Andi Kleissner
Eli Michael
Nina Joshi
Elico Teixeira

Team MadHatters Psuedo Code

MAIN MODULE:

Function: main(void)
Initialize timers
Initialize ports
Initialize SPI communication
Initialize PWM
Initialize LEDs
Initialize PID output captures
Enable interrupts after all inits
Initialize the state machine and loop forever to run it.
Initialize port data directions for outputs and inputs, initialize pwms, and set opto output low

Function: MAIN_Init(void)
Set outputs and clear inputs
Turn punch power on

ARM MODULE:

Function: ARM_Punch(unsigned char height)
Set punch servo to specified degree by setting the duty cycle to the servo
If height is short then set to one degree
If height is tall then set to the tall degree pwm duty cycle

BEACON MODULE:

Function: BEACON_CheckForBeacon()
Returns a BEACON_ON event if the phototransistor signal is between the specified thresholds. Otherwise, will return BEACON_OFF event. Will only return event if different from previous
If BEACON_BIT is higher than BEACON_ON_LOW_THRESHOLD and lower than BEACON_ON_HIGH_THRESHOLD, BEACON_ON
else BEACON_OFF
if current event is different from lastEvent,
set lastEvent = current event
return current event
return NO_EVENT

TAPE MODULE:

Function: TAPE_CheckForTape(void)
Returns a TAPE_FOUND event if the tape sensor signal greater than the specified threshold

Set the Tape0/1/2 module-level variable as ON or OFF based on the threshold set in the def.h file
Check any change in tape state and return TAPE_CHANGED or NO_EVENT

Function: TAPE_FollowTape(void)
if Tape 0 OFF, Tape 1 ON, and Tape 2 OFF... full drive forward
else if Tape 0 ON, Tape 1 ON, Tape 2 OFF... full drive forward
else if Tape 0 ON, Tape 1 OFF, Tape 2OFF... full drive forward
else if Tape 0 OFF, Tape 1 ON, Tape 2 ON... full drive forward
else if Tape 0 OFF, Tape1 OFF, Tape 2 ON... full drive forward
else if Tape 0 ON, Tape 1 ON, Tape 2 ON... need to initiate rotation, so set rotateflag to 1 until it is complete and put the bot in rotation mode, as well as start the timer to time the rotation
else if Tape 0 OFF, Tape 1 ON, Tape 2 OFF and rotateflag is set... clear rotateflag
else if Tape 0 OFF, Tape 1 OFF, Tape 2 OFF and rotate flag is clear (no tape seen)…just go 50% forward

STATE MODULE:

Function: STATE_InitStateMachine(void)
state = STOP
start checkpoint timer
turn off red led

Function: STATE_RunStateMachine(Event_t event)
Figure out which state we are in, and call the corresponding function to handle the event
if state is STOP, call StopState(event)
if state is SEARCHING, call SearchingState(event)

Function: StopState(Event_t event)
if event is CP_TIMER_EXPIRED, rotate clockwise and state = SEARCHING

Function: SearchingState(Event_t event)
if event is BEACON_ON, set red led, stop wheels, and state = STOP

Function: STATE_InitStateMachine(void)
state = STOP
start checkpoint timer

Function: STATE_RunStateMachine(Event_t event)
Figure out which state we are in, and call the corresponding function to handle the event
if state is STOP, call StopState(event)
if state is PUNCHING, call PunchingState(event)

Function: StopState(Event_t event)
if event is CP_TIMER_EXPIRED, set punch height timer and adjust punch height
if event is PUNCH_HEIGHT_TIMER_EXPIRED, stop punch height, set punch timer, punch out, state = PUNCHING
if event is PUNCH_TIMER_EXPIRED, stop punch

Function: PunchingState(Event_t event)
if event is PUNCH_TIMER_EXPIRED, stop punch, punch in, set punch timer, state = STOP

Function: STATE_InitStateMachine(void)
turn punch power on
state = WAITING;

Function: STATE_RunStateMachine(Event_t event)

