Project Acronym: MEDIS

Project Title: A Methodology for the Formation of Highly Qualified Engineers at Masters Level in the Design and Development of Advanced Industrial Informatics Systems

Contract Number: 544490-TEMPUS-1-2013-1-ES-TEMPUS-JPCR

Starting date: 01/12/2013 **Ending date:** 30/11/2016

Deliverable Number: 2.4.5

Title of the Deliverable: AIISM teaching resources - Industrial Networks and Fieldbuses – CAN

Task/WP related to the Deliverable: Development of the AIISM teaching resources - Industrial Networks and Fieldbuses

Type (Internal or Restricted or Public): Internal

Author(s): Mário de Sousa

Partner(s) Contributing: FEUP

Contractual Date of Delivery to the CEC: 30/09/2014

Actual Date of Delivery to the CEC: 30/09/2014

Project Co-ordinator

Company name :	Universitat Politècnica de València (UPV)
Name of representative :	Houcine Hassan
Address :	Camino de Vera, s/n. 46022 - Valencia (Spain)
Phone number :	+34 96 387 7578
Fax number :	+34 96 387 7579
E-mail :	husein@disca.upv.es
Project WEB site address :	http://medis.upv.es/

Context

WP 2	Design of the AIISM-PBL methodology
WPLeader	Universitat Politècnica deValència (UPV)
Task 2.4	Development of the AIISM teaching resources - – Industrial Networks and Fieldbuses – CAN
Task Leader	UP
Dependencies	UPV, MDU, TUSofia, USTUTT, UP

Author(s)	Mário de Sousa
Contributor(s)	Armando Sousa
Reviewers	

History

Version	Date	Author	Comments
0.1	22-04-14	Mario Sousa	Initial draft
0.2	19-09-14	Mario Sousa, Armando Sousa	Final version

Table of Contents

19

1 Executive summary

WP 2.4 details the learning materials of the Advanced Industrial Informatics Specialization Modules (AIISM) related to the Industrial Networks and Fieldbuses.

The contents of this package follows the guidelines presented in the Partner's documentation of the WP 1 (Industrial Networks and Fieldbuses)

1. The PBL methodology was presented in WP 1.1
2. The list of the module's chapters and the temporal scheduling in WP 1.2
3. The required human and material resources in WP 1.3
4. The evaluation in WP 1.4

During the development of this WP a separate document has been created for each of the chapters of the Industrial Networks and Fieldbuses Module (list of chapters in WP1.1). This document is for the fifth chapter – CAN Bus.

In this document, section 2 defines the lecture, sections 3 and 4 describe the laboratory work for weeks 9 and 10, sections 5 and 6 explain the seminar topics for weeks 9 and 10, and sections 7 and 8 define the requirements to fulfil for the mini-project during weeks 9 and 10. Section 9 lists the bibliography and the references.

2 Lecture

2.1 Problem Statement

Up to the current week, you have learned how to set up a Modbus based network, and how to build distributed applications on this network. Notice however that in these applications you always have one or more clients that have to ask the servers for an update of the data. This is basically a pull model, where the parties interested in the data have to pull it off from wherever it resides. If the client wishes to be notified that a data point has changed, it must poll the server continuously until it detects that state of the data point changing.

The above communication model is well suited to many industrial control situations, where a centralised controller communicates with several devices distributed among the plant floor. There are however other applications to which this model of data transfer is not well suited. The most glaring example is that of building automation. In this scenario, the toggling of a single light switch might result in many light fixtures changing state – for example, a light switch at a multi-purpose conference hall in a hotel may be used to set the hall lighting to a dinner mode where only the indirect up-lights are switched on, along with some specific spot lights for wall decorations (paintings) as well as plants/statues. Another switch may be used to set the lighting to a presentation mode, where all lights are switched off, except for the lights illuminating the entrances and exits, and yet another switch for conference mode, where all direct lights are switched on.

In this situation, each specific light fixture may need to switch on or off when specific set of switches are toggled. Note too that each light fixture may have a different configuration to all other light fixtures (or group of light fixtures). Note that the client/server communication model is not well adapted to this scenario, as it would require that each fixture work as a client, and each switch as a server. Every client would need to

continuously poll every server, resulting in many almost useless messages as the same data is continuously repeated, both in the time domain (the same data is sent repeatedly over time), as well as in the space domain(the same data for the same time instant is sent multiple times to each client).

A new communication paradigm, called publish/subscribe, is better suited for the communication requirements of the above described application. This is the model used by the CAN protocol that we will now be presenting.

2.2 Introduction to CAN

CAN (Controller Area Network) was developed by Bosch in 1985 with the intention of applying it in auto-mobiles – a CAN network was to replace the many bundles of wires inside a car. Today, it is used in many modern cars to control the many electrical accessories (power windows, seat heating, wing mirrors, etc.).

CAN is a specification for a protocol that falls in the Layers 1 and 2 of the OSI reference model. As you probably remember, Layer 1 specifies the physical aspects of the network connections (transmission medium, medium attachment or connectors, etc.), while layer 2 is the data link layer (physical signalling, media access control).

As a brief overview, one can state that CAN uses serial based communications based on electrical signals broadcast over an electrical cable (the communication medium), to which an unspecified maximum number of nodes may be connected simultaneously. Communication is half-duplex, at a maximum of 1 Mbaud, and with a maximum data payload of 8 bytes per frame.

CAN can therefore be said to be a communication medium with low bandwidth, but its strength lies in its low overhead and simple physical interface, which allows it to be easily implemented on low cost 8 bit micro-controllers. Even with the low bit rate, the small packet size means frames can be transmitted in a quarter of a millisecond, which is sufficiently fast for practical all use cases in the industrial automation environment.

What distinguishes CAN from most other networks is that the nodes connected to the CAN network do not posses unique network identifiers (or network addresses), so message routing is not based on the identification of the sender node, nor on the receiver node. Instead, identifiers are used to uniquely identify data (or message streams). Every data packet has a unique ID – the ID of the data stream to which it belongs. The network node on which the data stream originates sends the data packet with the appropriate ID out onto the network, while every other node listens to the network and reads that same data packet. Each receiver node then either accepts or discards the data, depending on whether it is interested or requires that data.

