|  |  |
| --- | --- |
|  | |
|  | |
| EG3205 Assignment 2 | |
| **Module code:** | **EG3205** |
| **Module name:** | **Programming Microelectronic and Multi-Core Systems** |
|  | |
|  | |
|  | |
| **Author(s):** | **Nishat Tarannum** |
| **Student ID(s):** | **169018358** |
| **Degree:** | **MEng Software Engineering** |
| **Tutor/Project supervisor:** | **Professor Tanya Vladimirova** |
|  | |
|  | |

By submitting this report for assessment I confirm that this assignment is my own work, is not copied from any other person's work (published or unpublished), and that it has not previously been submitted for assessment on any other module or course.

I am aware of the University of Leicester’s policy on plagiarism, and have taken the online tutorial on avoiding plagiarism.  I am aware that plagiarism in this project report may result in the application of severe penalties up to and including expulsion from the University without a degree.

Contents

[Introduction 2](#_Toc533154854)

[Master Qsys Design 3](#_Toc533154855)

[Slave Qsys Design 4](#_Toc533154856)

[Description of Software Developed for Master Node 5](#_Toc533154857)

[3.1)............... 5](#_Toc533154865)

[a) 5](#_Toc533154866)

[b) . 6](#_Toc533154867)

[c) Master and Slave Node Tasks 7](#_Toc533154870)

[d)................................................................................................................................................................... 8](#_Toc533154871)

[e) 9](#_Toc533154873)

[3.2) 9](#_Toc533154874)

[a) 9](#_Toc533154875)

[b) 10](#_Toc533154876)

[i) Startup 10](#_Toc533154877)

[ii) Normal Operations: no Button is pressed 10](#_Toc533154878)

[iii) Master Button0 Press 10](#_Toc533154879)

[iv) Slave Button0 Press 10](#_Toc533154880)

[c)................................................................................................................................................................. 10](#_Toc533154881)

[Conclusion 14](#_Toc533154886)