Figure out which state we are in, and call the corresponding function to handle the event
if state is WAITING, call WaitingState(event)
if state is MOVING_TO_TARGET_0, call MovingToTarget0State(event)
if state is PUNCHING, call PunchingState(event)
if state is MOVING_TO_TARGET_7, call MovingToTarget7State(event)

Function: WaitingState(Event_t event)
if event is NEW_TARGET, state = MOVING_TO_TARGET_0

Function: MovingToTarget0State(Event_t event)
move to target 0
once target is reached, punch arm, set punch timer, and state = PUNCHING

Function: PunchingState(Event_t event)
if event is PUNCH_TIMER_EXPIRED, reset punch
if event is NEW_TARGET, turn punch power off and state = MOVING_TO_TARGET_7

Function: MovingToTarget7State(Event_t event)
move to target 7
once target reached, SPI request beacon 7 to be illuminated
if event is BEACON_ON, update LED display accordingly
if event is BEACON_OFF, update LED display accordingly

Function: STATE_InitStateMachine(void)
Initialize state machine by setting initial state
state = WAITING

Function: STATE_RunStateMachine(Event_t event)
Figure out which state we are in, and call the corresponding function to handle the event
if state is WAITING, call WaitingState(event)
if state is MOVING, call MovingState(event)
if state is PUNCHING, call PunchingState(event)

Function: WaitingState(Event_t event)
if event is NEW_TARGET,
if first time entered, get target and set side accordingly for navigation and leds set end timer
get current target from SPI_WhichTarget();
NAV_SetTarget(curTarget)
state = MOVING
if event is RETURN_HOME
set current target to home
NAV_SetTarget(curTarget)
state = MOVING
if event is GAME_OVER_RED_WINS,
respond accordingly
if event is GAME_OVER_GREEN_WINS,
respond accordingly
if event is DELAY, break
if event is END_TIMER_EXPIRED,
update led display

Function: MovingState(Event_t event)
if event is REACHED_TARGET
if another target is in the moving queue and current target is not home
ARM_Punch(NAV_GetTargetHeight())
set punch timer

state = PUNCHING
else if current target is not home
ARM_Punch(NAV_GetTargetHeight())
set punch timer
state = PUNCHING
else state = WAITING;
if event is RETURN_HOME,
NAV_EmergencyStop()
set current target to home
NAV_SetTarget(curTarget)
state = MOVING
if event is NEW_TARGET,
get current target from SPI_WhichTarget()
set flag that another target is in the moving queue
if event is GAME_OVER_RED_WINS,
NAV_EmergencyStop()
respond accordingly
state = WAITING
if event is GAME_OVER_GREEN_WINS,
NAV_EmergencyStop()
respond accordingly
state = WAITING
if event is DELAY,
NAV_EmergencyStop()
state = WAITING
if event is END_TIMER_EXPIRED,
NAV_EmergencyStop()
update led display
state = WAITING
else, NAV_MoveToTarget()

Function: PunchingState(Event_t event)
if event is PUNCH_TIMER_EXPIRED,
Poise arm
if new target in punching queue,
NAV_SetTarget(curTarget)
clear punching queue
state = MOVING
else if new target in moving queue,
NAV_SetTarget(curTarget)
clear moving queue
state = MOVING
else, state = WAITING
if event is NEW_TARGET,
get current target from SPI_WhichTarget()
set that new target in moving queue
if event is RETURN_HOME,
Reset arm
set current target to home
NAV_SetTarget(curTarget);
state = MOVING
if event is GAME_OVER_RED_WINS,
Reset arm
respond accordingly
state = WAITING
if event is GAME_OVER_GREEN_WINS,

reset arm
respond accordingly
state = WAITING
if event is DELAY,
poise arm
state = WAITING;
if event is END_TIMER_EXPIRED,
poise arm
update led display

EVENT MODULE:

Function: EVENTS_CheckForEvents(void)
Check if a timer expired
Check if end timer expired
Check if command given
Check if navigation event has occurred
Check if beacon event has occurred
Check if the tape sensors have seen a change