This data passing mechanism constitutes the publish/subscribe interaction model. A single data producer broadcasts data to every subscriber interested in that same data. This interaction model is efficiently implemented on networks whose communication medium is broadcast based (which is the case of the electrical wire used in CAN), since the same data can be sent simultaneously to multiple subscribers. Implementation of the publish/subscribe model on networks that are not based on a broadcast medium requires that the same data be sent multiple times, once for each subscriber.

As was said previously, CAN only describes protocols up to layer 2, which is insufficient for most applications. For this reason CAN is commonly used together with an upper application layer communication protocol. Although several application layer protocols have been developed and standardised for use on CAN (e.g. DeviceNet, CAN Open, CAN

Kingdom, J1939), often developers opt to develop a new application layer protocol specifically targeted towards the objectives at hand. The next chapter in this module will focus on the CAN Open communication protocol.

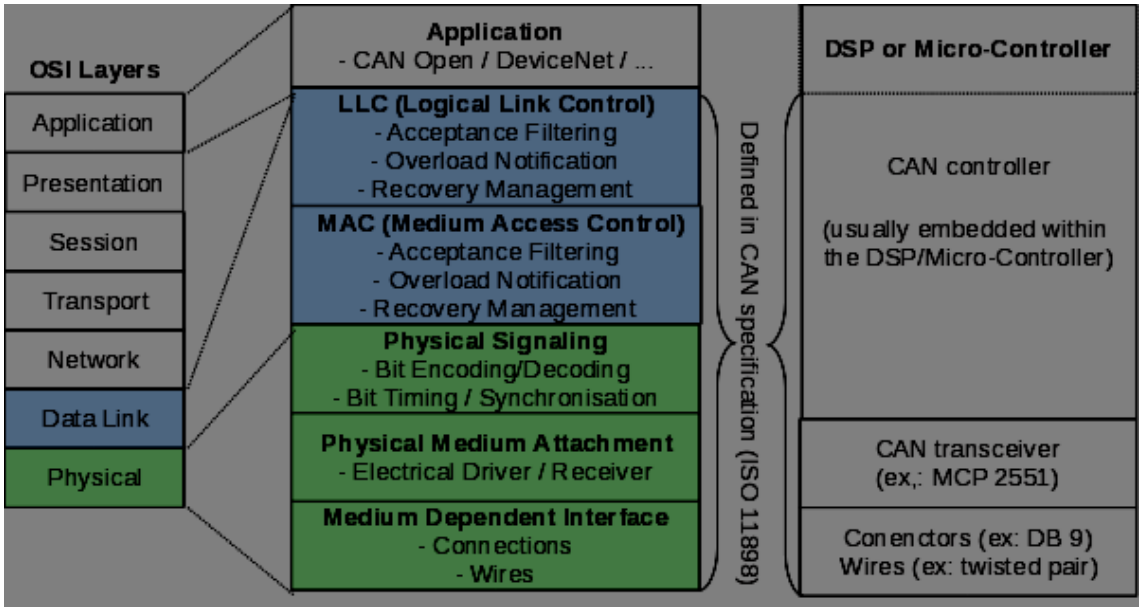
The remainder of this chapter will explain the detailed workings of the CAN communication layers.

2.3 Implementing CAN

A CAN network is based on a pair of electrical wires that constitute the communication medium, and network nodes connected to these wires. The network should follow the bus topology.

Each network node typically contains a generic micro-controller with an embedded CAN controller, along with a CAN transceiver circuit.

The software running on the DSP implements the application layer running over the CAN network. The embedded CAN controller typically implements both sub-layers of layer 2 (Logic Link Control, and Medium Access Control), as well as the physical signalling of the physical layer (bit encoding/decoding, bit timing/synchronization). An independent CAN transceiver circuit implements the electrical amplification and driving of the bus – it converts the 1s and 0s from the CAN controller into electrical pulses leaving a node, and then back again for the messages arriving at the node. The standard medium dependant interface are the bus wires, and the physical connectors (DB9) between the CAN transceiver and the bus wires.



The bus itself should be a twisted pair cable with an characteristic impedance of 120 ohms. In order to remove signal reflection at both ends of the bus, each end must be terminated by an 120 ohm resistor. One of the lines is named CANH (CAN High), and the other CANL (CAN Low).