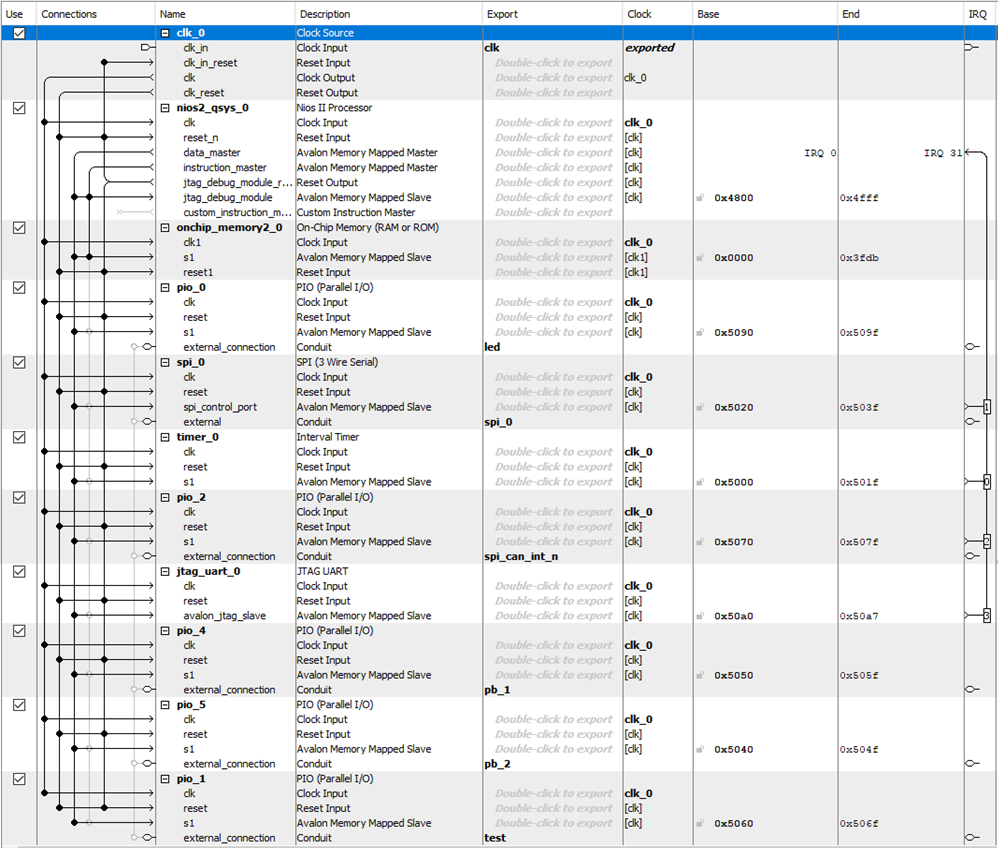
# Introduction

Using work completed in the previous two labs, the software of a shared-clock network design connecting a single-processor master node and a dual-processor slave node had to be implemented over a CAN bus. This dual-core processor design was intended for use in safety-critical embedded applications.

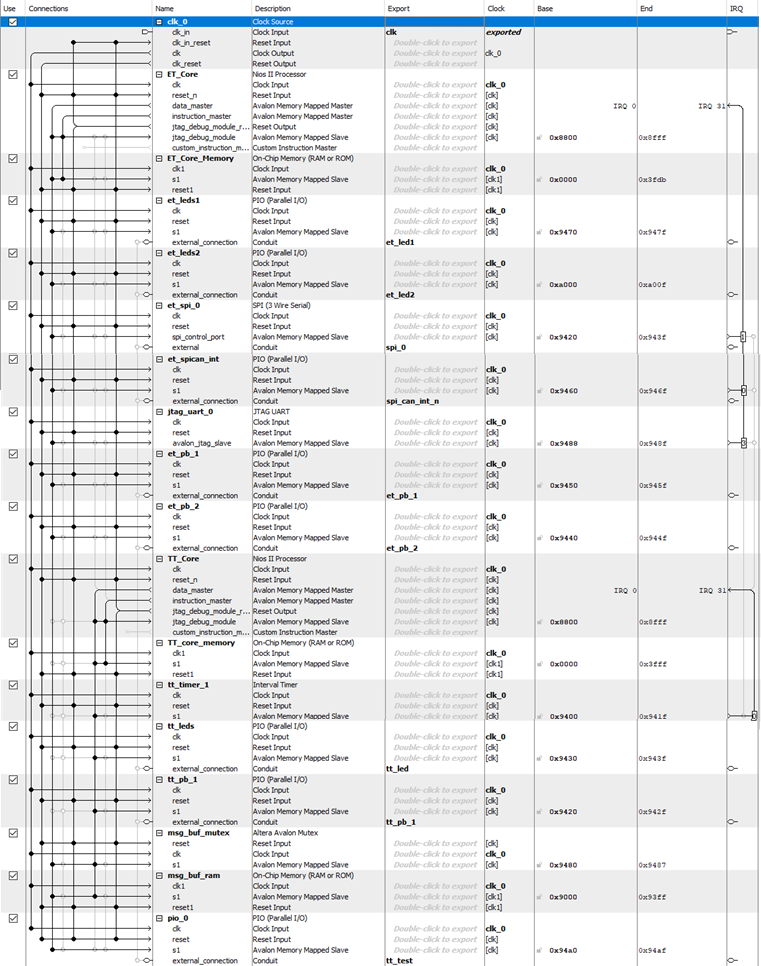
This CAN bus, via which the two nodes were connected was implemented using the CAN-SPI modules. The dual-core slave node had an event triggered (ET) core and a time triggered (TT) core. Tasks that were event triggered were handled by the ET core and the TT core handled time triggered tasks. A shared message buffer was used in order to allow the slave and master nodes to communicate. Communication between the two slave cores is also established via a mutex, which allowed both cores to have access to the buffer.

Using the messages that were received, the TT core had to respond accordingly by executing the tasks referred to in the message such as turn an led on/off. The ET core was connected to the master node via the CAN bus as a slave. The master node generated “bursts” of data at random time intervals. The when the ET core received these messages it had to process the messages again by executing certain tasks, similar to the TT core.