Function: CheckForTimerExpired(void)
Checks all timers and if any has expired, clear the timer and return the corresponding event
if any timer is expired, clear timer
return that timer has expired
else return no event

SPI MODULE:

Module Level Variables:
a structure that contains the Target Commander response and tower number
the command request sent to the target commander
last reply from the target commander

Initialize SPI system:
set prescale and scale for baud rate
enable SPI system
set for master
set MSB first
set polarity as active low
set clock phase to sample even edges
give SPI control of SS pin
enable SPI interrupt

Interrupt: Initialize OC timer
turn timer channel on
set timer clock
set first output compare
clear the OC flag
enable timer interrupt

Function: Timer interrupt
clear the timer flag
update for the next OC
read the SPI status register
read the SPI data register

send command request to target commander

Function: SPI interrupt
static char ping
static char read
if this is the first byte transfer (ping = 1)
 set ping = 0
 read the status register
 read the data register
 send empty byte to target commander
if this is the second byte to be received
 set ping = 1
 read the status register
 read the data register store it as read
 switch read on commands that do not include tower number store reply as reply type
 switch on commands that include tower number
  isolate 4 MSB to find reply, store as reply type
  isolate 4 LSB to find tower and store as tower
 display the received tower number on the LED display

To send different requests to the target commander
if request active target set module variable command = AA
if request illuminate beacon, set command as 0xEx with x being tower number
if request beacon status, set command as 0xBx with x being tower number

Function: SPI_CheckForCommand(void)
store current reply as newreply
if there has been a change in reply or tower since lastreply
store newreply as lastreply
return the reply event
else return no event

Function: SPI_WhichTarget(void)
return the last reply's tower

PWM MODULE:

Initialize the PWM system:
For motors enable PWM channel for Left and right motors
Enable port U pins for PWM control
Set the clock and period to give 10khz PWM
Set duty cycle to 0
For servo: enable PWM channel for punch servo
Set clock and period for 50Hz
Set duty cycle to 0
Enable port U pin for PWM control

To set PWM duty cycle:
Set the corresponding duty register

To set direction:
Change the port bit that controls the non-PWM half bridge.

LED MODULE:

To set the side to red side raise red side LED pin, lower greenside LED pin
To set the side to green side lower red side LED pin, raise greenside LED pin

Function: PulseCLK(void)
Set the clk bit on the shift register to high
Do a us delay loop
Set the clk bit low

To display character in LED display:
For each character there is a corresponding byte for each bit in the character, raise or lower the data bit as appropriate pulse the clk bit

PID MODULE:

Module Level Variables:
uPeriod6 - used to keep track of the period passed for each encoder tick on the left wheel
uPeriod7 - used to keep track of the period passed for each encoder tick on the right wheel
RequestedDuty6 - to save the output of the requested duty cycle that PID will control (left wheel)
RequestedDuty7 - same as above but for right wheel
TargetRPM6 - to save the input from the interface to request an RPM for the left wheel
TargetRPM7 - same as above but for right wheel
RPMError6 - error term for PID control (left wheel)
SumError6 - sum error term for PID control (left wheel)
RPMError7 - same as above but for right wheel
SumError7 - same as above but for right wheel
EncoderTicksLeft - to count the total encoder ticks that occured since last reset (left wheel)
EncoderTicksRight - same as above but for right wheel
RightDirect - to save the direction of the right wheel
LeftDirect - to save the direction of the left wheel

Function: GetEncoderTicks(void)
return EncoderTicksLeft

Function: ClearEncoderTicks(void)
set EncoderTicksLeft to zero

Function: PID_UpdateRPM_Direction(char RightRPM, unsigned char RightDirect1, char LeftRPM, unsigned char LeftDirect1)
Set TargetRPM6 equal to LeftRPM and same for TargetRPM7
Set RightDirect and LeftDirect equal to the input variables