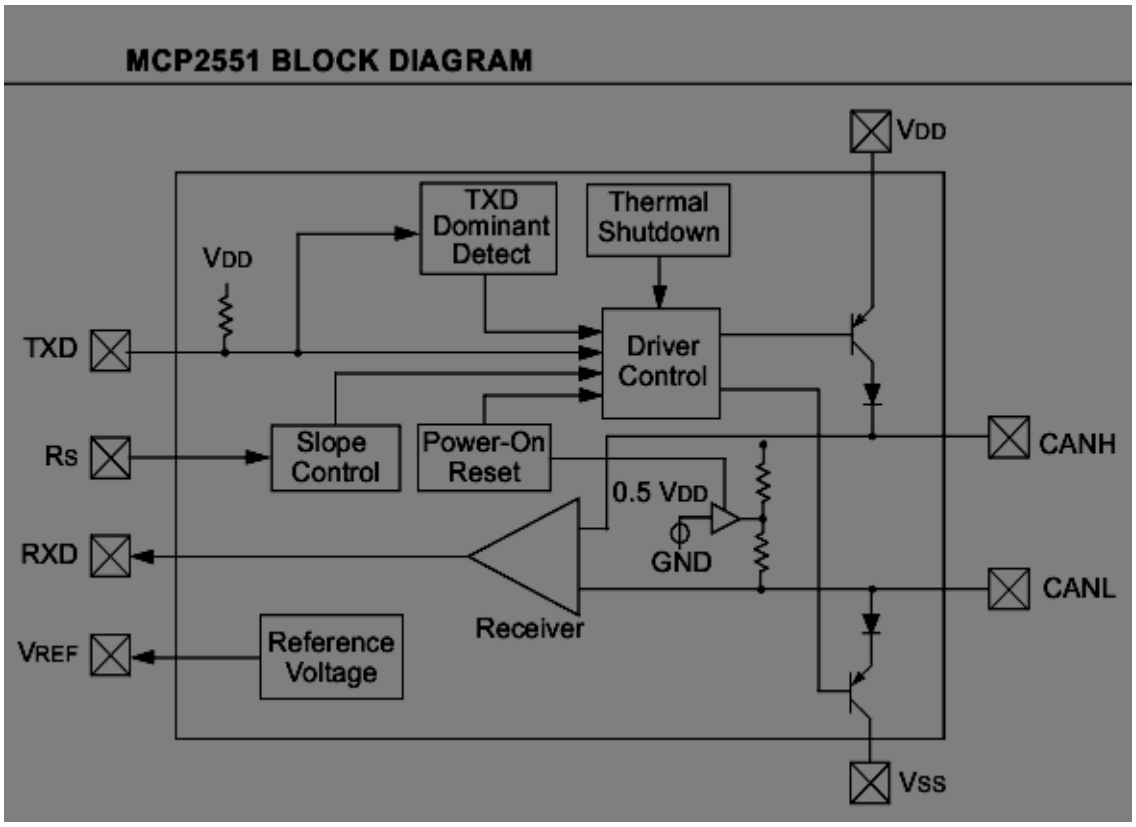
Transmission on these lines is done differentially, i.e., it is the voltage difference between the lines that indicates a '0' or a '1'. When both lines are at the same voltage level (or between -1V and +0.5V at the receiver input), then a logic '1' (recessive)) is considered on the line. When the voltage between the two lines is between 0.9V and 5V, then a logic '0' (dominant) is being transmitted. Note too that data transmission uses the NRZ (Non-Return to Zero) method of bit coding. This means that a logic '1' is transmitted by simply maintaining the bus lines at the correct electrical value for a logic '1' during the time corresponding to the length of a bit. The same occurs for a logic '0', where the bus lines are maintained constant at the correct voltage during the bit interval. (Another alternative method for bit coding, which is not used by CAN, is called Manchester encoding. In this scheme each bit is transmitted as a change of state in the bus. For example, a logic '1' could be transmitted by a change in the electrical value in the bus from 0V to 5V, whereas a logic '0' would be transmitted by changing the state of the bus from 5V to 0V).

An example of a CAN transceiver circuit is the MCP 2551 integrated circuit, whose block diagram may be found in figure XXX. Note that the 'driver control' block is followed by two transistors in the push-up (CANH line) or pull-down (CANL) line configuration. When neither of these transistors is conducting, the CAN bus is in the recessive (logic '1') state. When both conduct simultaneously, the transceiver is writing a logic '0' (dominant) on the bus. Note that the transceiver has no way of forcing both lines to the same voltage when it wishes to transmit a logic '1' – it can merely leave the bus lines at their natural voltage levels, which will tend to be the same voltage due to the two 120 ohms terminating resistances on the bus line.

This means that if two nodes, each with its own transceiver, simultaneously transmit onto the bus, one transmitting a '0', and the other transmitting a '1', the bus lines will be forced to the dominant state, a logic '0'. When both transmit the same value '0' (or the same value '1') then that value '0' (value '1') will show up on the bus.

The bus itself may therefore be considered to be doing an AND operation – if all nodes that are transmitting at a specified time transmit a logic '1', then the logic '1' will show up on the bus. If just one node transmits a logic '0', then the bus will go to logic '0', no matter what the other nodes transmit onto the bus. Since this AND operation is being done on the bus itself, this is commonly referred to as a wired-AND.

This property of the CAN bus is the basis of the medium access control algorithm. It is also used to allow receivers of a data frame to acknowledge correct receipt of that frame at the same time the data frame is being transmitted, therefore saving the need for a separate acknowledged frame, and saving considerable bus bandwidth.



(figure from Microchip application note AN228 – A CAN Physical Layer Discussion)

2.4 Logical Link Control

Data frames on a CAN network may have one of two formats. The first format, defined in CAN version 1.2, uses an 11 bit message (data stream) identifier providing the capability of uniquely identifying 2048 data streams (or data messages). From 1992 onwards, CAN version 2.0 allowed the use of a new data frame format with a 29 bit length identifier, providing up to 56 million unique identifiers. Both specifications are still in use, and may simultaneously shared the same CAN bus. The original 1.2 specification is known as Part A, while the new 2.0 version is called Part B.

SOF	**Arbitration**		**Control**			**Data**	**CRC**	**ACK**	**EOF**	**IFS**
1 bit	11 bits	1 bit	1 bit	1 bit	4 bits	0..64 bits	16 bits	2 bits	7 bits	>=3 bits
SOF=0	Identifier	RTR	IDE	r0	DLC	Data	CRC	ACK	=0	=1

Standard (Part A) CAN frame (11 bit Identifier)

SOF	**Arbitration**					**Control**			**Data**	**CRC**	**ACK**	**EOF**	**IFS**
1 bit	11 bits	1 bit	1 bit	18 bits	1 bit	1 bit	1 bit	4 bits	0..64 bits	16 bits	2 bits	7 bits	>=3 bits
SOF=0	Identifier	SRR	IDE	Identifier	RTR	IDE	r0	DLC	Data	CRC	ACK	=0	=1

Extended (Part B) CAN frame (29 bit Identifier)

Every data frame, be it either Part A or Part B, starts with a Start Of Frame (SOF) field, which consists of a single bit '0'.

In the Part A frames the next field is the Arbitration Field, which consists of the Identifier (11 bits) followed by the RTR (Remote Transmission Request) bit. The identifier is sent starting with the most significant bit (ID10), and ending with the least significant bit (ID0). The seven most significant bits (ID10 to ID4) cannot all be '0'. In data frames, the RTR has the value '1'.

