UNIVERSITY OF LEICESTER

Department of Engineering

EG3205 Programming Microelectronic and Multi-Core Systems

Part II. Multiprocessor and Multicore Systems

Prof. Tanya Vladimirova
MEng MSc PhD CEng FIET SMIEEE MACM FHEA
Email: tv29@le.ac.uk

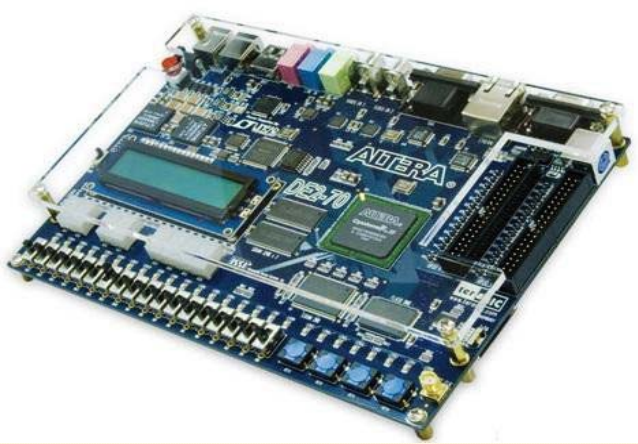
Outline

- Lab 4 Exercise: Shared Clock Scheduling over CAN SPI
  - MCP MCP2515 CAN Controller IC: overview
  - SPI CAN driver: details
  - Shared-Clock TTC Scheduler: implementation
- Assignment 2 brief

Lab 4 Exercise

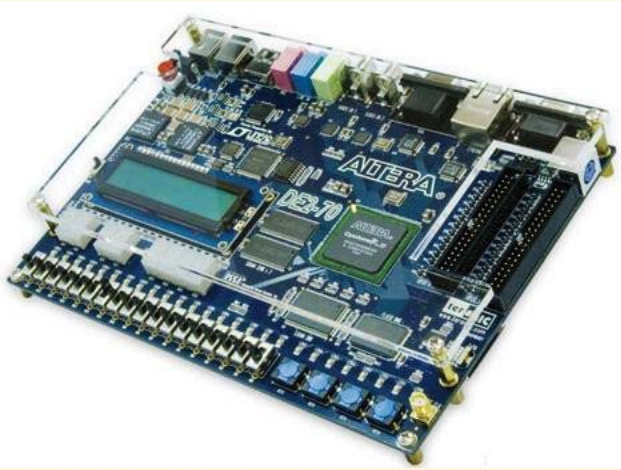
Dual-Processor System Design Using CAN and TTC-SC Scheduler

Lab 4 Setup: Flexible implementation platform



Master:
FPGA

+



Slave:
FPGA

Lab 4 Exercise

Shared Clock Scheduling over CAN SPI:

- Master and slave software applications:
  - `app_master.zip`
  - `app_slave.zip`
- An SPI CAN driver communicating with the MCP2515 CAN Controller:
  - `SPICAN_Driver.zip`

Quartus II Reference Materials:

- Introduction to the Altera Qsys Tool
- Nios II Hardware Development Tutorial
- Embedded Design Handbook

Quartus II Reference Materials:

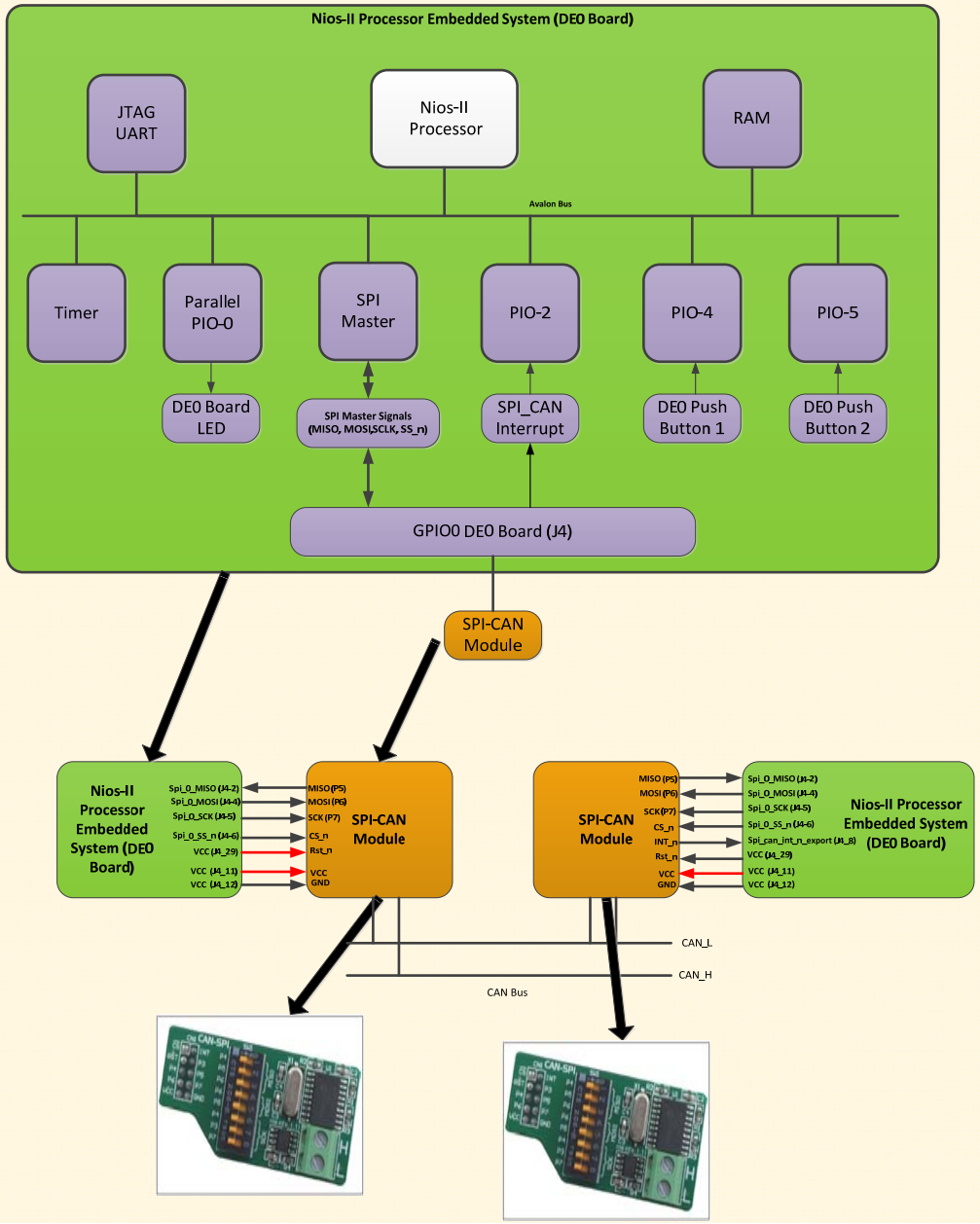
- DE0 board user manual
- CAN-SPI board manual

Scheduler Reference Materials:

- M Pont PTTES book

CAN / SPI Reference materials:

- SPI Spec
- Introduction to CAN
- MCP2515 CAN Controller IC



Controller Area Network (CAN):

