that is desired for the I2Cn interrupts

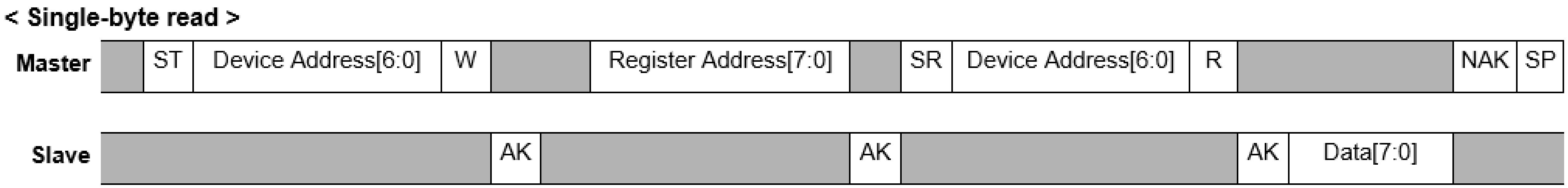
Writing your own I2C driver

- The I2C standard appears to be more of a physical bus standard than a bus protocol
  - Bus protocol in this usage is a defined sequence of operations that could be taken from one device to another with simple port to the specific devices specifications
  - I have found that many I2C devices use the I2C physical bus protocol, but do not easily fit into a standard I2C library

Writing your own I2C driver

- Where to start?
  - Go to the I2C slave's data sheet and find their I2C bus sequence of events diagram

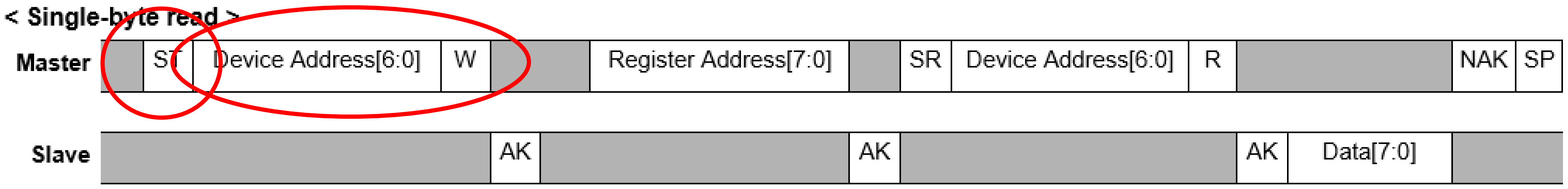
I^2C data sequence diagrams



Writing your own I2C driver

- Now, convert the visual diagram into a driver
- Prime the TX Buffer for the Start Command
  - I2Cn->TXDATA = (I2C_device_addr << 1) | R/W bit = 0 signifying write of address to the slave;
- Now send the Start Bit
  - I2Cn->CMD = I2Cn_CMD_START;

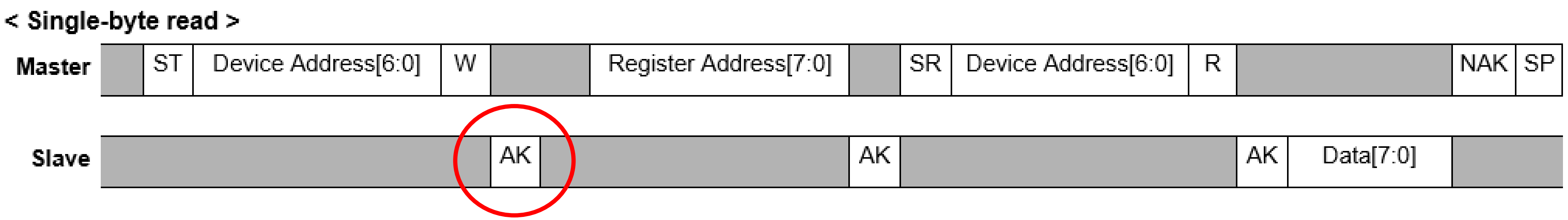
I^2C data sequence diagrams



Writing your own I2C driver

- Now, wait for the slave to respond
  - While ((I2Cn->IF & I2Cn_IF_ACK) == 0);
  - After the ACK has been received, it must be cleared from the IF reg
  - I2Cn->IFC = I2Cn_IFC_ACK;