The control field follows, with a length of 6 bits – two bits reserved for future use (r0 and r1), and the 4 bit DLC (Data Length Code). Of the two reserved bits, r1 was later redefined the IDE (Identifier Extension) bit in the extended CAN Part B. The reserved bits must be sent with the value '0', whereas the DLC bits indicate the number of following data bytes in the data frame, with the most significant bit (DLC3) being transmitted first.

Next come the data filed, consisting of up to 8 data bytes. Note that this field may actually be empty (length of 0). All data bytes are transmitted MSB (Most Significant Bit) first.

The CRC (Cyclic Redundancy Check) Field contains a CRC sequence, followed by a CRC delimiter (a single '1' bit). The CRC sequence is used to detect errors in the transmission and receipt of data frames, and uses a cyclic redundancy code optimized for frames smaller than 127 bits. The CRC sequence is calculated using standard CRC polynomial division arithmetic, with all bits from SOF (inclusive) to the last data bit being divided by the generator polynomial:

X15 + X14 + X10 + X8 + X7 + X4 + X3 + 1

As usual, this CRC sequence can be easily obtained using a 15 bit shift register CRC_RG (with bits from 14 to 0) and the following standard algorithm (taken from the CAN standard):

```
CRC_RG = 0;
repeat
    CRCNxt = NxtBit XOR CRC_RG(14);
    CRC_RG(14:1) = CRC_RG(13:0);  // shift left
    CRC_RG(0) = 0;
    if CRCNxt then
        CRC_RG(14:0) = CRC_RG(14:0) XOR (hex#4599);
    end_if
until (end of data)
```

In the above algorithm, NxtBit represents the bits from SOF to the last data bit. At the end of the algorithm, CRC_RG contains the CRC sequence.

The ACK field has 2 bits, the ACK slot and the ACK delimiter (always '0'). The acknowledge slot is used by the node receiving the frame to acknowledge correct reception to the data frame to the sending node.

All data frames are terminated by an EOF (End Of Frame) delimiter consisting of 7 '0' bits, followed by silence on the bus denoted by IFS (Inter-Frame Space). The Inter-Frame space itself consists of an INTERMISSION (during which no node may transmit) lasting 3 bit interval, followed by a Bus Idle period, of variable length, during which the bus may be used by any node wishing to transmit.

The Part B data frame, with a 29 bit identifier, is similar to the Part A frame. However, the SRR (Substitute Remote Request) and IDE bits are inserted after the first 11 bits of the identifier. The SRR bit will fall in the location of the RTR bit of Part A data frame,

and the IDE bit is used to indicate that an extended identifier will be transmitted. In Part A (11 bit identifier) frames IDE is sent with value '0', whereas in Part B (29 bit identifier) frames, this is sent with a '1'.

2.5 Medium Access Control

Since CAN uses a shared bus, in principle only one node should transmit on the bus at any time. To guarantee that two or more nodes do not write conflicting data onto the bus simultaneously an algorithm is used to decide which node may access the shared medium at any time.

The algorithm starts by stating that any node may only start transmitting a data frame when the bus is idle (i.e. during the Bus Idle period, following an INTERMISSION). In low traffic networks this may be sufficient as the network will spend most of its time in the bus idle state, and any time a node has data to transmit it simply checks the bus status and, finding it idle, starts to transmit.

In networks with more traffic it may occur that when a node (node A) decides it has data to transmit, it may find that another node (node T) is currently transmitting a data frame. Once node T finishes transmitting its data, node A will start transmitting a new data frame right after the INTERMISSION interval. The bus never reaches the bus idle state.

In highly congested networks two or more nodes may decide to transmit a data frame during the period of time that a node T takes to finish transmitting its own data frame. In this case, when the INTERMISSION interval terminates, two or more nodes will simultaneously start transmitting their data frames (fig XX). The medium access control algorithm will then be used to decide which of the nodes will be allowed to transmit the data frame to its completion. This algorithm is based on the fact that the bus works as a wired AND.

All the nodes wishing to transmit will start off by transmitting the SOF bit ('0') onto the bus, which results in the '0' showing up on the bus. All the nodes, while transmitting, also read back the data they are transmitting onto the bus, and will therefore use the SOF bit to synchronise the transmission of their data frames. This first SOF is therefore used to guarantee that all transmitters are perfectly synchronised in the transmission of the following field of the data frame, namely the message ID.

After the SOF bit, the transmitters continue with the most significant bit (MSB) of the message ID they wish to send. If the MSB all the messages IDs being transmitted are '1' (or a '0'), then that is the value that will end up on the bus. In this case, all transmitters correctly read back from the bus the bit they transmitted, and therefore continue onto the following bit of their message ID. If one message ID has a bit value of '0', while another has a bit value of '1', the value that will show up on the bus will be a '0'. In this case, the node that transmitted the value '1' will detect that another node is also transmitting at the same time, and will therefore abort the transmission of its data frame.

This algorithm is repeated for each of the message ID bits. Assuming that two nodes will never transmit messages with the same exact message ID, by the end of the message ID bits, only one node will remain transmitting, and will therefore be able to transmit to the end of the frame. Note that this medium access control algorithm does not require re-transmission in case of collisions, as these are handled in real-time, which saves considerable time and bandwidth.

Note too that this same algorithm also guarantees that messages with frame format Part A and Part B may peacefully co-exist on the same network. When two nodes start to

transmit, one using format A, and another using format B, one will usually grant the other access to the bus, as long as the 11 bit ID is distinct to the 11 MSBits of the 29 bit ID. In the unlikely event that these two values are the same, both transmitters will continue to transmit the RTR/SRR and IDE bits after the 11 bit ID (SRR in Part B, and RTR in Part A). Note that in Part B frames the SRR bit will always be '1', and the transmitter of the part A frame will also set RTR to '1', so both transmitters may continue with the frame transmission, and send send the IDE bit. However, here the Part A will send a '0', and Part B frame transmitter will send a '1', which will result in the transmission of the longer 29 bit ID Part B frame being aborted.

When two Part B frames are transmitted simultaneously, the medium access control algorithm extends to the end of the 29 but ID.