# Master Qsys Design



# Slave Qsys Design



**Description of Software Developed for Master Node**

In the master Qsys design, the various components can be seen:

* Clock: Synchronises the whole system and it runs system at a specific frequency.
* Interval Timer: Runs at 5ms ensures the system is synchronised according to the clock and operations are executed accordingly. Such as the rate of peripheral operations like blinking LEDs.
* RAM: Is volatile memory, used for storing various data on the board such as the software implementation. Thus, the data is wiped when the power is turned off.
* Parallel Input/Output (PIO): These are used to set-up and control the pushbuttons and the LEDs on the board and other peripherals.
* JTAG UART: Implements a method to communicate serial character streams, thus this is used to communicate with the core by reading and writing control and data registers on the processors.
* The SPI: allows for the CAN bus to be implemented between the master and ET slave node, it is a serial peripheral interface. It allows communication between the two nodes.

# 3.1)

## The Controller Area Network (CAN) protocol is a serial communications protocol defined by the International Standardization Organization (ISO). It uses a Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) protocol which is based on prioritising data packets. In this protocol, a node that receives a message (or packet) and wishes to transmit checks that the transmission channel i.e. the bus is free (thus no other node is transmitting at the time). If the bus is free/idle, then the highest priority message is sent. If the bus is not clear, the node waits until the bus is free; a node cannot transmit a message whilst another message is being transmitted. The maximum number of nodes (between 32 and 64 nodes) on a CAN bus depends on the data rate and the bus length. Each message can send up to 8 bytes of data. The bus length influences the speed at which messages are transmitted e.g. for a 40m bus it is 1Mbps. The cabling is cheap and CAN controllers are widely available. The CAN specifies two logical states for bit coding, a logic 1 which is recessive and a logic 0 is dominant. A non‐return to zero (NRZ) method of encoding is used which keeps the signal at the assigned binary level for the entire bit time. Thus, the voltage does not return to zero if the value of the signal is a logic 1 until the bit period is complete.

CAN messages have several formats however, the most common message type is the data frame format. This carries a message from the transmitter to the receiver. Each frame starts with a single dominant bit, interrupting the recessive state of the idle bus. The data frame is composed of the (7) different fields, there are two types the standard and extended data frame.

## In a shared clock network, an assumption is made that the CAN network is free when a node wishes to send either a tick or acknowledgement message under correct operation. On the master node there is a precise single clock (the clock drives its scheduler) which keeps the slaves that are connected on the network in sync.

## Tick messages are sent from the master to the slave(s) periodically and these are interrupts that drive the slaves’ shared-clock scheduler. The slave does not have its own timer-based interrupt to drive the scheduler, it relies on the tick message from the master. Thus, the interrupt is generated as a byte of data is sent from the master to the slave. There is a slight delay in between the tick generation on the master and slave nodes in some networks. However, it is important that the short delay is predictable and fixed. To transfer data from the slave to the master acknowledgement messages are sent (as well as the tick messages from the master to the slave) and this interleaving of messages establishes a time division multiple access (TDMA) protocol.

Slave Acknowledgement Messages

All slaves connected to the master generate an interrupt when a tick message is sent from the master. However, only the slave addressed in the tick message sends an acknowledgement message to the master, again this message is up to 8 bytes long. Each slave connected on the network has a unique ID. The first byte of the acknowledgement message is the ID of the slave and the remaining bytes are the message data (depends if a message is being sent). All the bytes of the acknowledgement message are sent in the tick interval in which the tick message was received.

The timer on the master triggers the scheduler tick. However, on the slave, the scheduler tick is triggered due to the tick message sent from the master via the CAN bus.

Message processing (Master)

The master first receives an acknowledgement message from a previous slave node. The master sends a tick message to the current slave using its corresponding ID. It then checks for errors and the scheduler task dispatcher is executed.

Message processing (Slave)

The slave checks if the ID sent in the tick message from the master matches the slave ID for the corresponding slave. No message is sent back if the IDs do not match. However, if the IDs match then an acknowledgement message is sent back to the master, containing the slave ID in the message. Then, the slave scheduler task dispatcher is executed. However, no message is sent back if the IDs do not match.

Handshake implementation

The handshake is used to start up the slaves, it establishes communication with each slave in the system and deals with missing slaves if necessary. This handshake also is used to keep the slaves synchronised. To implement this, the master sends a message containing 0x00 in the first data byte (byte 0) and the slave ID in the second data byte (byte 1). The slave with the corresponding ID will reply with the same message. If the master receives the correct reply from the slave (i.e. the correct code), then it has been able to set up communication with the slave. The master then continues to establish contact with the next slave connected on the bus.