I^2C data sequence diagrams

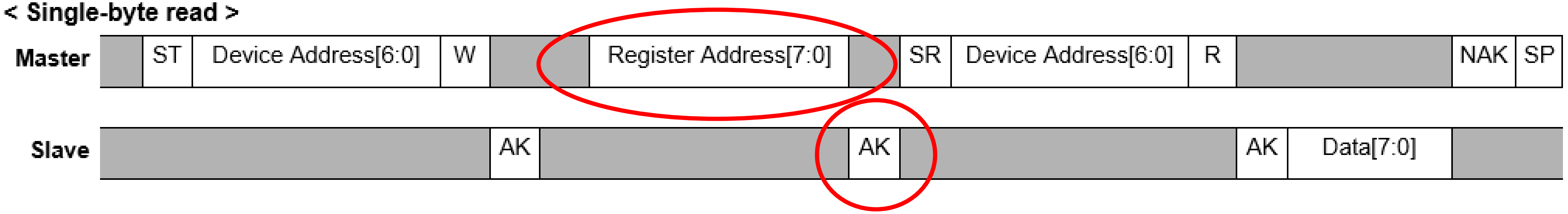


Writing your own I2C driver

- Now, send the I2C device register address
  - I2C->TXDATA = I2C_device_reg_add;
- Now, wait for the slave to respond
  - While ((I2Cn->IF & I2Cn_IF_ACK) == 0);
  - After the ACK has been received, it must be cleared from the IF reg
  - I2Cn->IFC = I2Cn_IFC_ACK;

I^2C data sequence diagrams

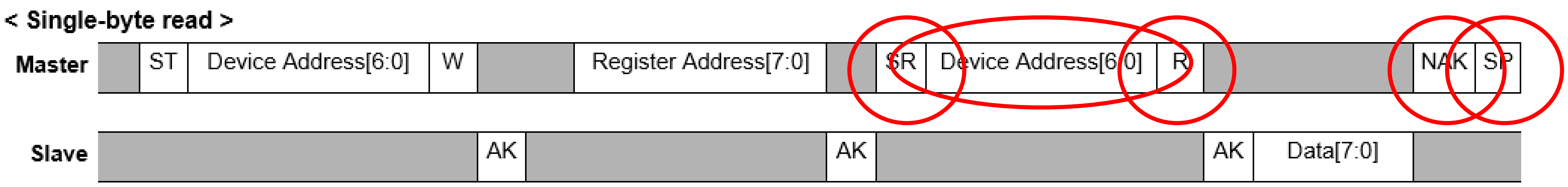
< Single-byte read >



Writing your own I2C driver

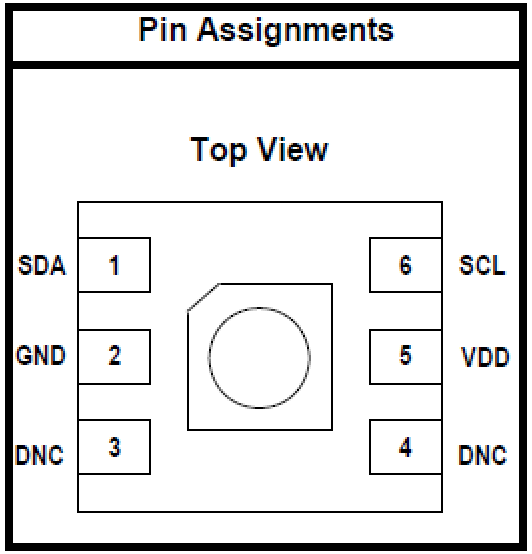
- Your driver continues down through out the visual diagram
  - Device Address
  - SR = Start Repeat
  - R = Read/Write bit set to 1 for Read Operation
  - NAK = NACK
  - SP = STOP