We can therefore conclude that the 11 or 29 bit ID, besides uniquely identifying the data message, will also be used a a priority level to decide, in case of collision, which of the messages gets to transmit first. Remember too that each transmitter will typically transmit several messages with distinct message Ids, which means that it will not always be the same transmitting node that will gain the access to the shared medium.

2.6 Error Handling

When a node transmits (publishes) a data frame, the correct receipt of this frame is acknowledged by all remaining nodes on the CAN bus. This is done during the transmission of the data frame itself, once again by taking advantage of the wired AND that is the shared bus. Compared to traditional error recovery techniques that require the receiving nodes to transmit an acknowledgment frame, this technique is considerably more bandwidth efficient.

As explained previously, the node transmitting the data frame will transmit a CRC sequence in the CRC field. The transmitter can either calculate this CRC sequence either before starting t transmit the frame, or it can do it in real-time while it sends the data frame itself. On the other hand, all receivers should determine the expected CRC sequence in real-time, as they receive each bit in the data frame. When receiving the CRC sequence, each receiver will compare the expected value (that they calculated) to the value actually received.

Each receiver that finds that the two values of the CRC sequence match, will acknowledge the receipt of the data frame. This is done during the acknowledge field of the same data frame. The transmitter will send a logic '1' in the 'acknowledge slot' bit, whereas every receiver that wishes to acknowledge the correct receipt of the frame will write a single '0' value over that same bit. The transmitter can therefore check whether the frame was correctly received by reading back from the bus if the logic '1' was overwritten by a '0'.

For a receiver, a message is considered valid if the whole data frame is received correctly, except for the last end of frame bit whose value is ignored. For a transmitter, a frame is considered correct if all the transmitted values show up correctly on the bus. In case of error during frame transmission, the transmission is aborted and a new attempt to re-transmit the same message is made. Note that this new attempt may itself need to be suspended in case another transmitter also starts to transmit another data frame with a higher priority message.

In order to ease clock synchronisation between the nodes on the bus, bit stuffing is used. Clock synchronisation is required because bit coding uses the NRZ method.

As was explained above, NRZ encoding merely means that a bit is transmitted by merely forcing the bus to a specific voltage level during a time interval corresponding to the bit length. What this means is that for some messages the data bus may remain at the same voltage level for relatively long periods of time. For example, if the 8 data bytes being transmitted are all be the same value (for example, '0'), the receiver nodes will merely see the bus always at the same '0' value for a a length of time corresponding to 64 bit lengths. During this time the clocks of the transmitters and receivers could become slightly de-synchronised. If the clock in the transmitter is slightly faster, the transmitter could consider having sent all 64 bits, while the receiver would consider having read only 63 bits.

Since the clocks can be synchronised whenever a change of state (change of electrical value) in the bus occurs , it is necessary that the bus change state relatively often. Bit stuffing is a scheme where additional bits, that do not represent data, are inserted into the the stream of bits written to the bus, with the sole intention of forcing changes of state in the bus. CAN uses bit stuffing after every 5 bits of identical value. For example, if a transmitter sends 5 consecutive bits at the logic level '0', it will then send another '1' bit which will be simply ignored by every receiver. Note that this means that the total length of time it takes to transmit a data frame will depend not only on the number of data bytes in the payload, but also the value of those bytes.

Bit stuffing is applied to all the fields from start of frame to the end of CRC sequence. Bit stuffing is not used in the transmission of the remaining fields of the data frame (CRC delimiter, acknowledge field, and end of frame).

Whenever any node detects an error condition (bit error, bit stuff error, CRC error, form error, or acknowledgement error), that node will transmit an error frame. This error frame merely consists of an error flag (6 consecutive logic '0' bits) followed by an error delimiter (8 consecutive logic '1' bits). Note that bit stuffing is not used in the error frame.

The above rules mean that when a data publisher sends a data message that is not acknowledged (because, for example, the subscriber for that data is currently switched off), then the acknowledge bit will not be overwritten with a '0', and the publisher will detect this error. The publisher will then transmit an error frame, followed by another attempt to transmit the same data frame. For each data frame that is not acknowledged, the publisher will increment an internal error counter. The publisher will continue sending data frames followed by error frames until either the subscriber acknowledges a data frame, or the error counter reaches an error limit. This error limit is internally set by the CAN protocol, and when it is reached the node places itself in a 'bus-off' state, where it isolates itself from the CAN bus. This protocol prevents a single node to block all communication on the bus.

2.7 Publish/Subscribe Handshaking

The data frame previously described is used by publishers wishing to broadcast new data to all subscribers. Although it can be said that this is the most often used method of sending data in CAN, the CAN protocol also allows subscribers to solicit an update of the data from the data publisher.

This is achieved through the use of the 'remote frame', which exists in both the standard Part A, and the extended Part B formats. The remote frame is sent by the data subscriber, and functions as a request that the data publisher should transmit the respective data message.

The remote frame itself is identical to the data frame, with the distinction that no data is sent (data size is 0 bytes long), and the RTR (Remote Transmission Request) bit is sent as a logic '1'. It is in fact the RTR bit which allows any receiver to distinguish a data frame from a remote frame using the exact same message ID. Note too that the fact that RTR is sent as '1' in remote frames, and as a '0' in data frames means that if both the subscriber and the publisher simultaneously start transmitting a data frame and a remote frame with the same message ID, the medium access control algorithm will determine that the data frame will be allowed to transmit, and the remote frame will be aborted.

Another distinction between a remote frame and a data frame is that whereas the DLC (Data Length Code) is used to indicate the number of data bytes in the data frame, in a remote frame it functions as a request for the publisher to publish that many data bytes.

The CAN protocol also includes a third frame, the overload frame, which is used by subscribers to request that all publishers temporarily suspend the sending of new data messages, as the subscriber in question is currently in an overload situation – i.e. it has too many tasks at hand, and it will not be able to process any data message should it arrive.