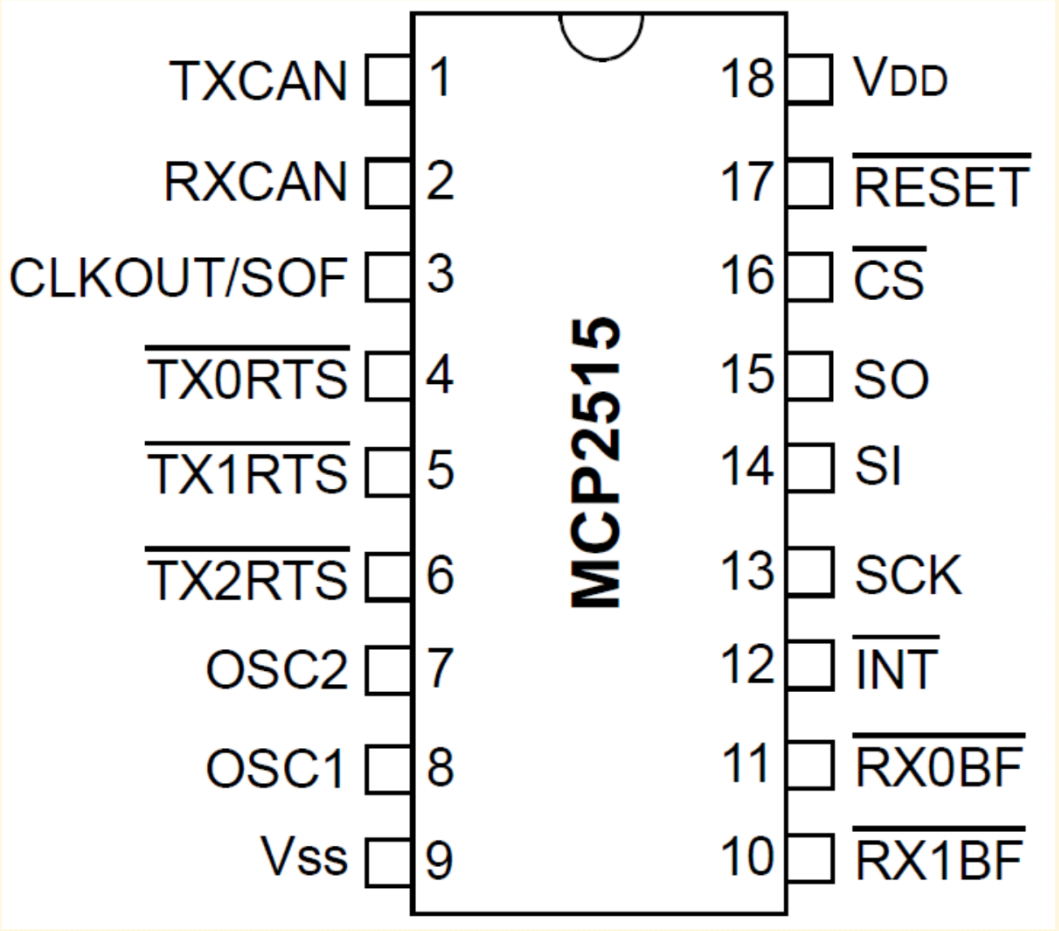
MCP2515 CAN Controller

MCP2515 CAN Controller

- Microchip Technology's MCP2515 is a stand-alone Controller Area Network (CAN) controller that implements
  - ➢the CAN specification, ver. 2.0B.

- Package Type:
  - ➢18-lead PDIP/SOIC

PDIP - Plastic Dual In-Line Package
SOIC - Small Outline Integrated Circuit



MCP2515 Datasheet

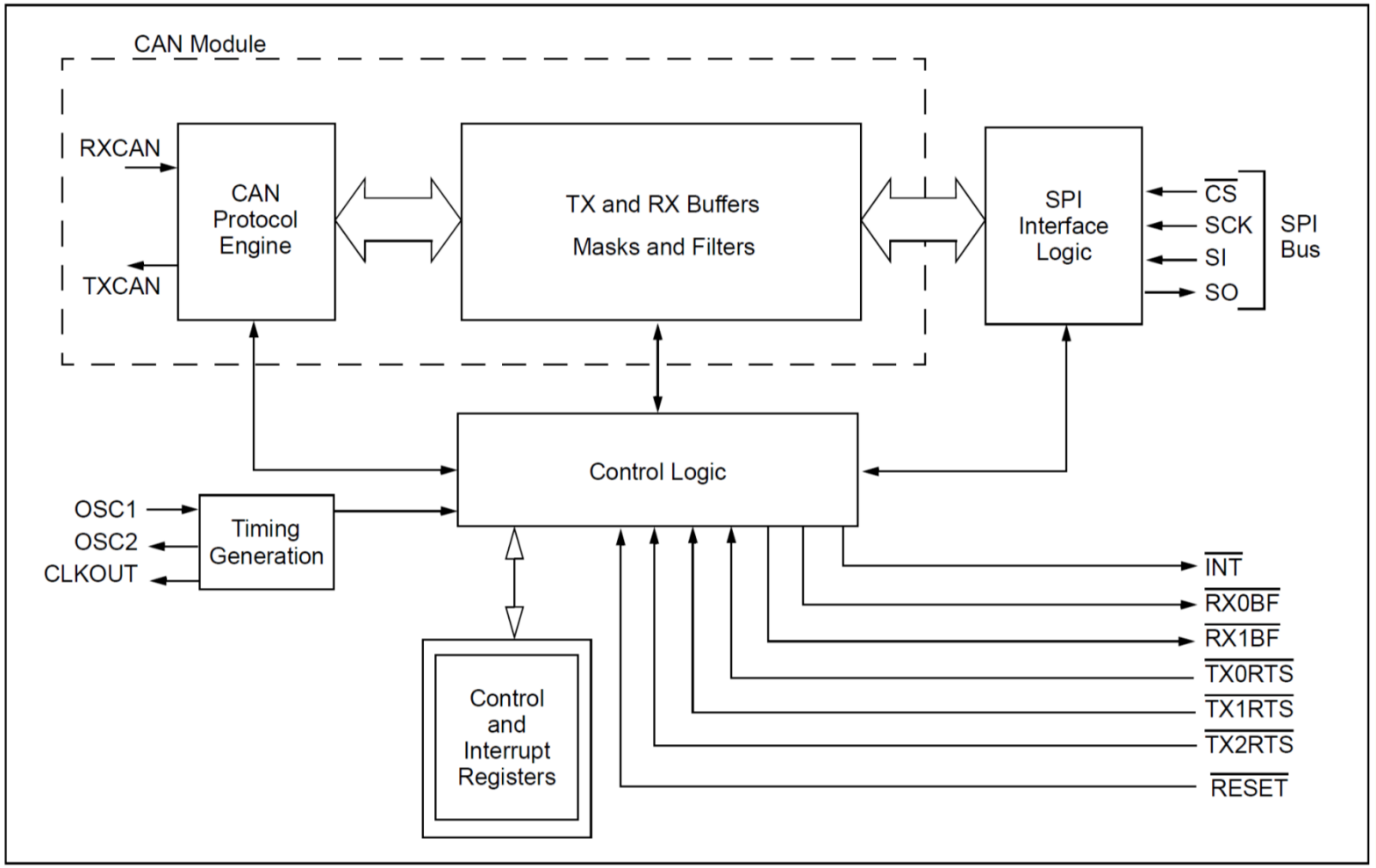
A simple block diagram of the MCP2515

Consists of three main blocks:

1. The CAN module, which includes the CAN protocol engine, masks, filters, transmit and receive buffers.

2. The control logic and registers that are used to configure the device and its operation.

3. The SPI protocol block.



MCP2515 Datasheet

MCP2515 Pinout

Name	PDIP/SOIC Pin #	TSSOP Pin #	I/O/P Type	Description	Alternate Pin Function
TXCAN	1	1	O	Transmit output pin to CAN bus	—
RXCAN	2	2	I	Receive input pin from CAN bus	—
CLKOUT	3	3	O	Clock output pin with programmable prescaler	Start-of-Frame signal
$\overline{\text{TX0RTS}}$	4	4	I	Transmit buffer TXB0 request-to-send. 100 kΩ internal pull-up to VDD	General purpose digital input. 100 kΩ internal pull-up to VDD
$\overline{\text{TX1RTS}}$	5	5	I	Transmit buffer TXB1 request-to-send. 100 kΩ internal pull-up to VDD	General purpose digital input. 100 kΩ internal pull-up to VDD
$\overline{\text{TX2RTS}}$	6	7	I	Transmit buffer TXB2 request-to-send. 100 kΩ internal pull-up to VDD	General purpose digital input. 100 kΩ internal pull-up to VDD
OSC2	7	8	O	Oscillator output	—
OSC1	8	9	I	Oscillator input	External clock input
VSS	9	10	P	Ground reference for logic and I/O pins	—
$\overline{\text{RX1BF}}$	10	11	O	Receive buffer RXB1 interrupt pin or general purpose digital output	General purpose digital output
$\overline{\text{RX0BF}}$	11	12	O	Receive buffer RXB0 interrupt pin or general purpose digital output	General purpose digital output
$\overline{\text{INT}}$	12	13	O	Interrupt output pin	—
SCK	13	14	I	Clock input pin for SPI interface	—
SI	14	16	I	Data input pin for SPI interface	—
SO	15	17	O	Data output pin for SPI interface	—
$\overline{\text{CS}}$	16	18	I	Chip select input pin for SPI interface	—
$\overline{\text{RESET}}$	17	19	I	Active low device reset input	—
VDD	18	20	P	Positive supply for logic and I/O pins	—
NC	—	6,15	—	No internal connection	

Note: Type Identification: I = Input; O = Output; P = Power

Use of the supplied SPI CAN driver

- Why using it?
  - To communicate with a CAN controller (in your case with the controller of the SPI-CAN module) via SPI
- How does it work?
  - In the datasheet of the MCP2515 CAN Controller, there is a list of CAN-SPI instructions (**RESET**, **READ**, **WRITE**, etc.) along with their description (see from page 63 onwards)

Use of the supplied SPI CAN driver

The SPI CAN driver implements the functionality for using the CAN-SPI operations via respective C functions:

- `MCP2515_Init()`: Initialises the CAN device via SPI
- `MCP2515_Reset()`: Resets the CAN device using the **RESET** instruction
- `MCP2515_Read_Register(Register_address)`: Reads from a register address using the **READ** instruction
- `MCP2515_Write_Register(Register_address,Register_contents)`: Writes to a register address some data using the **WRITE** instruction

Use of the supplied SPI CAN driver

The MCP2515 has 2 Receive buffers.