Function: PID_InitTimer1_OC5_IC6_IC7(void)
Turn the timer system on
Set pre-scale selection to 64 = 2.667 uS per clock tick, or 0.375MHz, 174.76ms overflow
Set up OC5 on Timer1 to time the screen updates to every 20 us:
Set TIM1_IOC5 of the timer to be output compare, the rest remain as inputs
No pin connected to TIM1_IOC5, which means pin PT5 remains free
Set the first output capture to happen one "Period" into the future
Clear the flag for the TIM1_IOC5
enable the interrupt for TIM1_IOC5
Set up TIM1_IOC6 and TIM1_IOC7 to be input captures that capture on FALLING edges
Set IC6&IC7 to respond to FALLING edges.
Clear IOC6 flag just in case it has been set on startup

Enable IOC6 as an interrupt
Clear IOC7 flag just in case it has been set on startup
Enable IOC7 as an interrupt
Then enable overall interrupts

Function: interrupt _Vec_tim1ch6 EncoderPeriodTimer6 (void)
Declare uLastEdge6 as a static variable
Calculate time elapsed since last interrupt
Update the new edge time to be the last edge time
Clear the flag for this interrupt!!
Keeping track of direction:
Check channel B of the encoder for the RIGHT wheel, if it is high then
we should be moving forward, and increment, if low then we decrement

Function: interrupt _Vec_tim1ch7 EncoderPeriodTimer7 (void)
Identical to interrupt _Vec_tim1ch6 EncoderPeriodTimer6 except everything for
the right wheel

Function: interrupt _Vec_tim1ch5 UpdateDutyCycles (void)
This interrupt will control the update period for setting our PWM duty cycles on the two motors that drive our wheels. Here is where the meat of the PID implementation lies. This assumes an update period of 20ms to control the motor RPM:
First clear the flag for this interrupt
Update the value of our comp4 for our clck with rolling over considered
Capture what "this period" was so we don't read it while it is being updated
Capture what "this period" was so we don't read it while it is being updated
Enable overall interrupts
Perform the PID calculations:
if EncoderTicksRight is equal to LastEncoderTicksRight then set RPMFloat6 to zero
else set it to some factor divided by uPeriod6
Then do the same if statements for RPMFloat7
Then set RPMError equal to TargetRPM minus RPMFloat and SumError equal to SumError plus RPMError for both 6 and 7
Then RequestedDuty is equal to the pGain times RPMError plus iGain times SumError
Then implement anti-windup for left and right wheels:
If the RequestedDuty is above 100 then set RequestedDuty to 100 and subract out the error that was just added in for both 6 and 7
If TargetRPM6 or 7 is zero, then reset all variables to be zero
Set the duty cycles for the corresponding wheels
Update the LastEncoderTicksRight and Left variables

PIDStateMachine MODULE:

Module Level Variables:
PID_Action_t PID_State - To keep track of the state we are in
Final_Distance - To keep track of the distance that we are going towards
trackreturndistnear - to keep track of whether we have reported the distance is near in the event checker

TypeDefs:
PID_Event_t
    NO_NEW_EVENT,
    DISTANCE_CLOSE,
    FINAL_DISTANCE_REACHED,
    FRONT_LF_LB_BUMPER,
    REAR_LF_LB_BUMPER,

LF_LB_BUMPER,
FRONT_LF_BUMPER,
FRONT_LB_BUMPER,
REAR_LF_BUMPER,
REAR_LB_BUMPER,
FRONT_BUMPER,
REAR_BUMPER,
LF_BUMPER,
LB_BUMPER,
NO_BUMPER

PID_Return_t
DISTANCE_REACHED,
CORNER_REACHED,
CORNER_TURN_COMPLETE,
ACCEPTED,
BUSY,
ERROR

PID_Action_t
FORWARD_CMD,
REVERSE_CMD,
FORWARD_TO_CORNER,
REVERSE_TO_CORNER,
CLOCKWISE,
COUNTERCLOCKWISE,
READY

Function: PID_Stop_Clear_Machine
if PID_State is any of the movement commands in a specified direction, then
Stop both wheels and return the current direction
else if they are turning
stop both wheels
else return 0