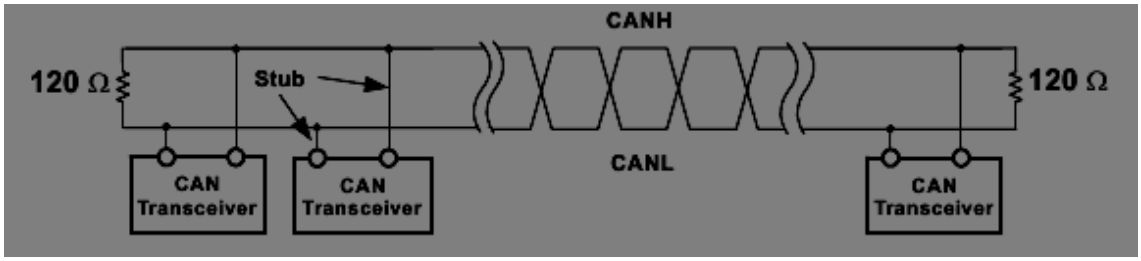
A node may only start sending an overload frame during the first bit time of an expected 'intermission' (i.e. the 3 bit interval during which a bus should remain idle at the end of data or remote frames). At most two consecutive overload frames may be sent, which means that data or remote frames may be delayed at most the length of time it takes to transmit these two overload frames. Any further requests for delay is not supported by the CAN protocol.

The overload frame itself is very simple, as it consists only of an 'overload flag' followed by an 'overload delimiter'. The overload flag consists of 6 consecutive logic '0' bits (with no bit stuffing). The 'overload delimiter' is simply 8 consecutive bits at logic '1'.

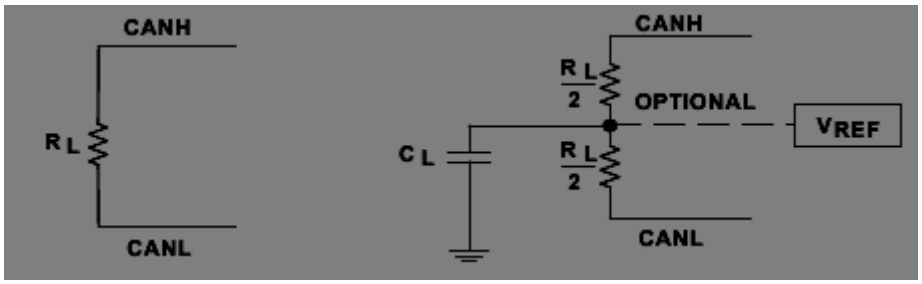
2.8 Physical Layer Requirements

The cable for the shared bus should be a twisted pair with a characteristic impedance of 120 ohm. Nevertheless, almost any cables should work as long as long distances and fast signalling rates are avoided. The use of shielded cables is not required, but it is recommended in electrically harsh environments. Remember that when using a shielded cable, the shield should be connected to the ground terminal of each node, and a single node should be connected to ground itself so as to avoid the existence of ground loops through parasitic currents may flow.

In order to remove signal reflection at both ends of the bus, each end must be terminated by an 120 ohm resistor. This may be done by either a simple terminator, or through a split terminator (two 60 ohm resistors in series) with a high frequency filtering capacitor connected to the mid point and ground. On a high speed CAN network, a typical value for the capacitor for is 4.7nF.



(figure from Microchip application note AN228 – Control Area Network Physical Layer Requirements)



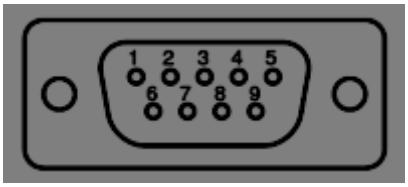
(figure from Microchip application note AN228 – Control Area Network Physical Layer Requirements)

Signal reflection may also occur in the stubs connecting the nodes to the CAN bus. In order to limit this interference, the standard recommends that the stub lengths should not exceed 0.3m when using a 1 Mbaud transmission rate. However, for a typical twisted pair transmission line with a propagation delay of 50ns/m, and a typical transceiver with a 50 ns signal transition time (rise or fall, whichever is faster), stubs may be safely extended up to 1.66m. By slowing down maximum transceiver transition times, even longer stub lengths are possible.

The CAN standard does not specify any requirements for the connectors, although connectors with the same characteristic impedance as the bus line should be used. Upper layer protocols using CAN (such as Devicenet and CAN Open) do specify the physical connectors to use. The CAN Open protocol, for example, specifies several alternative connectors and their respective pin assignments including DB9, RJ10, RJ45, M12, and a 5 pin mini connector.

Pin	Name	Description
1	Reserved	
2	CANL	CANL bus line
3	V+	Optional 3.3V or 5V supply for transceivers on the nodes
4	Reserved	
5	CAN_SHLD	Optional shielding
6	V-	Ground
7	CANH	CANH bus line
8	Reserved	
9	V+	Optional 3.3V or 5V supply for transceivers on the nodes

CANOpen DB9 connector wiring



The CAN standard allows for up to 30 nodes on a CAN bus, but more nodes may be connected if a transceiver with a high input impedance is used. As a result, higher layer protocols actually allow more nodes in a bus – 64 for Devicenet, 127 for CAN Open, and 255 for CAN Kingdom.

The maximum bus length depends on the signalling rate. This is due not only to electrical steady-state losses, but also to bandwidth limitations of the line that may introduce degradation of signal quality and inter-signal interference. However, the most restrictive limitation is typically due to the transmission delay in the line, and the fact that the CAN protocol is based on the line implementing a wired AND.

Line transmission delay is really the speed at which electrical signals propagate on the line, which is related to the speed of light. When a signal is sent by the transceiver of a node at one end of the bus line (for example, a logical '1'), it will take a non negligible amount of time to reach the other end of the bus. The node at the other end may wish to overwrite the bit being sent with a '0', which in turn must also travel the whole length of the bus line to the transmitting node, so it may recognize that the logical '1' that it wrote was overwritten by another node on the bus. This means that each bit that is transmitted must remain on the bus line at least the time it takes for electrical signals to propagate from one of the bus to the other and back again. This time is directionally proportional to the line length, which means that the maximum signalling rate of the CAN bus is also inversely proportional to the bus length.