- How to use the `MCP2515_Read_Register()` to read the `CANINTF` register*?
  - `MCP2515_Read_Register(CANINTF)`
- How to use the `MCP2515_Read_Register()` to read a specific byte from a specific Receive buffer?
  - `MCP2515_Read_Register(RXBnDm(n,m))`
  - RXBnDm(n,m): Receive buffer N data byte M
- Read byte 0 of Receive buffer 1
  - `MCP2515_Read_Register(RXBnDm(RXBnDm1,RXBnDm0))`

*When a message is moved into either of the receive buffers, the appropriate CANINTF.RXnIF bit is set.

Use of the supplied SPI CAN driver

The MCP2515 has 3 Transmit buffers.

- How to use the `MCP2515_Write_Register()` to write to a specific byte of a specific Transmit buffer?
  - `MCP2515_Write_Register(TXBnDm(n,m), Register_contents)`
  - TXBnDm(n,m): MCP2515 Transmit buffer N data byte M

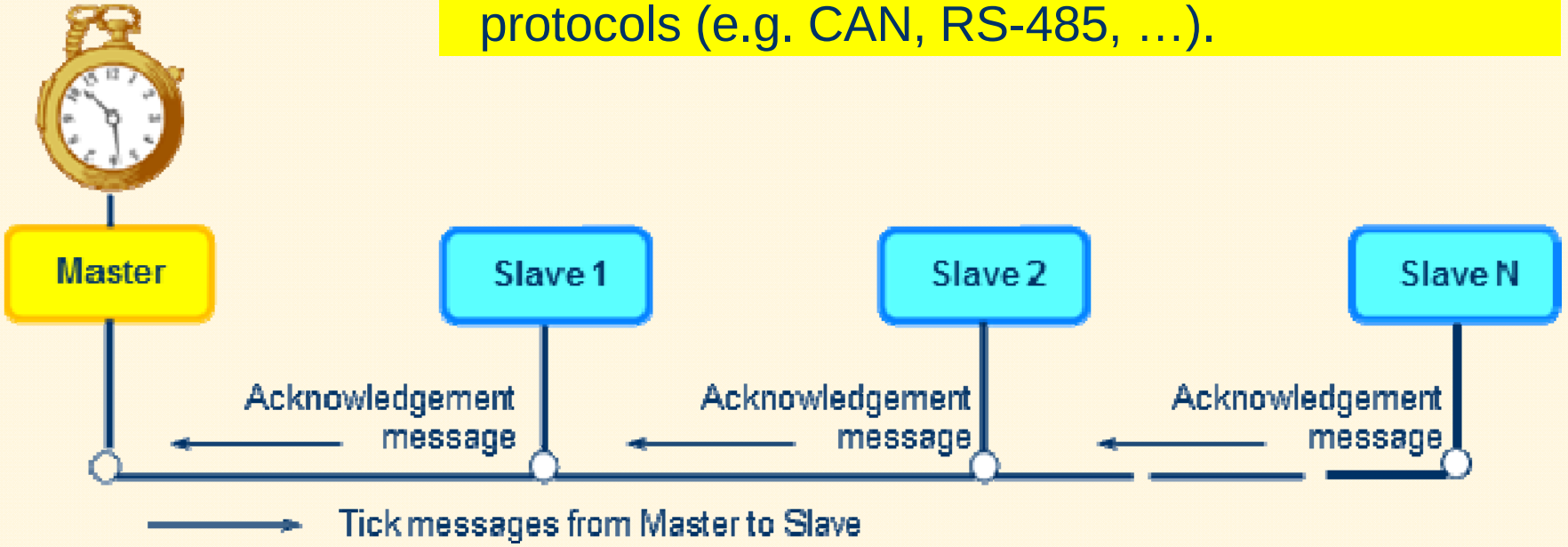
- Write the variable `SLAVE_ID` to byte 5 of Transmit buffer 0
  - `MCP2515_Write_Register(TXBnDm0,TXBnDm5), Slave_ID)`

Shared Clock TTC Scheduler

Implementation

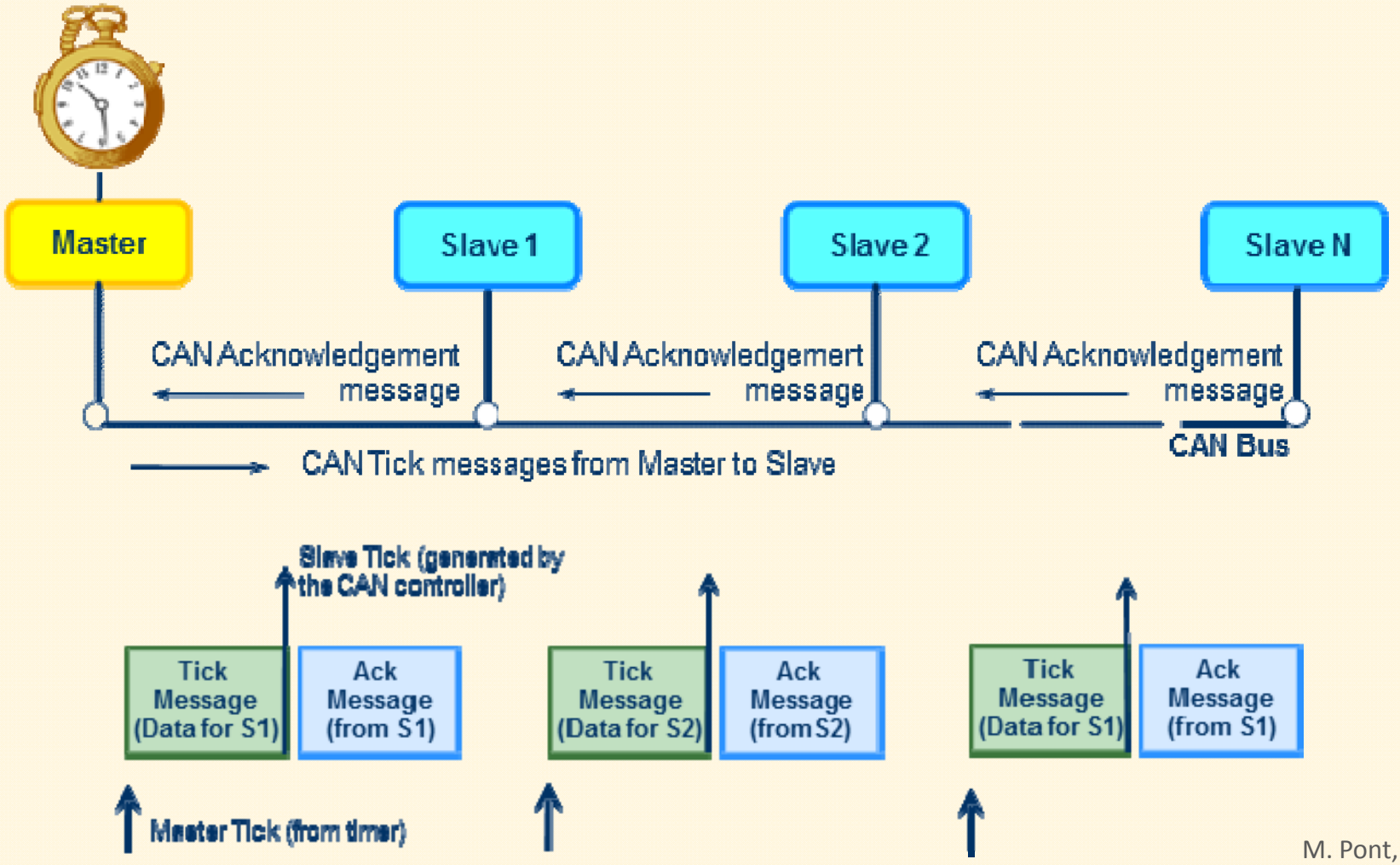
Synchronisation – Shared-Clock protocol

- All slaves are kept in sync with the Master clock.
- Robust.
- Low cost.
- Compatible with most “low level” network protocols (e.g. CAN, RS-485, …).



