E BLOCKS 2

CAN Bus Communications

CP2793

MATRIX

www.matrixtsl.com

CP2793

CAN Bus Training

Course Notes

Introduction

These notes are designed to introduce you to the concepts required to understand CAN networks and also to provide practical exercises with which to develop your skills as well as those of your students.

The course is structured into a number of sections that first take you through the basics of CAN and then into intermediate topics, such as messages and sending data. The course also deals with some more advanced topics, including the use of masks and filters. Examples and suggested work is provided as a basis for developing demonstrations and practical activities for your students.

These notes provide a framework for teaching CAN to students. How you use them for teaching is up to you. If you are teaching automotive students who do not need to know how to program you can simply make use of the downloadable example programs.

This course is carried out using Flowcode, a graphical programming language. The Flowcode CAN component is designed to allow students to learn about CAN without getting bogged down with the problems of programming in C or a lower level language.

When teaching automotive students about CAN we do not envisage a great deal of programming will take place in Flowcode. However we suggest that the supervisor should have some Flowcode experience for debugging purposes. This can be quickly and easily acquired.

More advanced students will want to use Flowcode extensively. There are a number of tutorial files and resources on the Matrix TSL Flowcode web site that students can go through to help them understand how Flowcode works. Students will find that they can make rapid progress using Flowcode's graphical interface.

This course is designed for use with two levels of student:

Firstly for use with automotive technicians at Level 3 to gain an appreciation of CAN technology and the equipment used in fault finding CAN systems, and how that fault finding takes place. These technicians are expected to download and review programs made in flow charts, but are not expected to carry out any programming tasks.

2. Secondly for more advanced students at Level 4 to gain an understanding of CAN technology and to allow them to construct networks which communicate in CAN and higher level protocols. These students are expected to develop their own CAN networks using flowcharts with CAN macros provided. The extensive use of flow charts will allow students to quickly and easily understand CAN protocols and communication, avoiding the need to become involved with the processes of lower level CAN bus software construction.

1 Demonstrations, Worked Examples and Exercises

There are three kinds of exercises found in the notes; *demonstrations*, *worked examples* and *examples*. We will briefly describe what we mean by each of these.

In the case of E-blocks2 based solutions, all the programs supplied on the CAN Solution CD require Flowcode V8 or higher to be installed on the host PC.

1.1 Demonstrations

Demonstrations are provided that can be used with technicians to see a CAN system in action. All programs will be provided so that they can be programmed into the nodes. No programming is required, but the Flowcode flowcharts are available to students to show how they work. The demonstrations are best used in conjunction with CANKing to allow students to see the messaging in action, and to note the effect different programs have on the network traffic.

Demonstrations can be used to teach the fundamentals of CAN to students who are not required to understand and check CAN systems but will not need programming skills.

1.2 Worked examples

A worked example is provided for the first basic example to allow students to be eased into both CAN and Flowcode. The emphasis here is on getting students ‘up and running’ with a simple CAN system in Flowcode that can then be used as a base of experience for later examples.

1.3 Exercises

Further exercises are provided, along with a set of accompanying notes. The notes give information on setting up various aspects of the solution and can form the basis for handouts.

1.4 Further work

Questions to ponder and suggestions for further work are given with each exercise. This further work can be used as the basis of differentiated student activities, thus meeting awarding body requirements in situations where the CAN solution is being used in conjunction with a formally assessed course.

2 Basic CAN Networking

2.1 Overview

This section is designed to get you up and running with a CAN Network as fast as possible. You will be introduced to messaging and how to send receive a simple signal that can be acted upon. The sample applications will introduce you to several basic CAN features, and will serve as a starting point for further study.

2.2 What is CAN?

CAN – Controller Area Network is a serial network protocol. By which we mean it is a pre-defined way to communicate between different parts of a system. Each part needs to speak the same language, and use a common set of signals and message structures in order to be able to understand the messages and in turn to be understood. CAN is one such system.

Other systems, such as RS-232 are often point to point systems where one device will talk directly to another. A limitation of this system is that you may need to run several different connections to speak to different parts of the system, or one part may need to talk to another but may only be able to do so via an intermediary.

CAN offers a simple solution to this problem. It sends the message to all parts of the system, and lets each part (or *node*) decide for itself if the message is for it or not. Built in *error checking* and responses to messages help prevent lost messages, or jammed systems where the system *hangs* whilst waiting for a response to come in. Also messages can be responded to by multiple devices or even none at all, making it easier to construct a system that only reacts wherever and whenever it needs to.

CAN has various inbuilt systems for *error detection*, and ways to prevent all the nodes trying to talk at the same time. But you never see this, it's handled by the CAN chip behind the scenes. All you need to do is decide what messages to send and receive.

Another benefit of the CAN system is that if you wish to add another part to the system it can often be as simple as programming it to respond to the appropriate messages and wiring it into the network. You don't need to connect it up in any specific place or sequence so you can slot the new node in wherever you want, or wherever the physical system design requires it to be.

CAN was originally designed by Bosch for the Automotive industry, evolving from a need to communicate between the various ECU's (Electronic Control Units) on luxury cars. Since then CAN has grown to become a popular network system, particularly in embedded systems. CAN is used on vehicles such as cars, boats, planes, trucks, and in many other areas of industry. CAN's high speed and robust nature make it particularly suited for industrial or high speed applications.

2.3 Nodes and Networks

CAN systems comprise of two or more nodes connected in a *network* as shown in Fig. 2.1.