## This CAN network is set up between the ET core and the master node in the system that has been implemented.

## Master and Slave Node Tasks

HEARTBEAT.c – in master, ET slave and LED\_flas.c in TT slave

This task flashes an LED on a specified port pin (this depends on the Qsys design), the HEARTBEAT\_Update(void) function just turns the LED on and off. This function has to be called at twice the required flash rate as the LED will be on for half of the period and off for the other half. On the master and the ET core, LED2 flashes constantly. However, for the TT core LED6 flashes constantly.

PUSHBUTTON.c – in master and ET slave

This task checks the duration for which the pushbutton has been pressed on the boards and then takes the corresponding action. So, for the master board, if the duration variable is greater than the value of the constant SW\_THRES then a tick message containing the code (0x03,0x04,0x05,0x06) is sent to the current slave. However, if the button is not held for long enough then the (0x00,0x00,0x00,0x00) is sent to the current slave instead. This message is also sent if the push button hasn’t been pressed at all. The value for SW\_THRES is chosen based on the switch debounce behaviour.

On the other hand, on the slave, the operation is the same except instead of a tick message an acknowledgement message is sent to the master. The duration that the pushbutton has been pressed has to be greater than SW\_THRES same as before, if it is then an acknowledgement message containing the value of 0xAA is sent. If the duration variable is not greater than SW\_THRES or if the pushbutton has not been pressed, then an acknowledgement message containing the ASCII value of ‘C’ is sent to the master instead.

LED\_ONOFF.c – in master and ET slave

This task is for blinking an LED on a specific port pin (depending on the Qsys design) after it receives a message from either the master or the ET slave. The LED\_ONOFF\_Update(void) function, will blink the LED on the master board if the ET slave sends the correct acknowledgement message containing the code 0xAA. If it doesn’t send this message, then the LED will remain turned off. In the case of the ET slave, the master must send the tick message containing the code 0x03 to blink the LED, else it will remain turned off on the slave board.

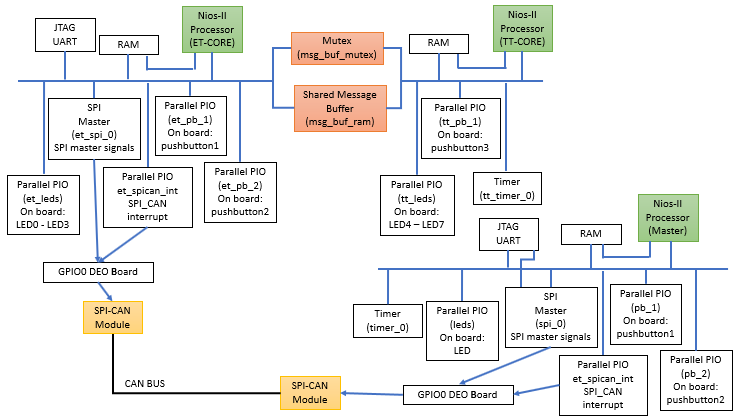
SharedMem\_Mutex.c – in ET slave and TT slave

This task uses the mutual exclusion to access shared memory. The ET slave is connected to the master and so it receives tick messages from the master. The SharedMem\_Update(void) function first tries to acquire the lock and then if the message buffer is empty (checks if the flag is set to NO\_MESSAGE), the function writes the contents of the Tick\_message\_data\_g array to the shared memory message buffer using pointers. And then changes the flag to MESSAGE\_WAITING (this flag means that a message has been put in the buffer). Finally, it releases the lock. However, if the lock is not acquired then an LED on a specific port pin on the is turned off/on depending on the previous state of the LED, on the ET core.

The TT slave accesses the shared memory when the flag is set to MESSAGE\_WAITING. The SharedMem\_Update(void) function will only acquire the lock if that flag is set. The function writes the contents of the message buffer to its own receive buffer using pointers. And then changes the flag to NO\_MESSAGE. Finally, it releases the lock. Then, it checks if position four of the receive buffer has the code 0x06, in which case, an LED on the TT core switches on/off depending again on its previous state.