M. Pont, Patterns for time-triggered embedded systems, 2014

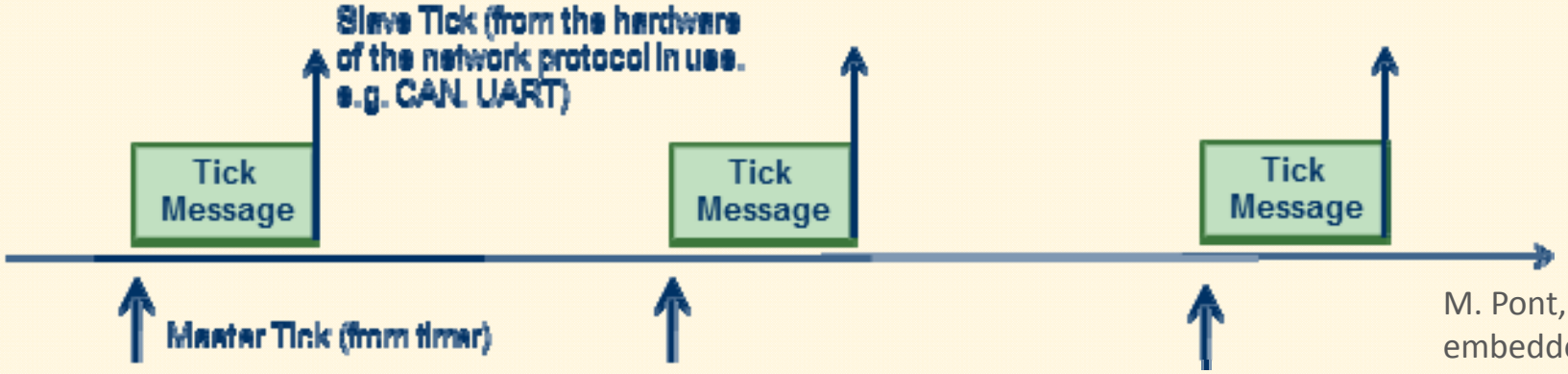
Shared-Clock CAN (TTC-SC in Lab 4)



M. Pont, Patterns for time-triggered embedded systems, 2014

Communication between the Master and Slave nodes in a CAN-based S-C network

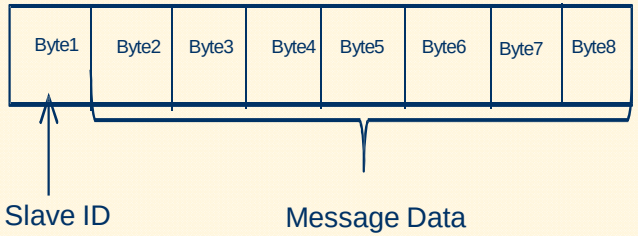
- This involves the transmission of the tick messages, triggered by the timer ticks in the Master.
- The receipt of the regular messages by the Slave is the source of the interrupt used to drive the scheduler in this node:
  - the Slave does not have a separate timer-based interrupt.
- By default, there is a slight delay between the tick generation on the Master and Slave nodes in some networks.
  - This delay is short (a fraction of a millisecond in most cases); equally important, it is predictable and fixed.



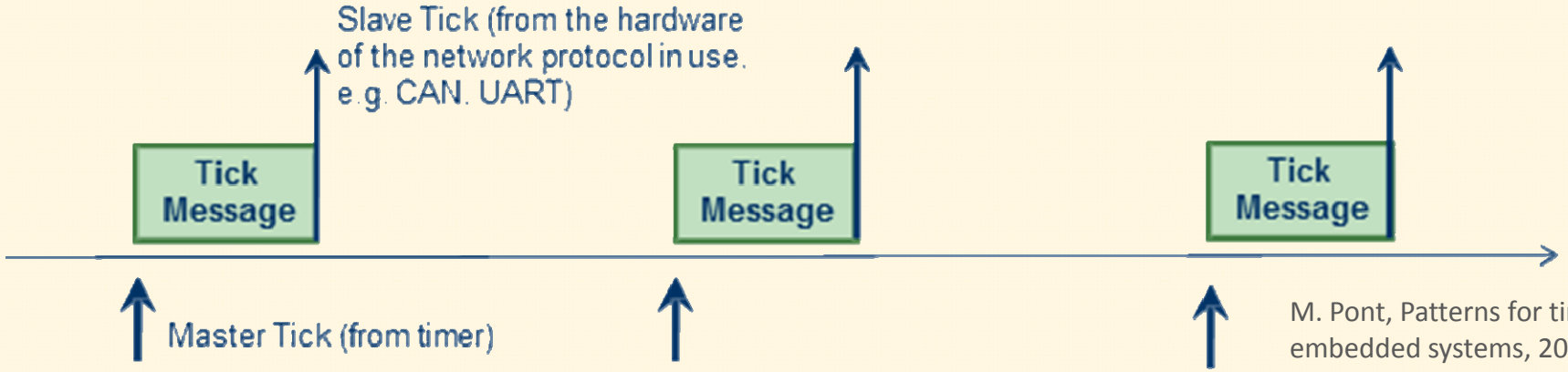
M. Pont, Patterns for time-triggered embedded systems, 2014

Tick Message Structure

- Each Slave on the network is given a unique ID (0x01 to 0xFF).
- Each Tick message is between one and eight bytes long; all the bytes are sent in a single tick interval.
- In all messages, the first byte is the ID of the Slave to which the message is addressed; the remaining bytes (if any) are the message data.



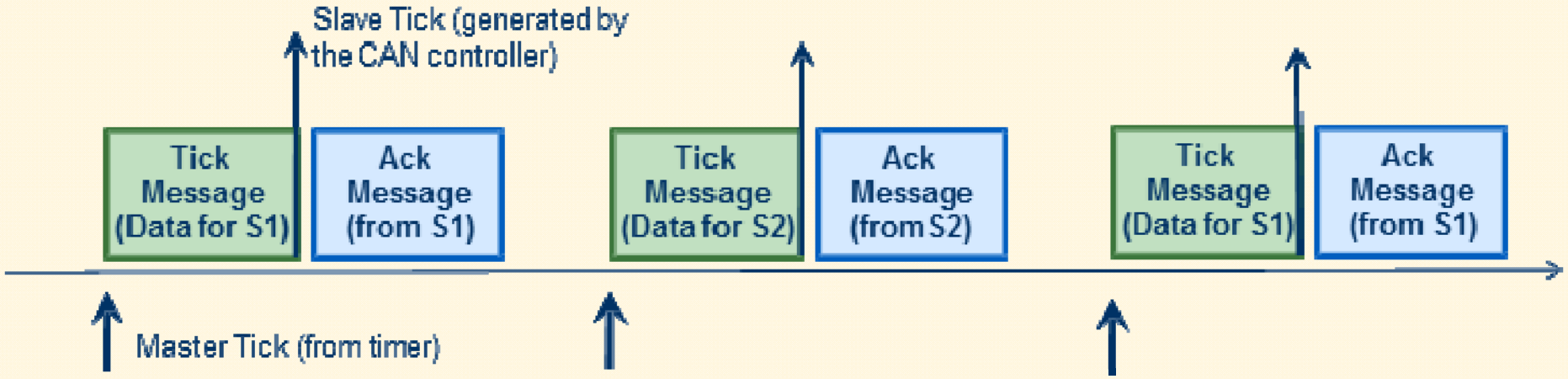
- All Slaves generate interrupt in response to every Tick message.



M. Pont, Patterns for time-triggered embedded systems, 2014

Data Transfer (Slave to Master)

- With Acknowledgement messages interleaved with Tick messages, the Shared-Clock Architecture now behaves in a TDMA fashion.



M. Pont, Patterns for time-triggered embedded systems, 2014

Shared Clock Scheduling over CAN:

Lab 4 Implementation

SC-CAN: A simple implementation of SC scheduling

- A master node is connected to slave nodes through a CAN bus (one master-multiple slaves schema)
- All nodes run a TTC scheduler
- Master node
  - The master node runs a TTC scheduler
  - The master node is responsible for triggering the slave node at the tick rate
  - The scheduler of the master node is triggered by a timer peripheral (timer interrupt)
- A slave node
  - A slave node runs a TTC scheduler
  - A slave node is triggered by the master node via the CAN bus (CAN interrupt)

SC-CAN scheduler: Master node

- Operation of the Master node
  - ➢ Initialise the scheduler
  - ➢ Start the scheduler
    - Start the slave (Implementation of the **Handshake protocol**)
    - Start the timer
  - ➢ Upon the arrival of a tick interrupt
    - Process the previous Acknowledgment message
    - Send a 'tick' message to all connected slaves (as a CAN DATA message)
- You can implement the Master node part of your SC-CAN scheduler using *six* functions