The *network* is the data highway to which the nodes are connected. Unlike many other systems where the connections run from device to device the CAN network is free standing. Each node feeds off the network, but does not block it, or prevent the other nodes from receiving the message.

The *nodes* can send messages onto the network, and can listen to the network to see if there are any messages on it. The node can then check the message to see if it should respond to the message, or if it should ignore it.

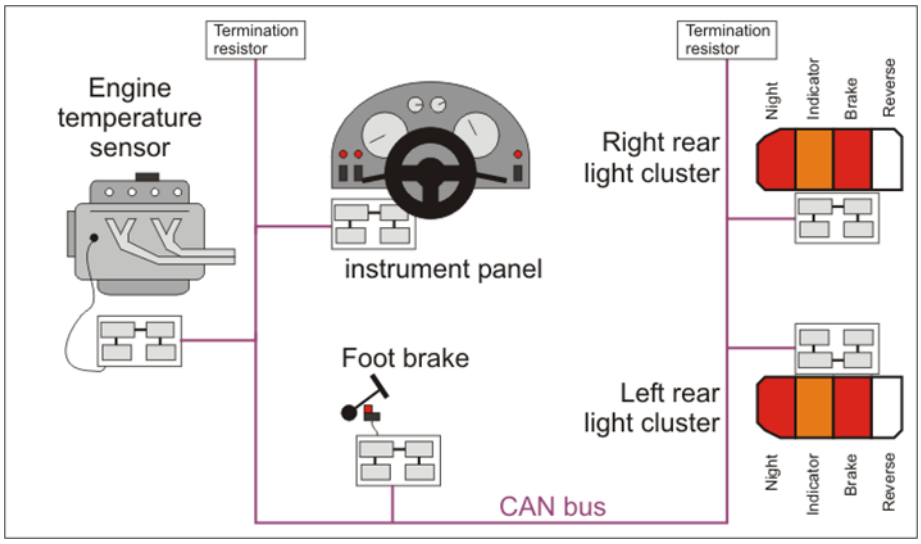


Figure 2.1 *Typical automotive CAN bus arrangement*

Nodes are independent of each other, and can be as simple or as complex as the system designer wishes them to be. A node could be a single light, or a whole dashboard. By virtue of the nodes being independent they can be added, removed or modified without needing to change any other part of the CAN network.

For instance two models of a car may have different dashboards, one a deluxe model with extra features not found on the basic model. The dashboards are both sent the exact same messages by the CAN network; it is what messages they accept, and what they do with them that makes them different. The CAN network does not care if a signal becomes a single light, or a strip of LED's. It does not even care if the 'Fancy RPM display' message is ignored when the basic dashboard is fitted. The CAN network simple puts the messages onto the network, and leaves the nodes to decide whether to use them or not. As there is no change to the messages sent, the CAN network does not need changing to accept the different dashboards so either dashboard can be slotted into the network.

2.4 The physical layer

The CAN specification does not specify the physical signal transportation layer, only the message format. By doing this CAN allows system designers to implement a *physical layer* appropriate to the system rather than having to adapt the system to match a preset physical layer.

This is a very important issue as it means that the way CAN is implemented in the physical layer can and often will differ from system to system. The physical layer for a plant wide heavy industry CAN system is likely to differ in many ways from one built into a luxury car. A CAN system such as the Matrix CAN board is essentially our implementation of CAN using our own Flowcode component and our own CAN board to drive the physical layer. Some parts of the overall system, such as the format of the message sent are an integral part of CAN as defined by the CAN specification. Others such as the RX and TX buffers (discussed later) are part of our implementation of a physical layer for CAN.

2.5 Messages

Messages are the beating heart of the CAN system. Without them it's just a load of redundant wires and circuit boards. Each message consists of an identifier (the Message ID that will become important later on) and a stream of data. Actually, there's more – acknowledgment bits, checksums, transmission details etc. but these are dealt with automatically by the CAN component. All you need to worry about is the ID and the data.

The Message ID's are used to help differentiate messages. A node could accept certain messages, and skip others depending on their ID values. This allows complex interrelated systems to be designed easily where multiple nodes can respond to the same message as easily as a single node can pick out a message that only it will respond to.

Messages can contain data or be completely empty. For many simple signals such as a brake light the act of sending a message may well be enough – a signal is sent and is reacted to. For others e.g. RPM, or temperature readings, data of some sort is required and can be passed along with the message. If data is sent it does not even need to be looked at. An aircraft RPM sensor reading could be used by one node to display the actual RPM, but on another node to simply activate an 'engine running' warning light without even looking at the data.

2.6 Higher Level Protocols

CAN is a *message system*. It is not responsible for the contents of the message. CAN does not care if data is expected, or if the incorrect amount of data is sent. It is not responsible for ensuring that the message is being sent to the appropriate node in the system, only that it is correctly sent to a node. CAN only cares that it is a correctly formed CAN message.

However, we do care about the data. We care about *which node it gets sent to*. We care about these things enough to create *higher level protocols* to deal with these kinds of issues. These protocols sit on top of CAN and help control the flow of messages and data on the network. Higher Level Protocols, or HLPs, are used in CAN systems to perform functions such as system startup procedures, error checking, connection and status monitoring and other administrative tasks.

Using an HLP may involve a CAN node having to send messages asking to be able to speak to another node, and requiring messages to be sent back agreeing to the communication before the real communication can begin. Such systems may seem like a major overhead when you are learning to send CAN messages, but once you understand CAN messages then your mind will automatically start to look for ways to error check and monitor the system. HLPs are the result of this natural progression.