For a typical twisted pair line, the suggested maximum signalling rates may be found in the following table.

Bus Length (m)	Signaling Rate (Mbaud)
40	1
100	0.5
200	0.25
500	0.1
1000	0.05

A conservative rule of thumb for bus lengths over 100m is to keep the product of the signalling rate in Mbps and the bus length in meters to values less than or equal to 50.

2.9 Bilbiography

– “Controller Area Network”, Konrad Etschberger, IXXAT Automation GmbH (August 22, 2001), ISBN-10: 3000073760 , ISBN-13: 978-3000073762

2.10 Further Reading

– “Introduction to the Controller Area Network (CAN)”, Steve Corrigan, Application Report SLOA101 - August 2002, Texas Instruments

– “Controller Area Network Physical Layer Requirements”, Steve Corrigan, Application Report SLLA270, January 2008, Texas Instruments

– “A CAN Physical Layer Discussion”, Pat Richards, Application Report AN228, 2002, Microchip Technology Inc.
– “The Fast Guide to Controller Area Network (CAN) Application Layers: The Basics of CANopen, DeviceNet™, J1939™ and CAN Kingdom”, John Rinaldi, Vince Leslie, 2003, Real Time Automation
– “CAN Specification 2.0, Part A”, CAN in Automation, Am Weichselgarten 26, D-91058 Erlangen
– “CAN Specification 2.0, Addendum” CAN in Automation, Am Weichselgarten 26, D-91058 Erlangen
– “CAN Specification 2.0, Part B”, CAN in Automation, Am Weichselgarten 26, D-91058 Erlangen

3 Lab – Week 9

The aim of this lab class is to connect two or more devices using a CAN network, and to have them exchange data through the sending and receipt of CAN messages.

3.1 Equipment

- Arduino Board
- CAN shield for Arduino
- Personal computer (PC) with Arduino development environment installed
- cables and connectors to connect CAN network interfaces of CAN shields.
- 120 ohm resistors (network terminators)

3.2 Setup

The Arduino board does not have a CAN interface - this is provided by the CAN Arduino shield, that includes an Microchip MCP2515 CAN controller, as well as the MCP2551 CAN physical transceiver chip. The CAN controller is connected to the Arduino board through the SPI interface:

- pin D11 – MOSI
- pin D21 – MISO
- pin D13 – SDK
- pin D10 - CS (chip select)
- pin D2 - CAN_INIT

The CAN shield also includes two leds, as well as a very small joystick. The leds may be controlled through the pins D7 and D8. The joystick is basically five digital buttons, accessible through the pins:

- pin A1 – up
- pin A3 – down

- pin A5 – right
- pin A2 – left
- pin A4 – centre (click)

Before working with the CAN network, we suggest that you first write, download and test a small program to the Arduino board. This will allow you to later tackle the CAN network with full confidence that the development environment and testing platform are correctly configured and working as intended.

If the Arduino Integrated Development Environment (IDE) is not yet installed, you may download it directly from the arduino website (http://www.arduino.cc/).

Connect the CAN shield to the Arduino board, and the board to your computer. Configure the Arduino IDE with the correct serial port through which to program the board. Load a sample test project into the IDE, compile and download it to the Arduino. At this stage you should also edit the program to make sure you can control the leds on the CAN shield, as well as read the joystick.

3.3 Physical CAN network

Start off by building a CAN network with only two devices by connecting your Arduino to the Arduino of another another group in your class (remember that once this is tested and running, you may later add more devices to your network).

Note however that this CAN shield uses a non-standard pin assignment on the DB9 CAN connector:

- pin 3 – CANH – CAN high
- pin 5 – CANL – CAN Low
- pin 2 – GND

The network cable only requires two conductors: all CANH pins should connect to each other, and the same applies for CANL. Do not forget to connect one 120 ohm resistor between the CANH and CANL lines at each end of the network.

3.4 Sending and Receiving messages

To start transmitting on the CAN network through the CAN controller, it is necessary to first initialise and configure the MCP2515 chip. This can be rather tedious due to the large number of configuration parameters. We therefore suggest that the student use a library to work with the controller chip.

Download the library from https://github.com/Seeed-Studio/CAN_BUS_Shield

This library provides several useful functions and macros, including:

`MCP_CAN CAN(10)`
: Declares that the CAN controller chip is selected on line 10

`CAN.begin(CAN_500KBPS)`
: Initializes the controller chip, and sets the transmission speed.
Other available speeds are:

CAN_5KBPS, CAN_10KBPS, CAN_20KBPS, CAN_31K25BPS, CAN_40KBPS,
CAN_50KBPS, CAN_80KBPS, CAN_100KBPS, CAN_125KBPS, CAN_200KBPS,
CAN_250KBPS, CAN_500KBPS, CAN_1000KBPS

`CAN.sendMsgBuf(INT8U id, INT8U ext, INT8U len, INT8U *data_buf)`

Sends a message over the network

id: message identifier

ext: Frame type (0 → standard frame with 11 bit id)
(1 → extended frame with 29 bit id)

len: number of data bytes in the frame (maximum of 8)

data_buf: pointer to array containing the data to send

`CAN.checkReceive()`

Poll the CAN controller – ask whether a CAN frame has been received

Returns 0 if no frame, and 1 if frame has been received

`CAN.readMsgBuf(INT8U *len, INT8U *data_buf)`

Get a frame that has been received

len: pointer to variable where number of bytes in frame will be stored

data_buf: pointer to array where the data bytes of the frame will be stored

To call and use the above functions, you will need to include the library definition into your project, using

```
#include <mcp_can.h>
#include <SPI.h>
```

You can now write two small programs. One program will run on one Arduino and should periodically send the status of your joystick over the can network (choose any message ID). The other program will run on the other Arduino, and should continually poll the CAN controller (calling checkReceive). When a message arrives, it reads the data and then switches on (or off) the local leds to reflect the status of two of the joystick buttons (example, up and down).

3.5 Transmission Speed

Try running the same exercises, but using a slower and a faster transmission speed.