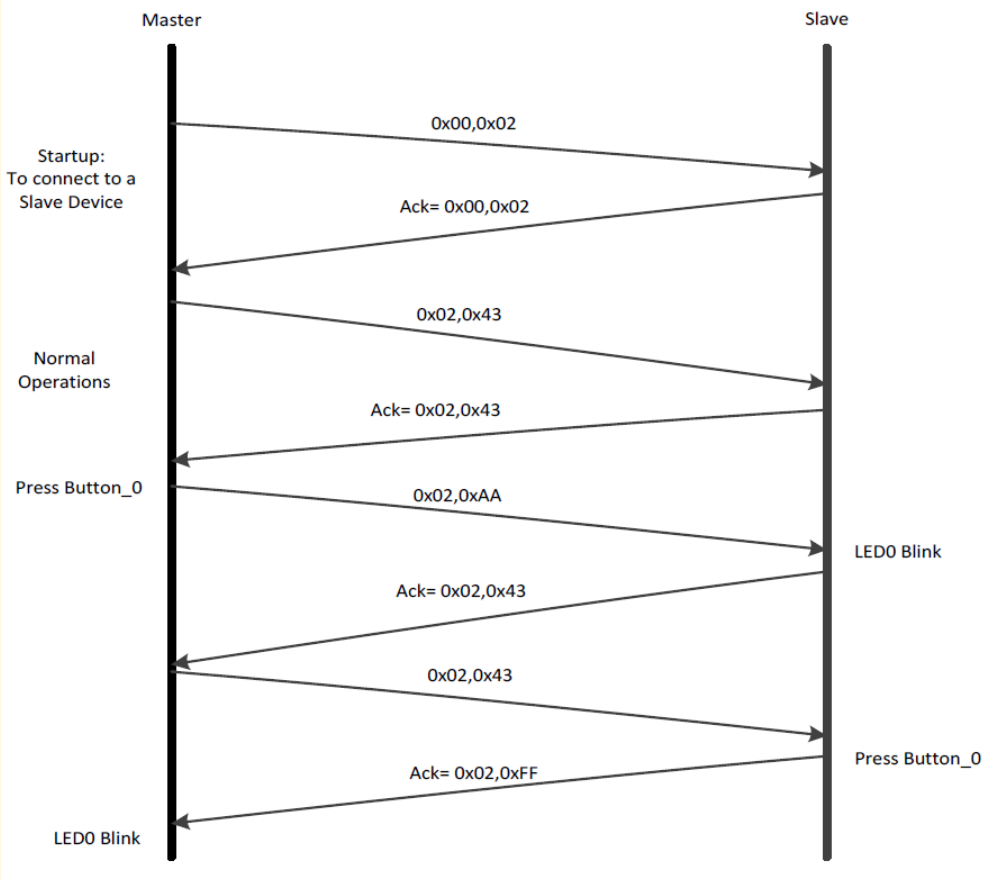
SC-CAN scheduler: Slave node

- Operation of the Master node
  - ➢ Initialise the scheduler
    - Set the scheduler to be triggered by a CAN interrupt
  - ➢ Start the scheduler
    - Implementation of the **Handshake protocol**
  - ➢ Upon the arrival of a CAN interrupt
    - Process the 'tick' message
    - If the 'tick' message was addressed to this slave copy its data and send an 'Acknowledge' message to the master
- You can implement the Slave node part of your SC-CAN scheduler using *five* functions

Shared Clock Scheduling over CAN:

Lab 4 Software Implementation

Communication and data sequence



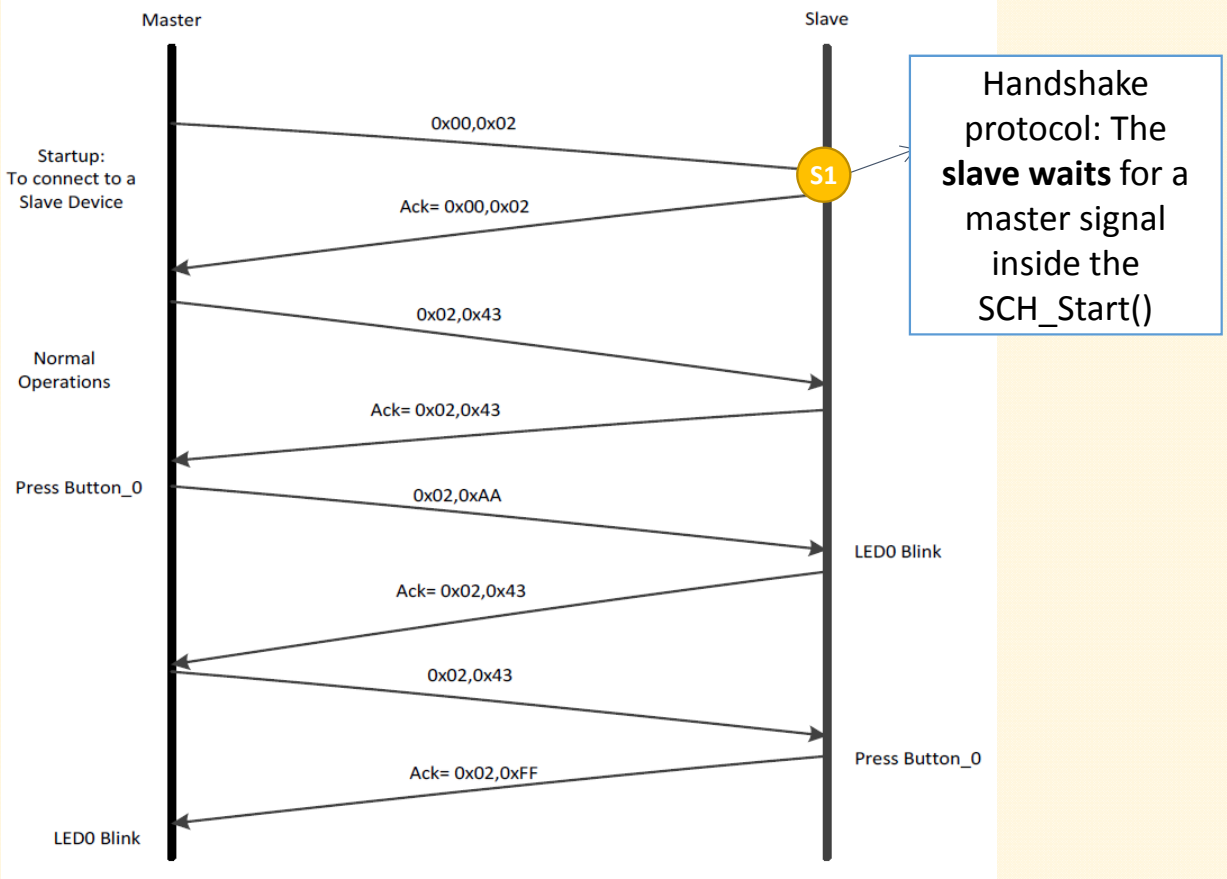
Master Node		
	Function Name	**Description**
Shared Clock Scheduler	Void SCH_Init_T0(void)	This function is used to initialize timer 0 and CAN-SPI.
	Void SCH_Start(void)	This function will try to establish connection with the slave on startup.
	Void SCH_Update(void * context)	This function will be executed in the master on timer interrupt. It will send a tick message, process ACK message and check whether a task is available to run or not.
	Void SCC_A_Master_Send_Tick_Message(const tByte SLAVE_INDEX)	This function will send a tick message to the slave over the CAN bus via the CAN-SPI module.
	Void SCC_A_Master_Start_Slave(const tByte SLAVE_ID)	This function will send a start message to the slave over CAN. It will also receive and process an ACK message from slave.
	Void SCC_A_Master_Process_Ack(const tByte SLAVE_INDEX)	This function will receive and process an ACK message from slave during normal operations.
	Void	This task updates the

Slave Node		
	Function Name	**Description**
Shared Clock Scheduler	Void SCH_Init_T0(void)	This function is used to initialize CAN-SPI and its interrupt.
	Void SCH_Start(void)	Wait for the master tick. Once it has arrived, process it and compare it with your own ID. Once ID is matched, enable the slave.
	Void SCH_Update(void * context)	This function will be executed in the slave, on receiving a CAN interrupt. It will process the tick message, send an ACK message and check whether a task is available to run or not.
	tByte SCC_A_Slave_Process_Tick_Message(void)	This function will process the tick message received from the master over the CAN bus via the CAN-SPI module.
	Void SCC_A_Send_Ack_Message(void)	This function will send an ACK message to the master.
	Void HEARTBEAT_Update(void)	This task updates the HeartBeat LED, which produces a blinking effect.

Handshake protocol:

Software Implementation

Communication and data sequence

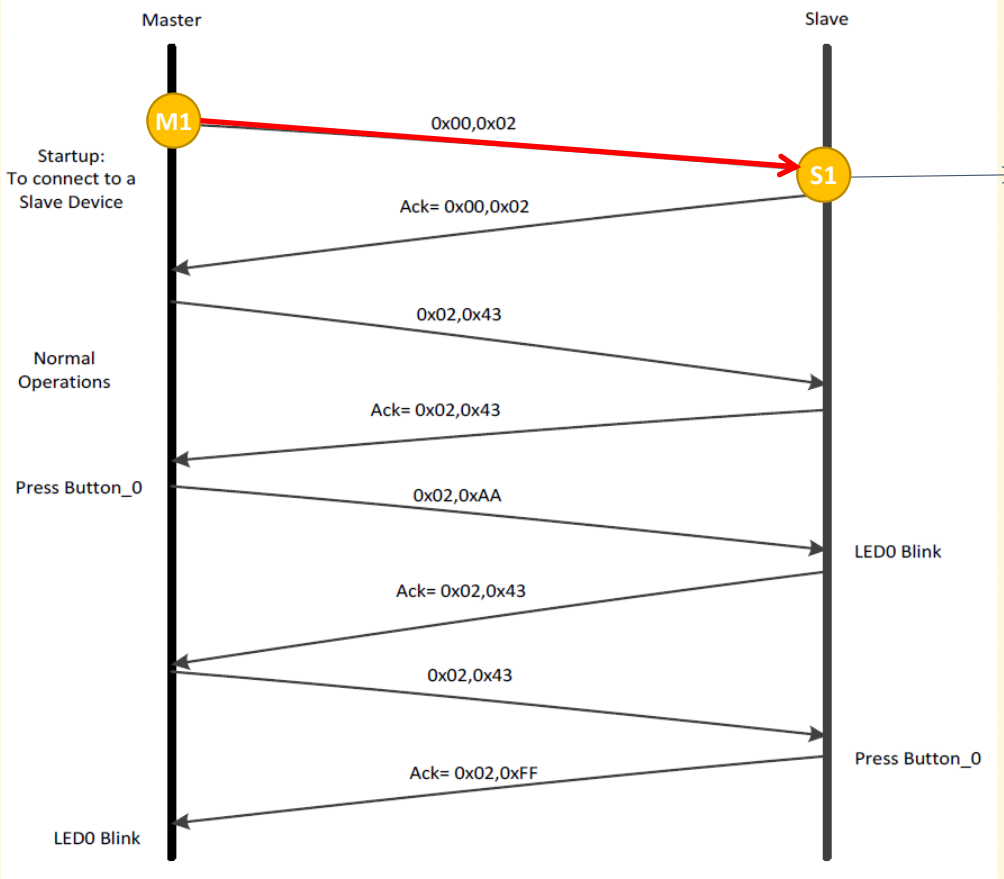


Slave node: `SCH_Start()`

```
// Now wait (indefinitely) for appropriate signal from the master
do {
    IOWR_ALTERA_AVALON_PIO_DATA(TEST_1_BASE,
                                IORD_ALTERA_AVALON_PIO_DATA(TEST_1_BASE) ^ TEST1_pin);
    // Wait for CAN message to be received
    do {
        CAN_interrupt_flag = MCP2515_Read_Register(CANINTF);
    } while ((CAN_interrupt_flag & 0x02) == 0);

    // Get the first two data bytes
    Tick_00 = MCP2515_Read_Register(RXBnDm(1,0)); // Get data byte 0, Buffer 1
    Tick_ID = MCP2515_Read_Register(RXBnDm(1,1)); // Get data byte 1, Buffer 1

    // We simply clear *ALL* flags here...
    MCP2515_Write_Register(CANINTF, 0x00);

    if ((Tick_00 == 0x00) && (Tick_ID == SLAVE_ID))
    {
        // Message is correct
        Start_slave = 1;

        // Prepare Ack message for transmission to Master
        MCP2515_Write_Register(TXBnDm(0,0) , 0x00);      // Set data byte 0 (always 0x00)
        MCP2515_Write_Register(TXBnDm(0,1) , SLAVE_ID);  // Slave ID

        /* Send RTS_TXB0_INSTRUCTION Instruction */
        MCP2515_RTS_TXB_Instruction_CMD(RTS_INSTRUCTION_TXB0 );

    }
    else
    {
        // Not yet received correct message - wait
        Start_slave = 0;
    }
} while (!Start_slave);
alt_irq_cpu_enable_interrupts(); //can move to down
```

Handshake protocol: The **slave waits** for a master signal

Communication and data sequence



Handshake protocol: The **slave waits** for a master signal inside the SCH_Start()

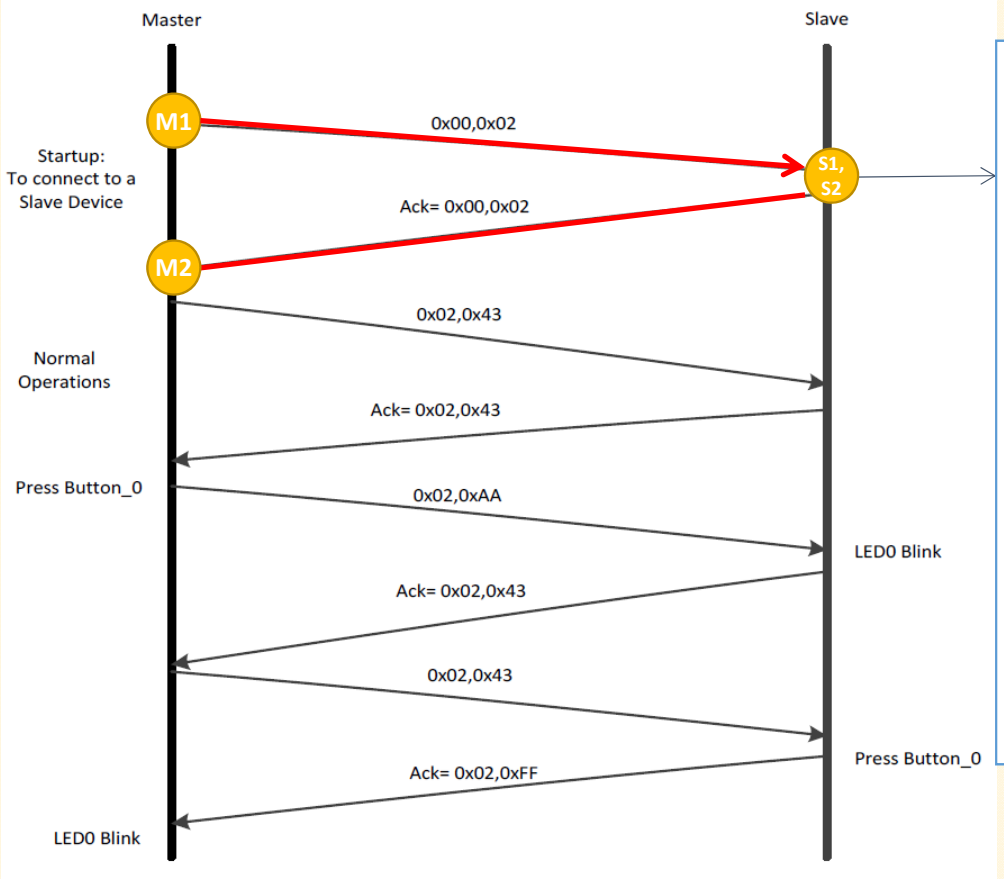
The Slave needs to be at S1 **before** the Master enters M1

Master node: `SCH_Start()`

```
// After the initial (long) delay, all (operational) slaves will have timed out.
// All operational slaves will now be in the 'READY TO START' state
// Send them a 'slave id' message to get them started
Slave_index = 0;
do {

  // Find the slave ID for this slave
  Slave_ID = (tByte) Current_Slave_IDs_G[Slave_index];

  Slave_replied_correctly = SCC_A_MASTER_Start_Slave(Slave_ID);

  if (Slave_replied_correctly)
     {
     Num_active_slaves++;
     Slave_index++;
     }
  else
     {
     // Slave did not reply correctly
     // - try to switch to backup device (if available)
     if (Current_Slave_IDs_G[Slave_index] != BACKUP_SLAVE_IDs[Slave_index])
        {
        // There is a backup available: switch to backup and try again
        Current_Slave_IDs_G[Slave_index] = BACKUP_SLAVE_IDs[Slave_index];
        }
     else
        {
        // No backup available (or backup failed too) - have to continue
        //Slave_index++;
        }
     }
  } while (Slave_index < NUMBER_OF_SLAVES);
```

Handshake protocol: The master sends a "handshake" Tick message to the slave with ID equal to "Slave_ID"

Communication and data sequence



Handshake protocol:
The slave has **accepted** the signal from the master, **sends** a special "handshake" Ack message back to the master and finally **enables CAN SPI interrupts**

Slave node: `SCH_Start()`

```
// Now wait (indefinitely) for appropriate signal from the master
do {
    IOWR_ALTERA_AVALON_PIO_DATA(TEST_1_BASE,
                                IORD_ALTERA_AVALON_PIO_DATA(TEST_1_BASE) ^ TEST1_pin);
    // Wait for CAN message to be received
    do {
        CAN_interrupt_flag = MCP2515_Read_Register(CANINTF);
    } while ((CAN_interrupt_flag & 0x02) == 0);

    // Get the first two data bytes
    Tick_00 = MCP2515_Read_Register(RXBnDm(1,0)); // Get data byte 0, Buffer 1
    Tick_ID = MCP2515_Read_Register(RXBnDm(1,1)); // Get data byte 1, Buffer 1

    // We simply clear *ALL* flags here...
    MCP2515_Write_Register(CANINTF, 0x00);

    if ((Tick_00 == 0x00) && (Tick_ID == SLAVE_ID))
    {
        // Message is correct
        Start_slave = 1;

        // Prepare Ack message for transmission to Master
        MCP2515_Write_Register(TXBnDm(0,0) , 0x00);      // Set data byte 0 (always 0x00)
        MCP2515_Write_Register(TXBnDm(0,1) , SLAVE_ID);  // Slave ID

        /* Send RTS_TXB0_INSTRUCTION Instruction */
        MCP2515_RTS_TXB_Instruction_CMD(RTS_INSTRUCTION_TXB0 );

    }
    else
    {
        // Not yet received correct message - wait
        Start_slave = 0;
    }
} while (!Start_slave);
alt_irq_cpu_enable_interrupts(); //can move to down
```