HLPs are the glue that helps keep the system ticking over nicely. For this reason large scale systems will most likely use a HLP. One problem with HLPs is the amount of them – over forty already. Which HLP you use would most likely depend on the company you work for, or the products you deal with.

HLPs are beyond the scope of this course. The diversity makes it difficult to deal with them in detail. And the size and complexity of code needed for a HLP is too much for most basic microcontroller systems to handle. However, if you wish to look into HLPs and how they are used you can find documentation on various CAN HLPs, such as CANOpen, on the Kvaser site www.kvaser.com.

3 CAN training solution

This course is based around a CAN training solution that is set up as a four node network. This provides a *digital input node* (switches), a *digital output node* (lights) and an analogue input node (sensors) together with a monitoring/control node (the dashboard). These four nodes should help you gain an understanding of CAN network tasks within, but not limited to, an automotive context. Not all nodes are required for every task, and for some tasks you may need to reconfigure some of the nodes. However for general training, and to teach the principles the four node network is ideal. A fifth connection point is available, which is used in conjunction with the Kvaser CAN Analyzer to monitor network traffic.

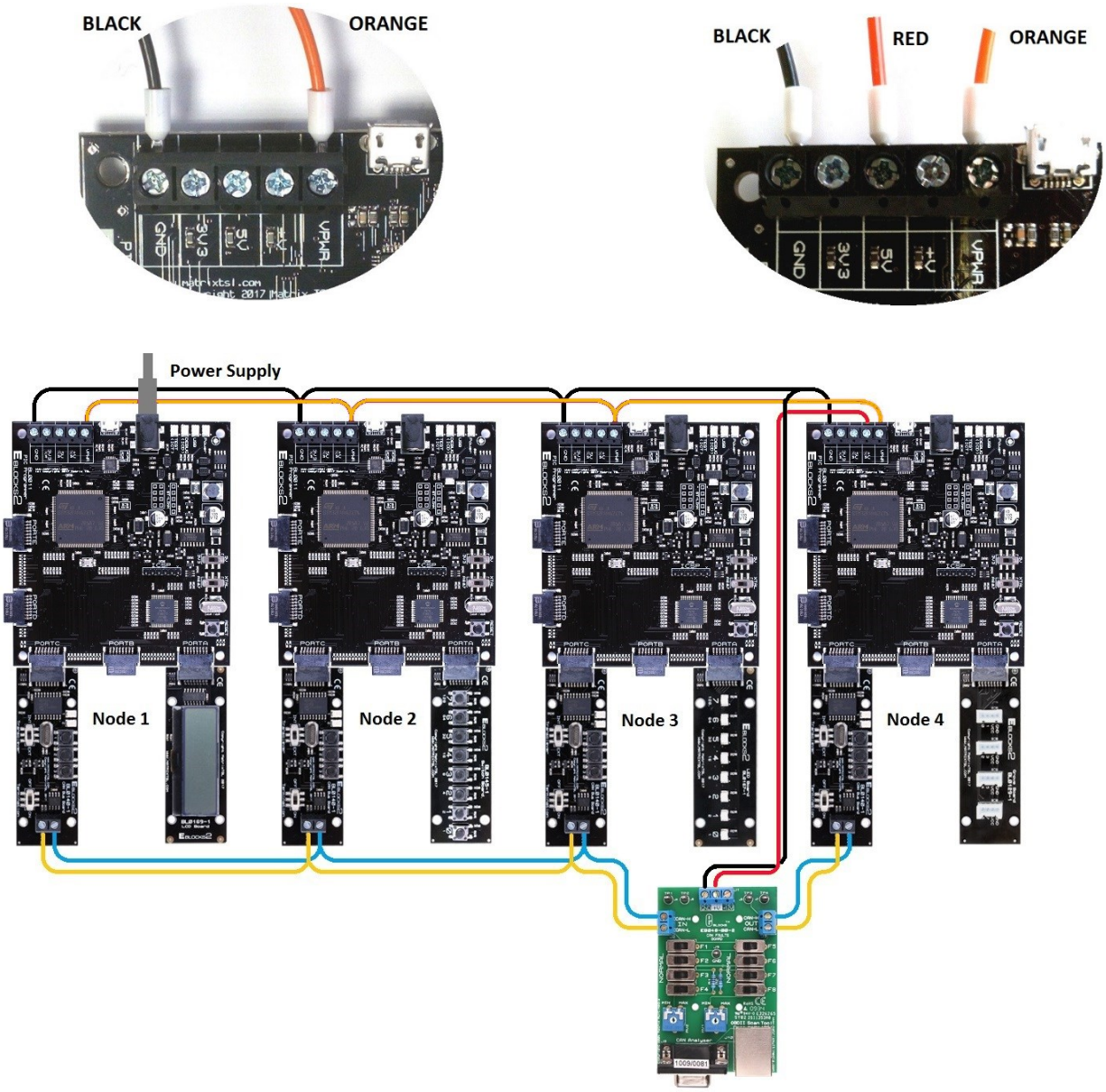


Figure 3.1 *The CAN training solution*

The CAN training solution consists of backplane panels with a node on each panel. The power for the panels can be derived from a single PSU attached to one of the programmer boards and then the VPWR and GND signals looped together as shown above. Only one USB cable is needed as each node requires programming separately. The USB cable can be connected to any of the nodes for programming. However, a second USB port on the PC is needed for the CAN Analyzer.

Important note: Information presented here is correct at the time this document was produced. Please check the Matrix web site www.matrixtsl.com for the latest E-blocks2 documentation.