## The ET slave core is connected to the master node via the CAN bus and it receives tick messages containing codes that will blink LED0 (on the ET core) or turn on/off LED5 (on the TT core). The ET core is event triggered; thus, periodic tick messages received from the master are the interrupts that drive the slaves’ shared-clock scheduler. The slave does not have its own timer-based interrupt to drive its scheduler, it relies on the tick messages from the master. Whereas the TT slave core, has its own timer, it is time-triggered. Thus, the interrupts are generated by its own timer which drive its scheduler. It does not have any direct connection with the master node.

## The ET core and the TT core communicate via the shared message buffer and the two cores access the buffer is by a mutex. The mutex is used to ensure that a critical region is not accessed by both cores at the same time. The buffer is a critical region because it is a shared resource. This could lead to race conditions i.e. if the cores access the buffer at the same time which could lead to a deadlock. Eventually leading to the system not being implemented as required.

Mutual exclusion is implemented using a lock mechanism and flags. The ET core acquires the lock first and checks if the flag is set to NO\_MESSAGE, if so then it proceeds to write the contents of the tick messages from the master to the shared buffer. Then, it changes the flag to MESSAGE\_WAITING and releases the lock. The TT core on the other hand, first checks if the flag is set to MESSAGE\_WAITING, if so, it acquires the lock. Next, it writes the contents of the shared buffer to its own receive buffer. Finally, it changes the flag to NO\_MESSAGE and releases the lock. This ensures that both cores access the buffer separately and neither core are using the shared resource at the same time.

## 

# 3.2)

## 

## 

## Startup

Initially, the master checks the presence of the ET slave by sending a tick message with the code (0x00, 0x02); the second byte is the slave ID. If the slave with the ID of 0x02 is present, it shall send an acknowledgement message of (0x00, 0x02) to the master. This is the handshake implementation. If the messages are exchanged successfully during start up process, then the slave is connected to the master. After which, the slave and the master can communicate with each other.

## Normal Operations: no Button is pressed

If no button is pressed, then the master sends a message of (0x02, 0x00, 0x00, 0x00, 0x00) to the ET slave. This time round the first byte is the slave ID. The following codes are used for normal operation. After receiving the message, the slave will reply with an acknowledgement message of (0x02, 0x43). Again, the first byte is the slave ID.

## Master Button0 Press

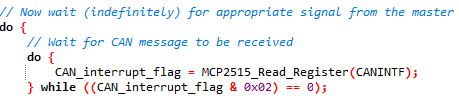
When button0 is pressed on the master board, the tick message sent from the master contains the following codes (0x02, 0x03, 0x04, 0x05, 0x06), again the first byte is the slave ID. When the slave receives this message, it sends an acknowledgement message containing the same code as normal operation (0x02,0x43). This will cause LED5 on the TT core to turn ON/OFF and LED0 will blink on the ET core.

## Slave Button0 Press

When button0 is pressed on the slave board, the acknowledgement message sent from the slave changes to (0x02, 0xAA), and this will blink LED0 on the master board.

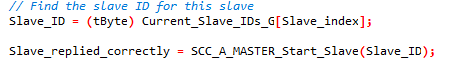
## The startup handshake protocol:

## The SCH\_Start(void) function on the master is used to set up a connection with the slave on startup. The ET slave waits for a master signal inside SCH\_Start(void) function.



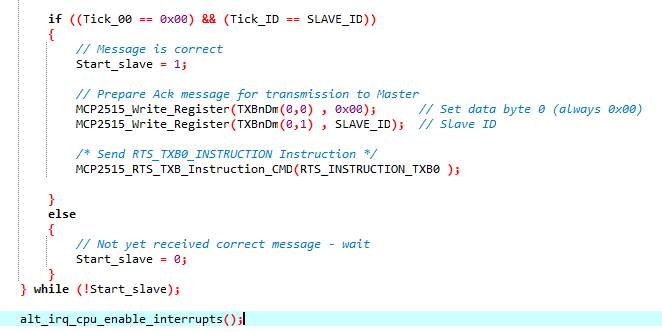
**Figure 1 – ET slave waiting for master in 2\_50\_XXg.c file**

The master sends a tick message to the slave with ID equal to SLAVE ID to implement the “handshake” using the SCC\_A\_MASTER\_Start\_Slave(Slave\_ID) function which is called within the SCH\_Start(void) function.