Function: PID_Set_Machine(PID_Action_t Action, int Dist)
This function only responds if it is in the ready state
if it is in some other state it will return BUSY. If you want to
stop it you need to use the function PID_Stop_Clear_Machine()
If in ready state, we are not working, so we should update to our new requested state
and set the new things to do (the action and the distance/degrees):
If the state is ready then
Update our new state to the task being asked for
then if pid state is a moving command
reset the trackreturndistnear variable
call the clear encoder ticks function
and set the Final_Distance variable to Dist
then set a variable functionreturn equal to PID_Machine and return its output

Function: PID_Machine(void)
Set PID_event equal to PID_CheckForEvents function to check for events
Then call functions depending on what state we are in and the functions should set the new state
internally depending on the event taken in, so use a switch statement that switches on
FORWARD_CMD, REVERSE_CMD, FORWARD_TO_CORNER, REVERSE_TO_CORNER,
CLOCKWISE, COUNTERCLOCKWISE
and calls their corresponding function

Function: PID_CheckForEvents(void)
Check all the bumpers for the possible combinations by calling the check for bumpers function
if event is not equal to NO_NEW_EVENT then return the event
Check where we are with distance (DISTANCE_CLOSE or FINAL_DISTANCE_REACHED)
if event is not equal to NO_NEW_EVENT then return the event
else return NO_NEW_EVENT

Function: CheckFor_Distance(void)
if trackreturndistnear is false and encodertickcount is greater than Final_Distance minus
some specified amount then
set trackreturndistnear to true and return DISTANCE_CLOSE
else if the encodertickcount is greater than Final_Distance
return DISTANCE_REACHED
else return NO_NEW_EVENT

Function: CheckFor_Bumpers(void)
Implement a debouncing technique that will simply return out of the function if
a certain amount of time has not passed since the last bumper check
Use if statements to check all the possible combinations of bumpers and to
return the event type of that combo of bumpers, i.e. FRONT_LF_LB_BUMPER, or LB_BUMPER

Function: MoveForward(PID_Event_t Event)
Use a switch statement to switch on what event we are seeing
Once we see DISTANCE_CLOSE we should set the flag
Once we see FINAL_DISTANCE_REACHED, reset closeyet flag, update speed to zero and reset state
as well as return DISTANCE_REACHED and turn off the wheels
in the case of seeing a bumper combination we are interested in, set the flag true
otherwise set the flag false for all other bumper combinations
Look for front left bumper (LF) switch to be engaged, if it isnt then
then veer slightly left until it is; the combos for the LF switch to get engaged are
LF, LF+LB, or Rear+LF+LB, Rear+LF
Now if closeyet flag is zero and our wall bumperflag is true
we should start going straight at max speed to reach the distance
else if the closeyet flag is zero then
we should move forward while veering left at a medium speed until the above situation occurs
If the closeyet flag is set to one and the wallbumperflag is true
we want to move forward slowly with the LF engaged and go straight
else if closeyet flag is set to one we should go at our slow speed and veer to engage it

Function: MoveReverse(PID_Event_t Event)
This function behaves exactly the same as the MoveForward function except that when PID is called for movement the wheels should be turning in reverse (although the left wheel still is the one to slow down for us to veer into the wall)

Function: MoveForwardToCorner(PID_Event_t Event)
Using a switch statement based on the event that we have taken in
Once we see DISTANCE_CLOSE we should set the flag
Once we see FINAL_DISTANCE_REACHED, reset closeyet flag, update speed to zero and reset state
as well as return DISTANCE_REACHED and turn off the wheels
in the case of seeing a bumper combination we are interested in, set the flag true
otherwise set the flag false for all other bumper combinations
Look for front left bumper (LF) switch to be engaged, if it isnt then
then veer slightly left until it is; the combos for the LF switch to get engaged are
LF, LF+LB, or Rear+LF+LB, Rear+LF

Now if closeyet flag is zero and our wall bumperflag is true we should start going straight at max speed to reach the distance
else if the closeyet flag is zero then
we should move forward while veering left at a medium speed until the above situation occurs
If the closeyet flag is set to one and the wallbumperflag is true we want to move forward slowly with the LF engaged until we see one of the front bumper combos engaged
else if closeyet flag is set to one we should go at our slow speed and veer to engage it

Function: MoveReverseToCorner(PID_Event_t Event)
This function behaves exactly the same as the MoveForward function except that when PID is called for movement the wheels should be turning in reverse. We should also only stop momvement and return distance reached in the case that the back bumper becomes engaged once our distance close flag has become true.