3.1 The CAN board

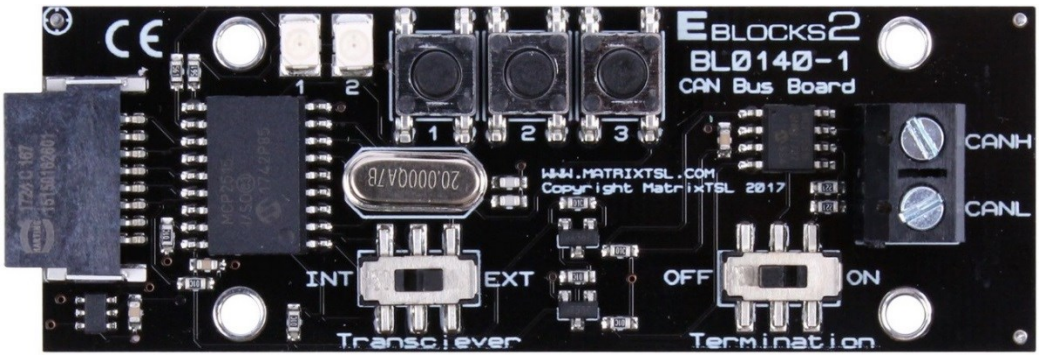
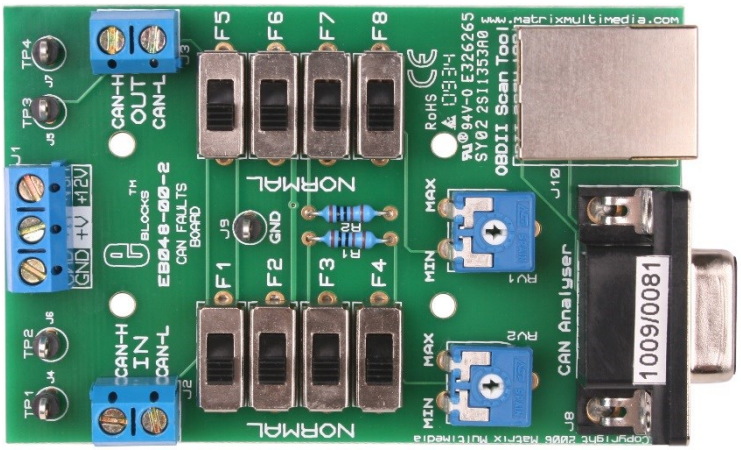


Figure 3.2 *The E-blocks2 BL0140 CAN board*

A multiway connector is used here to connect the CAN controller to the upstream microcontroller. The Transceiver switch should be set to EXT and the Termination switch set to ON for Nodes 1 and 4, and OFF for Nodes 2 and 3. The double screw terminal socket is used to connect the node to the CAN network.

Two features that are specific to this board, and may not be available on other systems, are the two LED's and the three switches. The LED's can be configured as general outputs or as buffer state indicators. The switches can be configured as general inputs or as CAN buffer activation switches.
This is dealt with later on in the course.

3.1.1 CAN faults board



The CAN faults board fulfils three functions: firstly it acts as a point where the Kvaser analyser can be attached to the CAN bus using the D-type connector on the CAN faults board, secondly it allows you to insert a number of faults onto the CAN bus, and thirdly it allows oscilloscope probes to be easily attached to the CAN high and CAN low lines using test pins TP 1 to TP4. The switches marked F1 to F8 allow each of the CAN lines to be placed in four separate fault conditions: short circuit to 5V, short circuit to ground, open circuit and partial open circuit. The potentiometers RV1 and RV2 allow you to vary the partial open circuit resistance. The circuit of the CAN faults board is available in the technical datasheet of the CAN faults board.

3.1.2 Installation

1. Install Flowcode
2. Check for any updates to Flowcode using the Help menu item "Check for updates"
3. The CANLeaf analyzer supplied as part of the CAN solution requires a USB Driver to be installed for it to function correctly. See the Kvaser folder on the accompanying CD for instructions and driver files. IMPORTANT – Do not plug the USB CANLeaf analyzer in until asked to do so by the Install routine.
4. The Kvaser analyzer requires the CANKing software to be installed. A copy of this is on the CD supplied with the CAN training solution. IMPORTANT – CANKing needs to be installed before the Kvaser analyzer or the CAN Analyzer may not be recognized correctly.

3.2 Setting up the CAN system

The basic CAN node we use here consists of either a PIC or Arduino based processor, with an BL0140 CAN board attached.

3.2.1 Setting up the CAN nodes

The CAN solution consists of 4 nodes, plus an attachment node for the Kvaser CAN analyzer:

Node 1) Monitoring and Display control node
Node 2) Input switch node
Node 3) Output display node
Node 4) Analogue Sensor node.
Fitted with sensors: Light (socket 0,1), Rotary (socket 2,3) and Temperature (socket 4,5)