I^2C data sequence diagrams



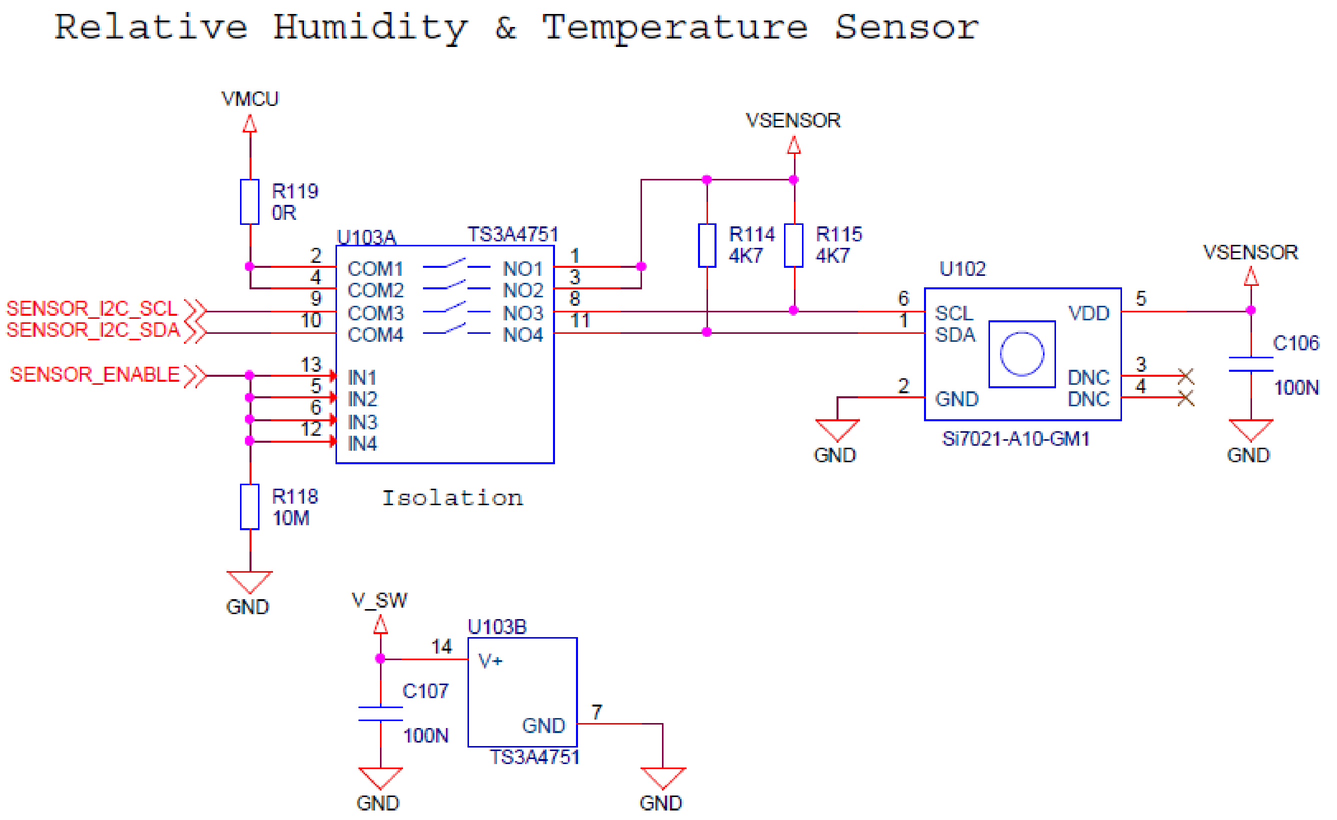
Si7021 and Load Power Management

- What is missing from the package pin assignments?
- With no interrupt, how does the lack of an interrupt change the use of this device in a low energy environment?