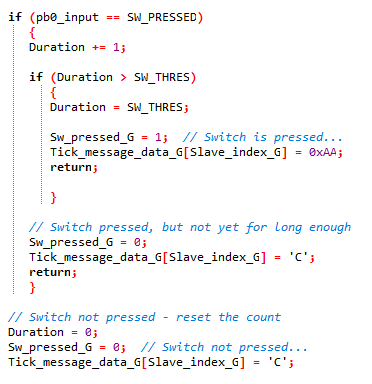
**Figure 2 – Master sends handshake tick message to slave in 2\_50\_XXg.c file**

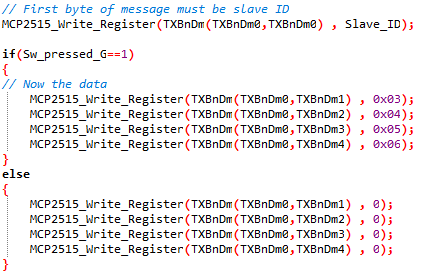
On receipt of the start message, the slave accepts the signal from the master (in the if statement). The slave sends a reply in the SCH\_start(void) function using MCP2515\_RTS\_TXB\_Instruction\_CMD(const tByte tx\_buffer) function which sends an acknowledgement message back to the master specifically for the handshake and finally enables CAN SPI interrupts.



**Figure 3 – Slave handshake acknowledgement message and enables interrupts in 2\_50\_XXg.c file**

## Normal Operations: no Button is pressed

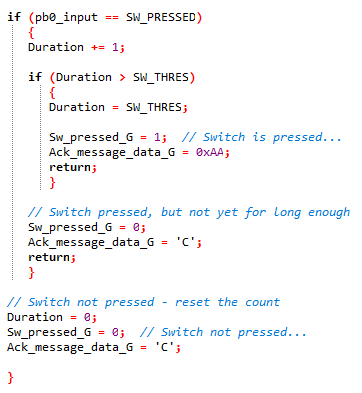
When no button is pressed (the value of Sw\_pressed\_G depends on the duration variable in the PushButton.c file), the master node sends a message of (0x02, 0x00, 0x00, 0x00, 0x00) to the ET slave in the SCC\_A\_MASTER\_Send\_Tick\_Message(const tByte SLAVE\_INDEX) function. The first byte being the slave ID.



**Figure 5 – How the Sw\_pressed\_G variable is dependent on the duration variable in PushButton.c file (master)**

**Figure 4 – Master tick message contents for no button pressed in 2\_50\_XXg.c file**

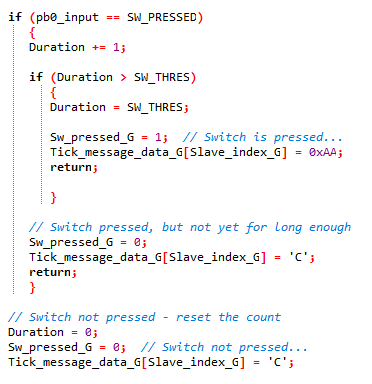
The Sw\_pressed\_G variable checks for the duration for which the button has been pressed in the PushButton.c file in the PushButton\_Update(void) function for the master. Depending on the value of the variable, the master node writes different values to the transmit buffers.

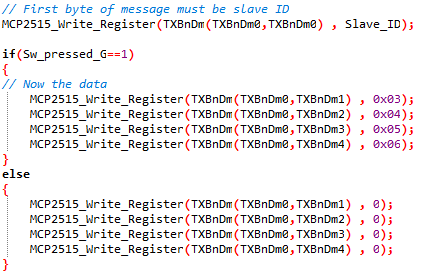


**Figure 6 – Acknowledgement message for no button pressed in PushButton.c file (ET slave)**

The slave sends an acknowledgement of (0x02, 0x43) which continues normal operation.

## Master Button0 is pressed:

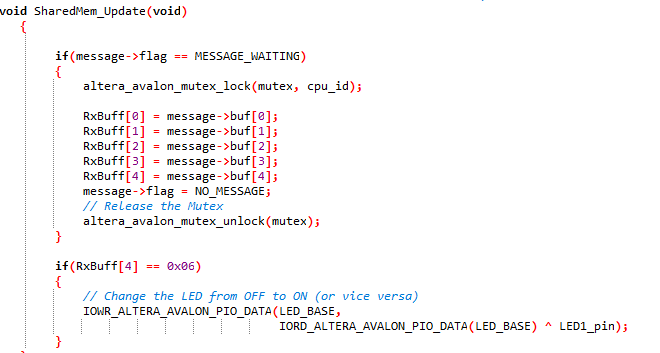
When pushbutton0 is pressed (the value of Sw\_pressed\_G depends on the duration variable in the PushButton.c file), the master node sends a message of (0x02, 0x03, 0x04, 0x05, 0x06) to the ET slave in the SCC\_A\_MASTER\_Send\_Tick\_Message(const tByte SLAVE\_INDEX) function. The first byte being the slave ID.



