C-Coding variables: **[u]int#_t**,
#=Number of bits. **float, double**
Bitwise Operations: & | ^ ~ Logical Operations: && || == != > < >= <=
Math Operators: + - * / % Value Operators: ! << >> ++ --
Functions: <return type> function_name(<arg_type> in1,<arg_type> in2,...)
printf("format string",var1,var2,...) **void** no arguments or return type
printf formats: %u decimal unsigned integer, %d decimal signed integer,
%x or %X Hexadecimal integer, %f float, %c character

0000=0x0 0001=0x1 0010=0x2 0011=0x3 0100=0x4 0101=0x5 0110=0x6 0111=0x7
1000=0x8 1001=0x9 1010=0xA 1011=0xB 1100=0xC 1101=0xD 1110=0xE 1111=0xF

Structs: Access fields through dot:
```
struct.field1 = 10;
struct.field2 = struct.field1
```

Pointers: **<type>*** declares as pointer
&var converts variable to pointer
```
function(in1,&in2); // pass in2 as pntr
```

GPIO DriverLib
uint8_t **GPIO_getInputPinValue**(uint8_t port,uint8_t pins)
void **GPIO_setOutputLowOnPin**(uint8_t port,uint8_t pins)
void **GPIO_setOutputHighOnPin**(uint8_t port,uint8_t pins)
void **GPIO_toggleOutputOnPin**(uint8_t port,uint8_t pins)
void **GPIO_setAsOutputPin**(uint8_t port,uint8_t pins)
void **GPIO_setAsInputPin**(uint8_t port,uint8_t pins)
void **GPIO_setAsInputPinWithPullUpResistor** ..or..
void **GPIO_setAsInputPinWithPullDownResistor**
(uint8_t port,uint8_t pins)
void **GPIO_setAsPeripheralModuleFunctionInputPin** ..or..
void **GPIO_setAsPeripheralModuleFunctionOutputPin**
(uint8_t port,uint8_t pins,uint8_t mode)

mode=GPIO_z_MODULE_FUNCTION,z=PRIMARY,SECONDARY,TERTIARY

Possible ports:
GPIO_PORT_Px
Possible pins:
GPIO_PINy

x=1..11, y=0..7

Multiple pins can be used at once by bitwise OR of the desired pins

Debouncing:
__delay_cycles(#)

void **GPIO_enableInterrupt/GPIO_disableInterrupt**(uint8_t port,uint8_t pins)
void **GPIO_interruptEdgeSelect**(uint8_t port,uint8_t pins,uint8_t edgeSelect)
edgeSelect=GPIO_LOW_TO_HIGH_TRANSITION or GPIO_HIGH_TO_LOW_TRANSITION
void **GPIO_registerInterrupt**(uint8_t port,<function_name>)
uint16_t **GPIO_getEnabledInterruptStatus**(uint8_t port)
returns bitwise OR of pins that triggered interrupt (eg. GPIO_PIN1|GPIO_PIN3)
void **GPIO_clearInterruptFlag**(uint8_t port,uint8_t pins)

ADC14 DriverLib *<initialization of ADC14 will not be tested>*
void **ADC14_setResolution**(uint32_t resolution) resolution=ADC_xBIT x=8,10,12,14
bool **ADC14_toggleConversionTrigger**(void) bool = 1 or 0 (True/False)
bool **ADC14_isBusy**(void) memorySelect=ADC_MEMx, x=0..31
uint16_t **ADC14_getResult**(uint32_t memorySelect)

Single-Ended: $N_{ADC} = 2^{N_{bits}} \frac{V_{in+} - V_{R-}}{V_{R+} - V_{R-}}$ $V_{R-} \le V_{in+} < V_{R+}$

Differential: $N_{ADC} = 2^{N_{bits}-1} \frac{V_{in+} - V_{in-}}{V_{R+} - V_{R-}} + 2^{N_{bits}-1}$ $V_{R-} \le V_{in+}, V_{in-} < V_{R+}$

PID: x_d desired output, x_m measured output, y ctl. signal, y_o ctl. offset

Continuous PID $y(t) = y_o + k_p\epsilon(t) + k_d\dot{\epsilon}(t) + k_i \int_0^t \epsilon(T)\,\mathrm{d}T$ $\epsilon(t) = x_d(t) - x_m(t)$

Discrete PID $y(k) = y_o + k_p\epsilon(k) + k_d\left(\epsilon(k) - \epsilon(k-1)\right) + k_i \sum_{n=0}^{k} \epsilon(n)$ $\epsilon(k) = x_d(k) - x_m(k)$

Timer_A DriverLib timer=TIMER_An_BASE n=0..3

Configuration struct **Timer_A_[M]ModeConfig** fields: **[M] = Up,UpDown,Continuous**

.clockSource = TIMER_A_CLOCKSOURCE_x
x=EXTERNAL,ACLK,SMCLK,INVERTED_EXTERNAL_TXCLK

.clockSourceDivider = TIMER_A_CLOCKSOURCE_DIVIDER_y
y=1,2,3,4,5,6,7,8,10,12,14,16,20,24,28,32,40,48,56,64