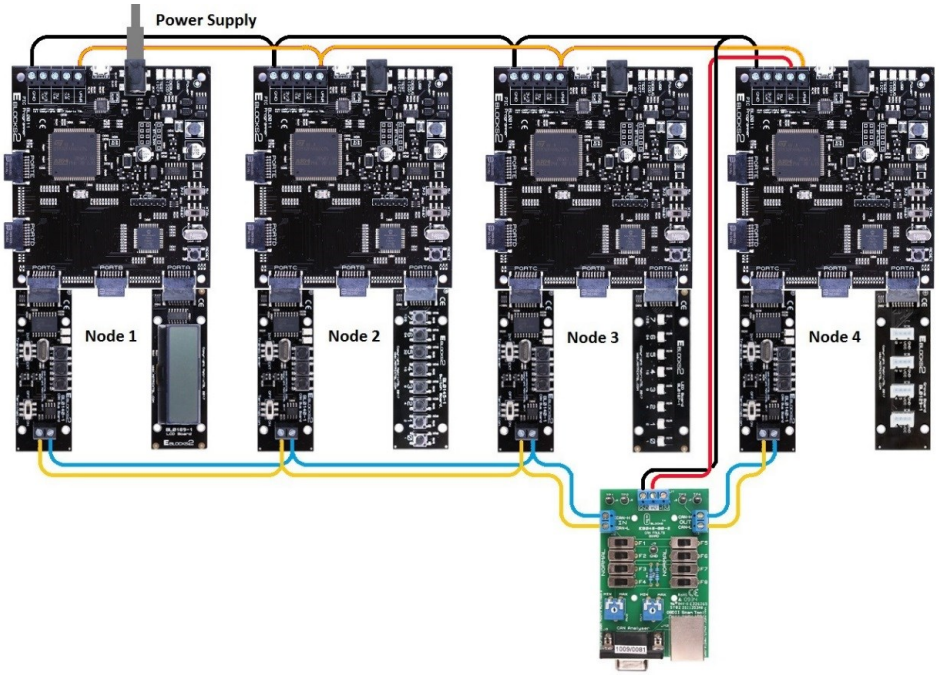


Figure 3.3 *CAN system – wiring of the power connections*

3.2.2 E-Blocks2 board configurations

	PIC BL0011			**Arduino BL0055**		
	Port A	Port B	Port C	A0-5	D0-7	D8-13
Node 1	BL0169		BL0140	BL0169		BL0140
Node 2	BL0145		BL0140	BL0145		BL0140
Node 3	BL0167		BL0140	BL0167		BL0140
Node 4	BL0129		BL0140	BL0129		BL0140

3.2.3 Testing the CAN system

Set up the CAN system with the panels powered up, and the Kvaser analyzer attached and connected to the PC. As supplied the CANH (CAN High) line has a blue wire, and the CANL (CAN Low) line has a yellow wire (however it's always best to check just to be on the safe side). Nodes 1 and 4 are End Nodes. Ensure that the Termination switch is set to ON. For Nodes 2 and 3 the Termination switch is set to OFF.

The test programs are available in Flowcode format and can be compiled and downloaded to the appropriate CAN nodes

- Node 1 – LCD display and low fuel warning
- Node 2 – brake pedal switch
- Node 3 – brake light

Node 4 – fuel sensor board

Once the four nodes have been programmed you can test Node 1 and Node 4 by moving the sensor rotary and watching for a corresponding change on the Node 1 display. Nodes 2 and 3 can be checked by pressing push switch 0 on Node 2 and watching for a signal on LED 0 on Node 3.

3.3 Testing your programs

Because CAN requires two nodes to be useful, and hence two separate programs running at the same time, we can't simulate them in Flowcode. To get around this problem we need to use some kind of analyzer that plugs into the CAN network and monitors the messages sent.
In order for you to be able to monitor and test your programs we have included the Kvaser CANLeaf analyzer in our Solutions packs. The Kvaser CAN analyzer can be plugged into our CAN network at the Analyzer node and connects to a PC via USB to allow you to monitor the network.

When testing your CAN system make sure that the switches on the EB048 CAN faults board are in the Normal position.

3.4 Setting up the Kvaser CANLeaf analyzer

The Kvaser CAN analyzer comes with set up instructions, documentation and software on the accompanying Kvaser CD. Please refer to the provided documentation for detail on installing and setting up the software.

3.4.1 Using the Kvaser CANLeaf analyzer

Connect the CANLeaf analyzer to the Kvaser analyzer node with the channel 1 D type connector.
Connect the analyzer to the PC with the USB cable connector.
The CAN analyzer program is called CANKing and should be in your Programs menu. Open CANKing.

The main two parts we are interested in right now are the Start and Pause buttons and the Message screen. The Start and pause buttons do as they say, they allow you to start pause and restart the analysis.

The Output Window displays all the messages on the CAN network.

The Message Id, data length, data items, time sent and other bits of information are noted. This can help you track down problems in your code due to missing data, wrong ID's etc., or to check that a node is actually getting the right information sent to it if it is not responding correctly. You can also insert custom messages onto the network for testing and debugging. The analyzer makes life much easier and should be used as a matter of course when programming.