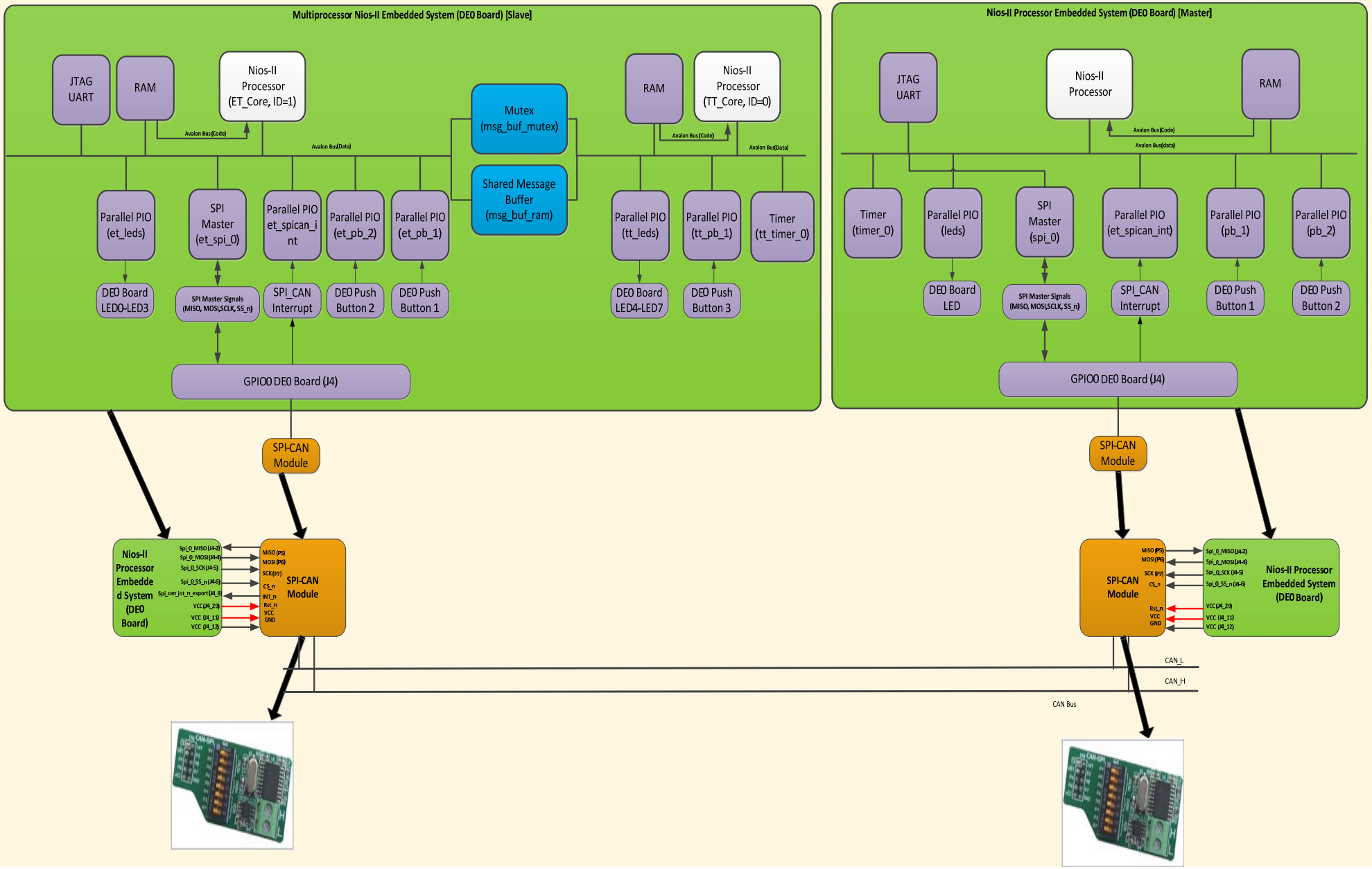
Handshake protocol: The slave has **accepted** the signal from the master

Handshake protocol: The slave **sends** a special “handshake” Ack message back to the master

Enable CAN SPI interrupts

Assignment 2:

Overview


Multiprocessor Nios-II Embedded System (DE0 Board) [Slave]
JTAG UART
RAM
Nios-II Processor (ET_Core, ID=1)
Avalon Bus(Code)
Avalon Bus(Data)
Mutex (msg_buf_mutex)
Shared Message Buffer (msg_buf_ram)
RAM
Nios-II Processor (TT_Core, ID=0)
Avalon Bus(Code)
Avalon Bus(Data)
Parallel PIO (et_leds)
SPI Master (et_spi_0)
Parallel PIO et_spican_int
Parallel PIO (et_pb_2)
Parallel PIO (et_pb_1)
Parallel PIO (tt_leds)
Parallel PIO (tt_pb_1)
Timer (tt_timer_0)
DE0 Board LED0-LED3
SPI Master Signals (MISO, MOSI,SCLK, SS_n)
SPI_CAN Interrupt
DE0 Push Button 2
DE0 Push Button 1
DE0 Board LED4-LED7
DE0 Push Button 3
GPIO0 DE0 Board (J4)
Nios-II Processor Embedded System (DE0 Board) [Master]
JTAG UART
Nios-II Processor
RAM
Avalon Bus(Code)
Avalon Bus(data)
Timer (timer_0)
Parallel PIO (leds)
SPI Master (spi_0)
Parallel PIO (et_spican_int)
Parallel PIO (pb_1)
Parallel PIO (pb_2)
DE0 Board LED
SPI Master Signals (MISO, MOSI,SCLK, SS_n)
SPI_CAN Interrupt
DE0 Push Button 1
DE0 Push Button 2
GPIO0 DE0 Board (J4)
SPI-CAN Module
SPI-CAN Module
Nios-II Processor Embedded System (DE0 Board)
Spi_0_MISO (J4-2)
Spi_0_MOSI (J4-4)
Spi_0_SCK (J4-5)
Spi_0_SS_n (J4-6)
Spi_can_int_n_export (J4_8)
VCC (J4_29)
VCC (J4_11)
VCC (J4_12)
MISO (P5)
MOSI (P6)
SCK (P7)
CS_n
INT_n
Rst_n
VCC
GND
SPI-CAN Module
MISO (P5)
MOSI (P6)
SCK (P7)
CS_n
Rst_n
VCC
GND
Spi_0_MISO (J4-2)
Spi_0_MOSI (J4-4)
Spi_0_SCK (J4-5)
Spi_0_SS_n (J4-6)
VCC (J4_29)
VCC (J4_11)
VCC (J4_12)
Nios-II Processor Embedded System (DE0 Board)
CAN_L
CAN_H
CAN Bus

EG3205: Assignment 2

- Start Date: Tuesday the 27th November
- Submission Date: Tuesday the 18th December
- Demonstration: by Thursday the 13th December
- Work based on Labs 4 and 5:
  - Task1: Develop Hardware
  - Task2: Develop software for Master board
  - Task3: Provide evidence of understanding of the underpinning software
  - Task4: Write a report
- Submit report and two Quartus II projects as one zip file to BB.

Appendix

Lecture 4

Schedulers

- Static (offline) Schedulers:
  - The order of task execution is determined at design time.
- These systems usually rely on
  - a timer interrupt to keep track of the passing of time and
  - dispatch tasks according to the pre-programmed scheduling sequence
- Examples of such systems include clock driven cyclic executive schedulers like time-triggered cooperative (TTC) schedulers.
- Dynamic (online) Schedulers:
  - The order of task execution is determined at run time based on an online scheduling algorithm or on external and internal events.
  - The tasks are run on the basis of their priorities.

M. Hanif, Design and evaluation of flexible time-triggered task schedulers for dynamic control applications, 2013

Time-Triggered Cooperative (TTC) Schedulers

- These schedulers use data structures to keep track of the time units or ticks remaining untill the next execution of each task.

- The time-triggered cooperative (TTC) scheduler given by Pont in 2014 is
  - ➢a statically-scheduled cooperative scheduler
  - ➢that uses a single timer interrupt to control the execution of all tasks.