.timerInterruptEnable_TAIE = TIMER_A_TAIE_INTERRUPT_ENABLE / _DISABLE

.captureCompareInterruptEnable_CCR0_CCIE *= TIMER_A_CCIE_CCRO_INTERRUPT_ENABLE*

.timerPeriod *= 0 to 65535 (Sets value of CCR0) / _DISABLE*

.timerClear = TIMER_A_v_CLEAR,TIMER_A_v_CLEAR v=DO,SKIP

void **Timer_A_configure[M]Mode**(uint32_t timer,Timer_A_[M]ModeConfig * config)

void **Timer_A_startCounter**(uint32_t timer,uint16_t timerMode)
timerMode=TIMER_A_UP_MODE,TIMER_A_UPDOWN_MODE,TIMER_A_CONTINUOUS_MODE

void **Timer_A_stopTimer**(uint32_t timer)

void **Timer_A_clearTimer**(uint32_t timer)

uint16_t **Timer_A_getCounterValue**(uint32_t timer)

$$DutyCycle[\%] = 100\frac{T_{PW}}{T_{timer}} = 100\frac{N_{PW}}{N_{timer}}$$

$$f_{TCLK} = \frac{f_{source}}{divider}, N_{timer} = 1 + CCR0, T_{timer/PW} = N_{timer/PW}T_{TCLK}, f_{timer} = \frac{f_{TCLK}}{N_{timer}}, \Delta t = \frac{N_{cap}(k) - N_{cap}(k-1) + 2^{16}N_{ovf}}{f_{TCLK}}$$

Capture/Compare Modes ccr=TIMER_A_CAPTURECOMPARE_REGISTER_n n=0..4

Configuration struct **Timer_A_[C]ModeConfig** fields: **[C]=Capture,Compare**

.[c]Register = ccr **[c]=capture,compare**

.[c]InterruptEnable = TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE / _DISABLE

.[c]OutputMode = TIMER_A_OUTPUTMODE_z z=OUTBITVALUE,SET,RESET,SET_RESET,
RESET_SET,TOGGLE,TOGGLE_SET,TOGGLE_RESET

.compareValue = 0-65535

.captureMode = TIMER_A_CAPTUREMODE_a
a=NO_CAPTURE,RISING_EDGE,FALLING_EDGE,RISING_AND_FALLING_EDGE

.captureInputSelect = TIMER_A_CAPTURE_INPUTSELECT_b b=CCIxA,CCIxB,GND,Vcc

.synchronizeCaptureSource = TIMER_A_CAPTURE_SYNCHRONOUS / _ASYNCHRONOUS

void **Timer_A_init[C]**(uint32_t timer,Timer_A_[C]ModeConfig * config)

uint16_t **Timer_A_getCaptureCompareCount**(uint32_t timer,uint16_t ccr)

void **Timer_A_setCompareValue**(uint32_t timer,uint16_t ccr,uint16_t value)

void **Timer_A_registerInterrupt**(uint32_t timer,uint8_t intSel,<func.name>)
intSel = TIMER_A_CCR0_INTERRUPT,TIMER_A_CCRX_AND_OVERFLOW_INTERRUPT

uint32_t **Timer_A_getEnabledInterruptStatus**(uint32_timer)
Returns either TIMER_A_INTERRUPT_PENDING or TIMER_A_INTERRUPT_NOT_PENDING

void **Timer_A_clearInterruptFlag**(uint32_t timer)

uint32_t **Timer_A_getCaptureCompareEnabledInterruptStatus**(uint32_t timer, ccr)
Returns 0,TIMER_A_CAPTURECOMPARE_INTERRUPT_FLAG / _OVERFLOW

void **Timer_A_clearCaptureCompareInterrupt**(uint32_t timer, uint16_t ccr)

Inter-Integrated Circuit, I²C, DriverLib module=EUSCI_Bn_BASE n=0..3

Configuration struct **eUSCI_I2C_MasterConfig** fields:

.selectClockSource = EUSCI_B_I2C_CLOCKSOURCE_SMCLK / _ACLK

.i2cClk = 0..4294967295 ← ClockSource rate for calculation of dataRate

.dataRate = EUSCI_B_I2C_SET_DATA_RATE_x x=1MBPS,400KBPS,100KBPS

.byteCounterThreshold = 0..255 (# bytes/packet, 0 to disable)

.autoSTOPGeneration = EUSCI_B_I2C_NO_AUTO_STOP (other values omitted)

void **I2C_initMaster**(uint32_t module,eUSCI_I2C_MasterConfig * config)

void **I2C_enableModule**(uint32_t module)

Void **I2C_writeData**,**I2C_readData**(uint32_t module,uint8_t PeriphAddress,
uint8_t StartRegister, uint8_t * data, uint8_t length)

Write: START-ADDR+W(0)**-*ACK*-StartReg-*ACK*-DataByte-*ACK*-DataByte-*ACK*-..-*ACK*-STOP**

Read: START-ADDR+R(1)**-*ACK*-*DataByte*-ACK-*DataByte*-ACK-*DataByte*-ACK-..-NACK-STOP**