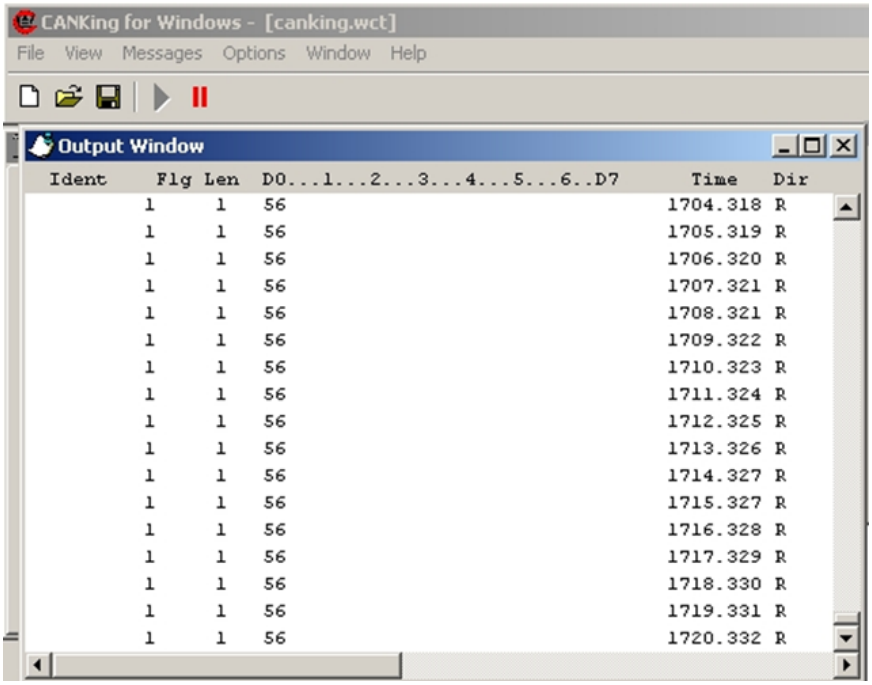


Figure 3.4 *The CANKing output window*

3.4.2 Analyzer network settings

In order to get the Analyzer working correctly on the network we need to set up the bus parameters. If you use the suggested setting for the CAN component these should match up already. If however you need to change them for any reason they are on the Bus Parameters tab of the CAN controller window.

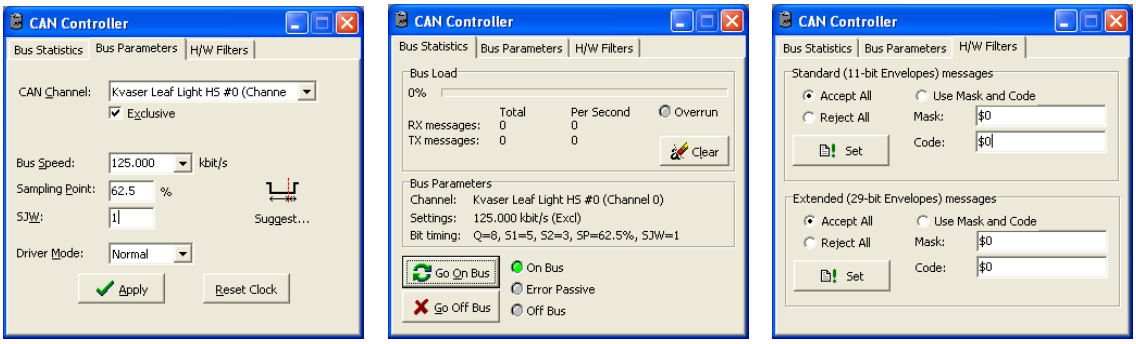


Figure 3.5 *CAN controller set-up dialogue*

Statistical details and current Bus status are contained on the Bus Statistics page; along with the On/Off Bus connection buttons (see Fig. 3.5). The standard settings used by CANKing work with the suggested settings option for the Matrix CAN board are:

CAN Channel: Select the USB CANLeaf options
Exclusive: On
Bus speed: 125 kbps
Sampling Point: 62.5%
SJW: 1
Driver mode: Normal

4.1 The physical layer

The physical layer used in the Matrix CAN system comes in the form of a *twisted wire pair*, ending in termination resistors. The resistors are added to the ends to help prevent signal loss or interference. The CAN board has a CANH (CAN High Line) and a CANL (CAN Low line) terminal socket which is used to connect the node to the network. As supplied a blue wire is used for CAN High, and a Yellow wire is used for CAN Low. This wire color code is not obligatory, you can use your own color code if you already one set up, but you must wire CANH to CANH and CANL to CANL when wiring up connections.

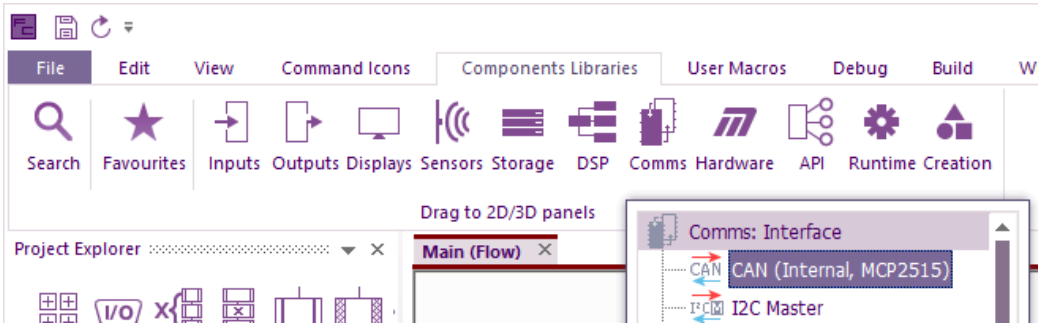
Termination resistors are set by means of a Jumper – J. Move the jumper to the END NODE box position to set the termination resistor for the nodes at the ends of the network.

The board uses both a CAN Controller (MCP2515) and a CAN Transceiver (MCP2551). The CAN controller uses the SPI™ bus to configure the CAN controller for transmitting and receiving CAN information. Information sent and received is stored is a series of buffers. Three transmit buffers and two receive buffers are used to store the data. Note that these buffers are part of our implementation of a physical layer for CAN and are not a part of the CAN specification itself.