Function: RightCornerTurn_GoingForward(PID_Event_t Event)
Switch on the event to set the correct bumper flags each time we enter this function
so we know what bumper flags we are looking to be true in certain situations
if just entered is true and we see our front bumper is set, then make sure wheels are not turning
however we are in the just entered phase but see a LF+LB combo then
we can move forward while veering slightly to the wall
else if we just entered but we dont see that we are engaged in the corner, or an LF+LB combo
we should back up to align with the wall (veer towards the wall), now if we no longer just entered, then we want to back up for a short time, and subsequently start the rotation
using flags to make sure that things are carried out in the correct order
so start reversing and start our timer reverse
once we have reached the end fo the timer reverse set the we are positioned to begin our rotaiton flag
then start the timer going again and put one wheel on so that we rotate
once the timer expires then we know we have completed our rotation
if we have completed our rotation then we should reset all the flags to their original values
and return CORNER_TURN_COMPLETE

Function: LeftCornerTurn_GoingReverse(PID_Event_t Event)
We perform the exact same actions as RightCornerTurn_GoingForward function with the exact same flags except now we move in the reverse direction for all movements.

NAV MODULE:

Module Level Variables:
where_we_at – char used to keep track of where we are
where_we_going – char used to update where we are going
virtual_going – char used to track where we are going (differentied from where_we_going because we choose to switch variables when we are on the green side
flag – a flag variable to keep track of which movement we are currently calling from the PID module
justenteredconvert – type char also used as a flag to track when we should convert variables
which_side_we_on – to keep track of being on the red or green side

Function: NAV_MoveToTarget(void)
Call MoveToTargetOuput
If the output of the MoveToTargetOutput is reached
    Then set the module level variable eventcheck_nav to the certain event
    Else set it as no_event

Function: NAV_CheckNavEvent(void)

Initialize a function level variable called lasteventcheck_nav
If eventcheck_nav is not equal to lasteventcheck_nav
    Then return eventcheck_nav
    Else return no_event

Function: NAV_GetTargetHeight(void)
If where_we_at is equal to 2, 5, a or d
    Then return short
    Else return tall

Function: NAV_SetTarget(unsigned char Target)
Set where_we_going equal to Target

Function: NAV_Setside(unsigned char side_nav)
Set which_side_weon equal to side_nav

Function: NAV_MoveToTargetOutput(void)
If which_side_we_on is green, then use a switch statement to flip the variables of the board across a horizontal axis
Using a switch statement on where_we_at, for each case call a different function GoingFromX, where GoingFromX contains all the directions going from that specific case to the where_we_going variable. The return from GoingFromX should also be captured here and returned to the function at the end. At this point we should also set virtual_going to where_we_going for cases 0 to 7, and use the switchthetarget() function based on where_we_going for cases 8-f to flip the variable on the board across a vertical axis.

Function: NAV_exit(void)
This function, used as an exit function upon completion of the discrete movements that are called from PID, is called everytime we reach the position we should be at.
Set where_we_at to current
Set where_we_going to w
Set flag to be zero
And set justenteredconvert to false

Function: GoingFromX(void) (X ranges from 0 to 7)
This function should use a switch statement and which takes as its variable the virtual_going variable. Based on this case, it should call the appropriate PID movement (either turn a corner, move in a certain direction for a specified amount, or move to a corner) and use flags to keep track of which movement is being called and to keep the movements in the correct order.
Switch virtual_going
Case 1: directions from X to beacon 1
Case 2: directions from X to beacon 2
Etc. up until beacon f, and then to home.
Symmetry is then used to get from any quadrant to any other target on the board.