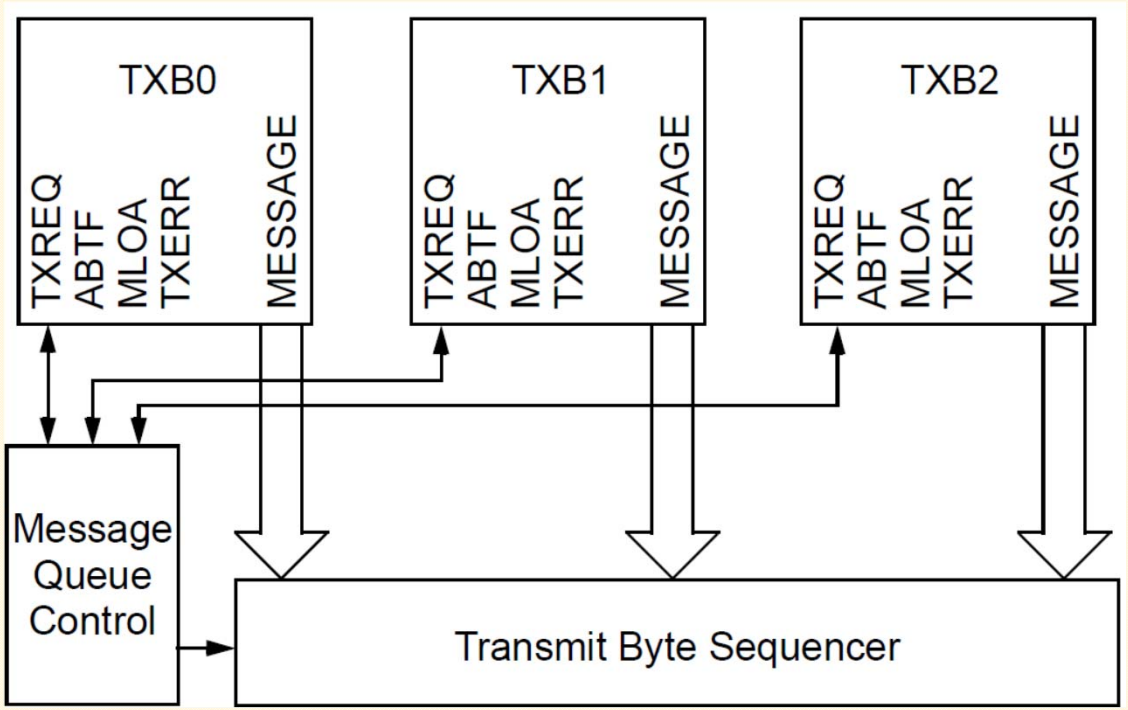
M. Hanif, Design and evaluation of flexible time-triggered task schedulers for dynamic control applications, 2013

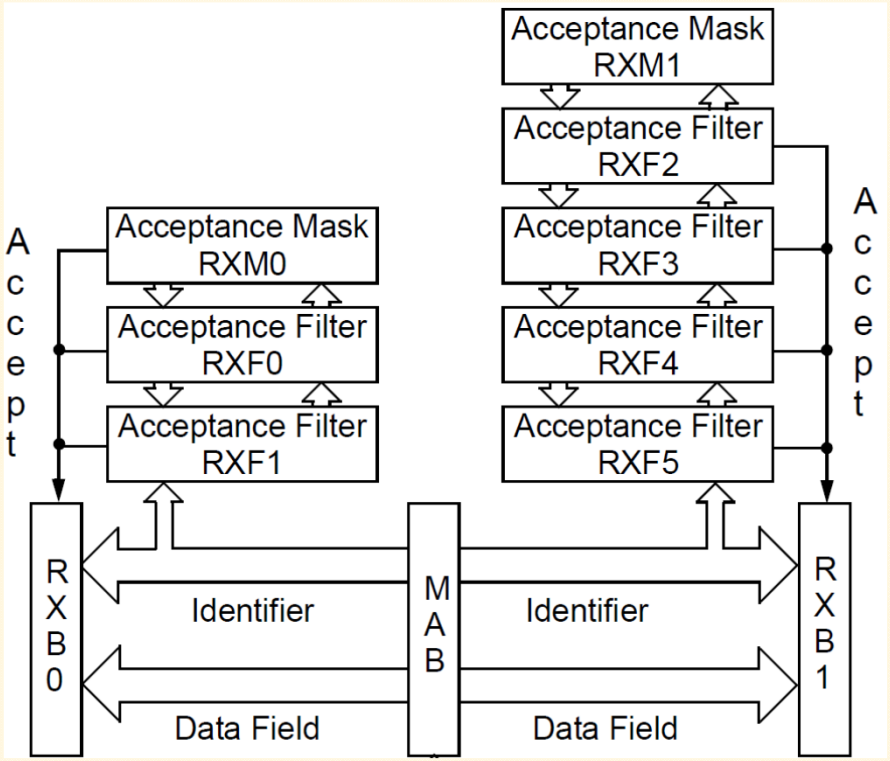
Time-Triggered Cooperative (TTC) Schedulers

- At a lower level, a scheduler can be viewed as a single timer interrupt service routine that is shared between many different tasks.
  - scheduler tick - we have a timer that fires at a certain interval
- The TT co-operative scheduler:
  - Tasks are scheduled to run at specific times.
  - The task runs to completion, then returns control to the scheduler.
  - The time periods of all tasks must be multiples of the base tick period.

M. Pont, Patterns for time-triggered embedded systems, 2014

CAN MCP2515 Buffers





The MCP2515 has three transmit and two receive buffers.

MCP2515 Datasheet, p. 5

CAN MCP2515 Buffers

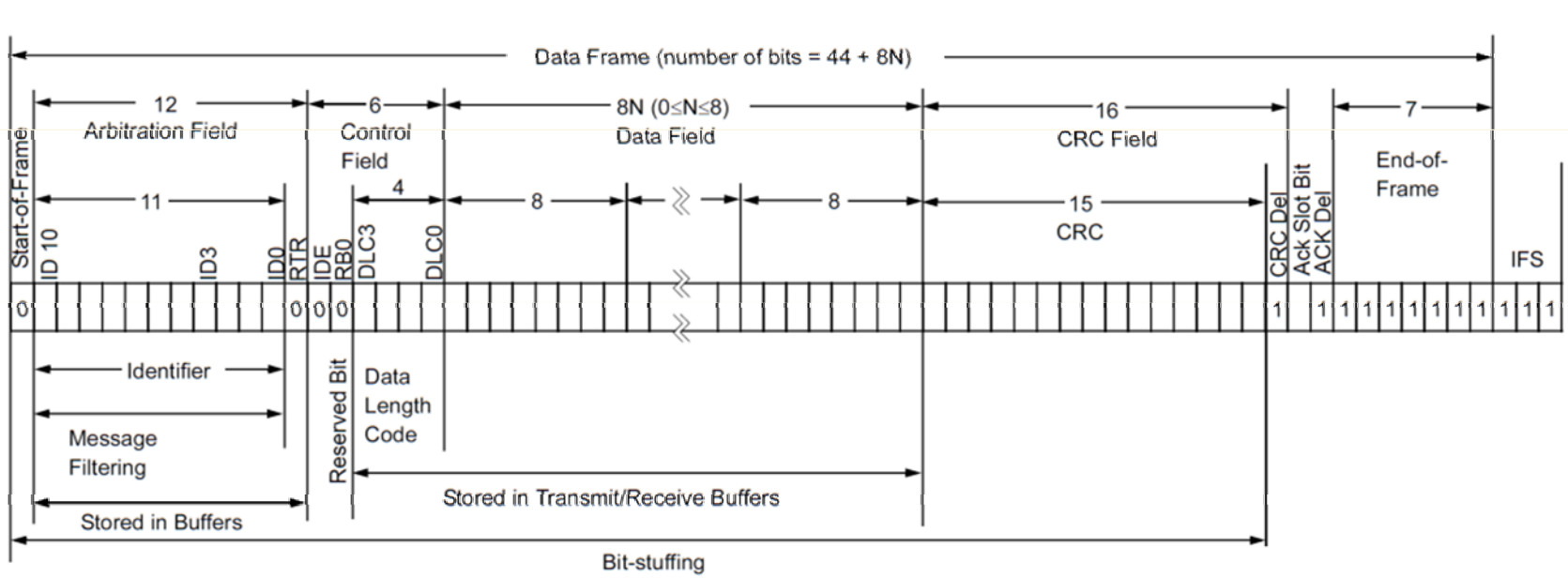
- The MCP2515 implements three transmit buffers.
- Each of these buffers occupies 14 bytes of SRAM and are mapped into the device memory map.
- The first byte, TXBnCTRL, is a control register associated with the message buffer.
  - The information in this register determines the conditions under which the message will be transmitted and indicates the status of the message transmission (see Register 3-2, p. 19).
- Five bytes are used to hold the standard and extended identifiers, as well as other message arbitration information (see Register 3-4 through Register 3-7).
- The last eight bytes are for the eight possible data bytes of the message to be transmitted (see Register 3-8, p.21).

TXBnDm – TRANSMIT BUFFER n DATA BYTE m

MCP2515 Datasheet, p. 5

MCP2515 Standard Data Frame

The MCP2515 is a stand-alone CAN controller developed to simplify applications that require interfacing with a CAN bus.



RTR (Remote Transmission Request) - to distinguish a data frame (dominant) from a remote frame (recessive)
IDE (Identifier Extension) bit, must be dominant to specify a standard frame
RB0 (Reserved Bit Zero) - defined as a dominant bit by the CAN protocol
DLC (Data Length Code), which specifies the number of bytes of data
CRC (Cyclic Redundancy Check) field
CDR Del - recessive CRC Delimiter bit

ACK (Acknowledge) - two-bit field
During the **ACK Slot** bit, the transmitting node sends out a recessive bit. Any node that has received an error-free frame acknowledges the correct reception of the frame by sending back a dominant bit
ACK Del (acknowledge delimiter) - the recessive **ACK Del** completes the acknowledge field and may not be overwritten by a dominant bit.

MCP2515 Datasheet, p.37