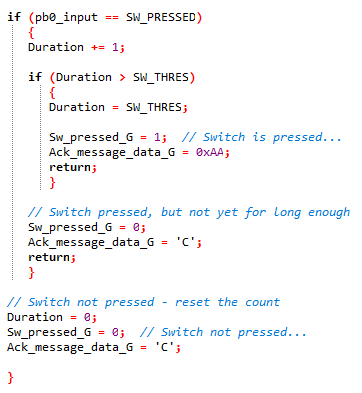
**Figure 7 – Master tick message contents for button0 pressed on master in 2\_50\_XXg.c file**

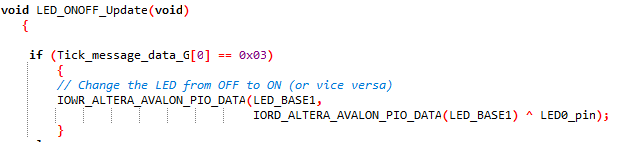
**Figure 8 – How the Sw\_pressed\_G variable is dependent on the duration variable in PushButton.c file (master)**

Now this will lead to LED5 on the TT core switching ON/OFF (the SharedMem\_Update(void) function in SharedMem\_Mutex.c checks position 4 of the receive buffer which has the code 0x06) and LED0 on the ET core will blink (the LED\_ONOFF\_Update(void) function in LED\_ONOFF.c checks the position 0 of the tick message data array which has the code 0x03). The ET slave will send an acknowledgement message containing the same code as normal operation (0x02,0x43).



**Figure 9 – LED5 turning ON/OFF due to button0 pressed on master in SharedMem\_Mutex.c file (TT slave)**

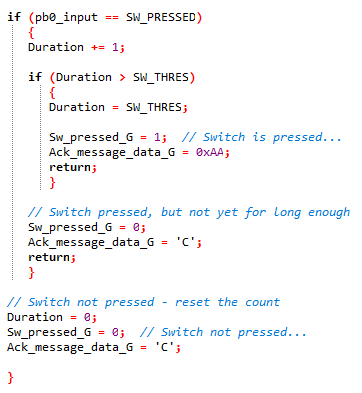


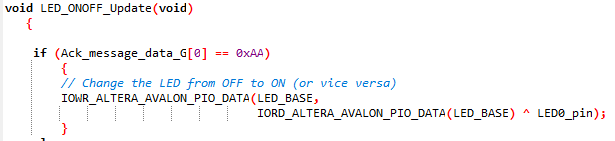


**Figure 10 – LED0 blinking due to button0 pressed on master in LED\_ONOFF.c file (ET slave)**

**Figure 11 – Acknowledgement message for button pressed on master in PushButton.c file (ET slave)**

## Slave Button0 is pressed:

The acknowledgment message changes from the (0x02,0x43) to (0x02, 0xAA) in the ET slave app. The LED\_ONOFF\_Update(void) function looks for the 0xAA code in the acknowledgment message. Hence, LED0 blinks on the master.



**Figure 13 – Acknowledgement message for button pressed on slave in PushButton.c file (ET slave)**

**Figure 12 – LED0 blinking on master due to button0 pressed on slave in LED\_ONOFF.c file (master)**

# Conclusion

Overall, this piece of work has helped in understanding a shared clock CAN bus implementation through a single core master - dual core slave node setup. These types of embedded application systems are generally used in safety critical tasks. The use of a mutex to access shared memory through a message buffer was incorporated in this assignment, which has taught how differently triggered processors can communicate with each other, whilst being on the same node.

Through the assignment, the following has been achieved:

* Implementation of inter-processor communication between an event triggered slave core and a time triggered slave core. This was possible through the shared memory mutex.
* How a dual core slave interacts with a single core master was learnt through this exercise.