TIMER MODULE:

Module Level Variables:
timer0 (unsigned int) used to keep track of Timer overflows by using the Input Capture
Period (unsigned int) based on the 43.69mS overflow, or 1.5 MHz clock
flag0_X (of type Flags_t) where X ranges from 0 to 7 used to keep track of whether the time has expired for each of the four timers
flagact0_X (of type Flags_t) where X ranges from 0 to 7 flags used to keep track of whether the time has expired for each of the four timers

NewTime0_X (unsigned int) where X ranges from 0 to 7 and the new time keeps track of the time each of the four timers will be checking to be expired.

Function: TIMER0_Init(unsigned char NewRate)
This function turns timer system on and sets the timer to count at a certain rate using the TIMER_RATE_XX definitions:
Turn the timer system on
Set TSCR2: divide by 16, so timer overflow occurs every 43.69mS
Use a switch statement to set the timer period, where 1mS = 1500, 2mS = 3000, 4ms = 6000, etc. and we set the variable Period equal to this number.
Setup OC4 to time the updates, and to trigger the first interrupt that will keep track of these times
Set IOC4 of the timer to be an output compare, the rest remain as inputs
Set no pin connected to IOC4, which means pin PT0 remains free
Set the first output capture to happen one "Period" into the future
Clear the flag for the IOC4
Enable the interrupt for IOC4
Enable interrupts

Interrupt Response for Timer IOC4:
This interrupt will be keeping track for timer0 channel 4, and will be setting flags for timer0 on channel 4-7 which will allow the TIMER0_InitTimer(TimerNumber, TicksToCount) function to let the user know if the timer is expired:
interrupt _Vec_tim0ch4 Timer0Counter (void)
Clear the flag for the IOC4
Update for the next interrupt to keep track of the ticking rate
EnableInterrupts
Add one to timer0 to indicate that one clock tick has passed by
Now we check to see if any of the flags have expired and update:
    If timer0 is equal to newtime0_x and flagact0_X is active then
    Set flag0_X to set
    Set flagact0_X to not active
Repeat this if statement for all the X timers, 0 through 7

Function: TIMER0_InitTimer
This function will take in two paramaters, the first a char Num which will choose which timer we are starting/restarting, and the second an unsigned int NewTime which will give the number of ticks until that timer is expired. Everytime this function is called, that timer will start counting up to the the number of NewTime ticks that it is asked for. The user must be sure that this NewTime of ticks does not pass the overflow otherwise this will be inaccurate.
    TIMER0_InitTimer(unsigned char Num, unsigned int NewTime)
    Use a switch statement to choose which timer that the user wants by the timer number and be sure to set the NewTime for that timer as well as to clear its flag, then for Case X (0 through 7):
    Set NewTime0_X equal to timer0 plus the input New Time;
    Set flag0_X to be cleared
    Set flagact0_X to be active

Function: TIMER0_IsTimerExpired
Checks to see if the the timer associated with the parameter input Num has expired (which clock, 0-7), the number of the timer to test. This function will return TIMER0_EXPIRED or TIMER0_NOT_EXPIRED, or TIMER0_ERR
TimerReturn_t TIMER0_IsTimerExpired(unsigned char Num)
Use a switch statement to choose which timer that the user wants to check and then return on the basis of whether it has expired or not, so for each case X:
If flag0_X is set then
    Return timer0 is expired

Else if flag0_X is cleared
    Return timer0 is not expired
Else
    Return timer error

Function: TIMER0_ClearTimerExpired
This takes as a paramater an unsigned char Num which chooses which timer flag that we want to clear. This can be used to show that an event has been serviced.
TIMER0_ClearTimerExpired(unsigned char Num)
Again we use a switch statement to choose which timer that the user wants to clear and clear the according flag, so for each case X:
Set flag0_0 to be cleared

Function: TIMER0_GetTime
This function takes in no parameters, and will return an unsigned int representing the current clock count which will be between 0 and 65535. It simply returns the free-running counter of timer0.
unsigned int TIMER0_GetTime(void)
Return the value for timer0