4.2 The Flowcode CAN component

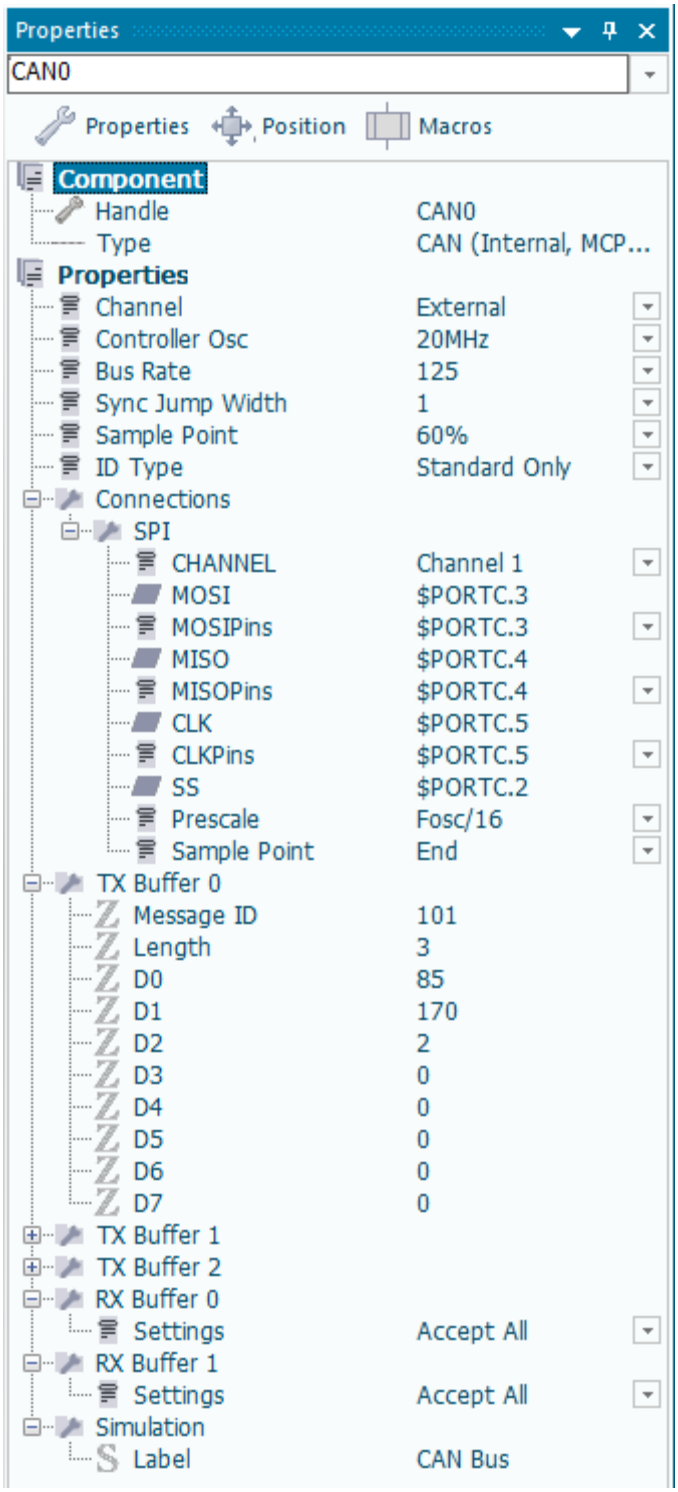
The Flowcode CAN component uses a series of properties and macros to provide CAN messaging in Flowcode. The properties are used to set up defaults both general, such as baud rate and sample point, and CAN message data, such as Message ID's and default data. The macros allow the user to initialize the CAN system and to send and receive data. Macros allow the user to edit the main parts of the CAN message, such as Message ID and data sent. Other parts of the CAN message not related to data transmission, such as ACK bits and bit stuffing are handled behind the scenes automatically.

The CAN component can be found in the Comms section of the Components Libraries Toolbar



CAN component icon

Details of the macros and properties of the Flowcode CAN component are contained in the CAN component help file. The settings shown in Figs. 4.1 to 4.4 and listed in sections 4.3 to 4.5 are used for all projects in this course unless specifically stated in the instructions for that task.



CAN Component Properties
The CAN component properties can be found on the 'Properties Panel' when the CAN component is selected / highlighted.

Properties
The *Properties* section allows you to set the CAN configuration settings.

Connections
The *Connections* section allows you to set the *SPI* settings for the CAN. For the PIC is Port C, for Arduino/AVR is Port B.

TX Buffer
The 3 *TX Buffer* sections allow you to set the default details for the three transmit buffers (*TX Buffer 0* to *TX Buffer 2*) used in the CAN component. Unless modified in the program by macros, these default settings will be the *Message ID* values and data (*D0* to *D7*) values sent

RX Buffer
The 2 *RX Buffer* sections allow you to set *Masks* and *Filters* to process and select Message IDs to be received (more on this later). As well as the ability to configure the *Settings* of each buffer individually.

4.3 Target microcontroller devices

This course was written using PIC and AVR microcontroller device. If you wish to use other microcontroller devices you will need to adapt the settings, programs and instructions to match the new device.

4.4 CAN component settings

The required settings are as follows:

Channel: External
Controller Osc: 20MHz
Bus rate: 125 (kbps)
Sample point: 60%
SJW: 1
ID Type: Standard Only

Channel: Channel 1
Chip Select: Port pin 2

Note: Ensure the ***Bus Rate*** is set to 125, as it is set to 500 by default.

4.5 Flowcode Configuration settings

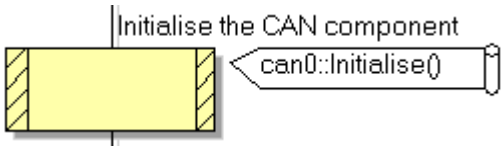
PIC Target: BL0011
Arduino Target: BL0055

4.6 PIC development board/E-blocks2 settings

Voltage Selector: 5V

4.7 CAN Initialise macro

The CAN component needs to be initialized before you can use it in your program.
To initialize the CAN component place the CAN “Initialise” macro in your program before any other CAN macros are used. Ideally it should be placed right at the start of the program.