Is the network able to work at the maximum speed? In this case determine the maximum distance between the two devices at which the network stops working (due to transmission/reception errors).

If the network does not work at the maximum speed, determine the maximum speed at which it will work. What happens if you reduce the physical length of the network?

4 Lab – Week 10

4.1 Event driven messages

Start off with the program you wrote for the previous lab session.

Instead of sending periodic messages on the network, using up precious bandwidth, you should now only send a message when one of the buttons changes state.

Change your sending program to reflect this event driven approach. Two implementation alternatives exist, as you may probably remember from the module discussing micro-controller programming:

- use an external interrupt to warn the processor when the input pin changes
- periodically poll the input pins, to detect changes in their values. The polling itself may either be handled by an periodic timer driven interrupt, or by a cyclic loop in the main program.

The first alternative will only work if the pins to which the buttons are connected can function as external interrupts. This is not the case, which is why it is probably best to poll the inputs inside a periodic interrupt handler. In principle interrupt handlers should work as fast as possible, so calling the CAN library functions from within an interrupt handler is discouraged. One possible approach is to have the interrupt handler set a flag, and the main program run an infinite cycle while(1) loop continuously checking whether this flag has been activated. If so, only then is the function to send the CAN message called.

Below is some sample code for Arduino that will configure a periodic interrupt handler.

```
//timer setup for timer1
#define ledPin 13

void setup(){
  pinMode(ledPin, OUTPUT); //set pin as output
  noInterrupts(); // disable all interrupts
  TCCR1A = 0;
  TCCR1B = 0;
  TCNT1 = 0;
  OCR1A = 31250; // compare match register 16MHz/256/2Hz
//OCR1A = 31;    // compare match register 16MHz/256/2kHz
  TCCR1B |= (1 << WGM12); // CTC mode
  TCCR1B |= (1 << CS12); // 256 prescaler
  TIMSK1 |= (1 << OCIE1A); // enable timer compare interrupt
  interrupts(); // enable all interrupts
}//end setup



//timer1 interrupt
ISR(TIMER1_COMPA_vect){
digitalWrite(ledPin, digitalRead(ledPin) ^ 1); // toggle LED pin
}

void loop(){
  //do other things here
```

```
}
```

4.2 Filtering messages

Besides the previous library functions, the library also includes two functions that will let you configure which frames you are willing to receive. Any frames that do not conform to these requirements are simply discarded by the CAN controller, and are therefore treated as if they did not exist.

```
init_Mask(INT8U num, INT8U ext, INT8U ulData);
init_Filt(INT8U num, INT8U ext, INT8U ulData);
```

Instead of sending all the joystick buttons on the same frame, you can now map each joystick button to a specific message ID. With this approach, the receiver can now configure the CAN controller to ignore all the messages that are not related to the 'up' and 'down' buttons, which are the only buttons currently used for deciding which leds to light up.

4.3 multi-task program

Change the programs written in the previous lab so that they can simultaneously work as both a sender and receiver of information. Two Arduinos running this same program would have the joystick of one control the leds of the other Arduino.

5 Seminar – Week 9

The students are asked to research different application areas of the CAN bus. Either each group should make a summary of all areas they can find, or alternatively each group should focus on one specific application area.

Example application areas: Automotive, avionics, industry, embedded systems, etc.

6 Seminar – Week 10

The students are asked to research different protocols that work over the CAN bus, with special focus on protocols used in the industrial automation field.

Either each group should make a summary of all protocols they can find, or alternatively each group should focus on one specific communication protocol.

Example upper layer protocols: DeviceNet, CAN Open, CAN Kingdom, J1939, etc.

7 Mini-project – Week 9

Integrate your Modbus slave, developed in the previous week, with the code to send and receive data over the CAN network.

Notice that you will need to have two programs running 'simultaneously' that both act as slaves to requests coming over two distinct networks. Design a software architecture that will allow you to do this, implement it, and test it.

Map the data sent over the CAN network onto the address space of the Modbus slave program. You should now be able to press buttons on an Arduino, have this data sent over the CAN network to another Arduino, forward this same data over Modbus serial to the controller, and have the controller respond to the button press. Test this by having a button

on one Arduino act as the STOP button for the process controller running on a PC connected to a second Arduino.

8 Mini-project – Week 10

Connect three or four Arduinos on the same CAN network.

An additional STOP2 button is now added to every Arduino on this CAN network. This STOP2 button, when pressed. Should not only stop the process on the PC to which the Arduino is connected, but also the process of the next Arduino on the network. The STOP2 button on the last Arduino on the CAN network loop around and also stop the process on the PC connected to the first Arduino on the network.

Configure an extra two leds on the Arduino. Use these leds to reflect the status of the RUN led of the two processes immediately before and after the current Arduino. On each Arduino you sould now have a led indicating the RUN state of the process connected to that Arduino, as well as the processes on the previous and next Arduino boards.

Configure the filters and the data transmission on the CAN network to fulfil these additional requirements.

9 References

- "Controller Area Network", Konrad Etschberger, IXXAT Automation GmbH (August 22, 2001), ISBN-10: 3000073760 , ISBN-13: 978-3000073762
- "Introduction to the Controller Area Network (CAN)", Steve Corrigan, Application Report SLOA101 - August 2002, Texas Instruments
- "Controller Area Network Physical Layer Requirements", Steve Corrigan, Application Report SLLA270, January 2008, Texas Instruments
- "A CAN Physical Layer Discussion", Pat Richards, Application Report AN228, 2002, Microchip Technology Inc.
- "The Fast Guide to Controller Area Network (CAN) Application Layers: The Basics of CANopen, DeviceNet™, J1939™ and CAN Kingdom", John Rinaldi, Vince Leslie, 2003, Real Time Automation
- "CAN Specification 2.0, Part A", CAN in Automation, Am Weichselgarten 26, D-91058 Erlangen
- "CAN Specification 2.0, Addendum" CAN in Automation, Am Weichselgarten 26, D-91058 Erlangen
- "CAN Specification 2.0, Part B", CAN in Automation, Am Weichselgarten 26, D-91058 Erlangen