Si7021 and Load Power Management

- What does this schematic tell us about Load Power Management?



Si7021 and Load Power Management

- Load Power Management Turning ON sequence
  - Enable power to the Si7021 via the GPIO Sensor_Enable pin
  - Wait for the Si7021 to complete its Power On Rest, POR, and/or SCL and SDA pull ups to ramp up to “high” which ever is the longest period of time
  - Set the SCL and SDA gpio pins to “WiredAND”
  - Sequence SCL 9 times to reset all I2C peripherals on the bus
  - If the Blue Gecko I2C peripheral is busy, abort the operation to reset the I2C peripheral
  - Initialize the Si7021 to match the functionality required
  - Enable the function to request a Si7021 temperature measurement

Si7021 and Load Power Management

- Load Power Management Turning OFF sequence
  - Disable the application function to request a Si7021 temperature measurement
  - Take SCL and SDA off the I2C bus by placing the pins in “Disable” mode
    - Only good practice if there is no other I2C device on the bus. If there was another I2C, the application may still want to access this other I2C device
  - Set to “0” or clear the “Sensor_Enable” pin to turn off power to the I2C pullups and Si7021

BLE: The Attribute Protocol

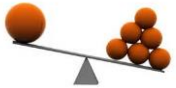
- Defines six types of messages:
  - Requests sent from client to the server
  - Responses sent from the server to the client in reply to request
  - Commands sent from the client to the server that have no response
  - Notifications sent from the server to the client that have no confirmation
  - Indications sent from the server to the client
  - Confirmations sent from the client to the server in reply to an indication
- Communications can be initiated by both the client and the server

BLE: The Attribute Protocol

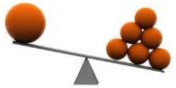
- Attributes are addressed, labeled bits of data
  - Each attribute has a unique handle that identifies that attribute
  - Type that identifies the data stored in the attribute
  - And a value
- For example, an attribute with type Temperature that has a value of 20.5C could be contained within an attribute with the handle 0x01CE
- The Attribute Protocol does not define any attribute types, although it does define that some attributes can be grouped, and their groups can be discovered via the Attribute Protocol
- The Attribute Protocol also defines that some attributes have permissions:
  - To allow a client to read or write an attribute's value
  - Or, to only allow access to the value of the attribute if the client has been authenticated itself or has been authorized by the server
- The Attribute Protocol is mostly stateless
  - Each individual transaction such as a read request and read response does not cause state to be saved on the server
  - The one exception is the prepare and execute write request. These store a set of values that are to be written in the server and then executed all in sequence in a single transaction

BLE: Asymmetric Design

- A major philosophy of the Bluetooth Low Energy Architecture

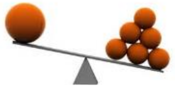
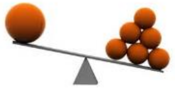
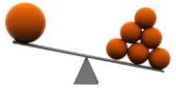


  - Devices with smaller energy sources be given less to do
  - Conversely, devices with larger energy sources be given more to do
- A fundamental assumption is the most resource-constraint device will be the one to which all others are optimized



  - Advertising is less energy consuming than scanning
  - A slave has less energy than a master
    - A master has to manage the piconet timing, the adaptive frequency hopping set, encryption, and many other complex procedures

BLE: Asymmetric Design

- At the Generic Attribute Protocol Layer, the two type of devices are:
  - Client
    - Determines what data the server has and how to use it
    - The client sends request to the server for data
  - Server
    - The Server holds data
    - Similar to the slave at the Link Layer, the server just does what it is told
- The security architecture works on a key distribution scheme by which the slave device gives a key to the master device to remember
  
  - The burden is on the master to remember the bonding information, not the slave

- This implies the most resource-constraint device will want to be the advertisers, slaves, and servers

- Conversely, the devices with the most resources will be the scanners, masters, and clients