5 Basic CAN signals

CAN is about both sending signals from nodes onto the network, and about receiving signals from the network and acting upon those signals if they are meant for that node. The two tasks are separate processes. A node can send a signal or it can receive a signal it does not need to do both. However it can do both, which adds to the flexibility of CAN systems.

A basic CAN system requires a Node that can send signals and a Node that can receive signals. There can be more nodes, either sending or receiving signals, but a minimum of one sending node and one receiving node are required for communication.

As mentioned earlier the CAN signals are sent and received using our implementation of the Physical layer – CAN specifies the message leaving the lower physical layer for us to implement. In our system we have TX Transmit Buffers and RX Receive Buffers that are used to store the data for sending, or for us to examine when it is received. Each Message is sent with a Message ID value which can be checked for by other nodes and acted upon if it matches a list of ID's to be accepted.

5.1 Implementing basic CAN signals in Flowcode

5.1.1 Initializing the CAN component

Whatever function a node has – sending, receiving or a mixture of both, it requires the CAN component to be initialized in order for the component to work.
Add an *Initialise* macro to your program, preferably at the start where it can be checked for quickly.
Basic CAN signals require two separate nodes to set up – a sending node and a receiving node.

5.1.2 Sending nodes

The CAN component's default settings for the three TX transmit buffers are all set to 0. This means that they need configuring in order to function. Once configured, the messages can be sent with the *SendBuffer* macro. The *Buffer* parameter refers to the TX0-TX2 buffer to be sent, and hence can be set from 0 to 2 respectively. Below is the TX Buffer properties which are used in the example files. For TX Buffer 0, the Message ID is set to 688 and the Length is set to 2, which determines how many bytes of data will be sent – 85, 170, 0, 0, 0. We will learn more about Message IDs and data later on. For now just remember that this data will be sent with this Message ID when TX Buffer 0 is sent.

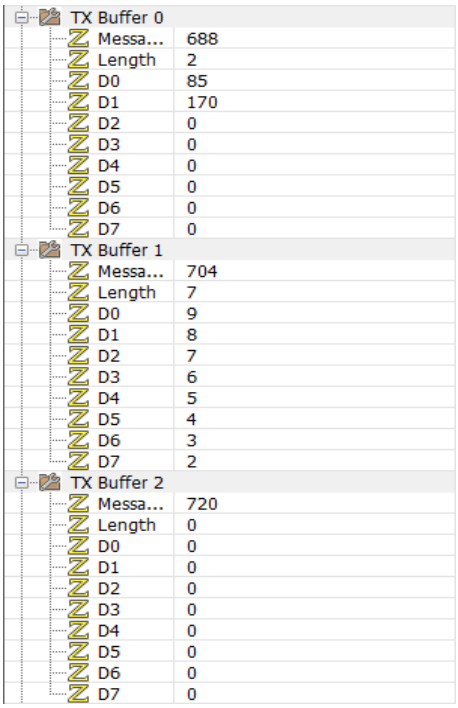


Figure 5.1 *Sending node settings*

5.1.3 Receiving nodes

The receiving node needs to be set up to accept incoming messages in order to function. By default the RX Receive settings are set to reject all messages. To alter this to go to the RX Buffer section on the 'Properties Panel' and set the RX Buffer 0 Buffer settings to 'Use Mask and Filter', this accepts a set of specific messages. Note for the settings specified in the example files set the Filter 0 property to '100'. When using the example settings, ensure that you change the RX Buffer 1 settings to 'Accept All'.

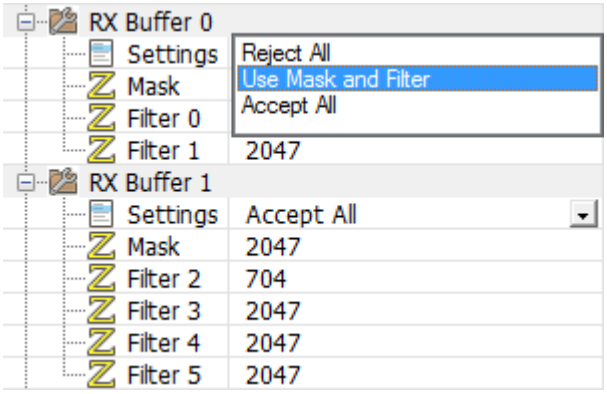


Figure 5.2 *Receiving node settings*

The node will now accept all transmissions on that particular buffer regardless of who sent them.

You can check for arriving messages with the *CheckRx* macro. The buffer parameter selects which receive buffer to check. In this case we need to check buffer 0 for RX Buffer 0. If a message has arrived *CheckRx* will return a non-zero value. If we have a non-zero return value the node can then respond to the message by performing whatever functions it has been programmed to do.

5.2 Using basic CAN signals

A simple system can be set up using basic CAN signals. However the system is restricted to only one activating signal, and only one response. More than one activating node can be present that can send an activating signal but as the receiving node accepts all signals it does not matter who sends the signal. In a similar way, any node set to receive all signals will react to the message sent. It does not matter who sent the signal, or who to, only that a signal was sent. This allows us to have more than one node respond to the same signal, but as they all accept all messages the response will be the same for any signals sent.

This method of communicating is very basic. All signals are reacted to in the same way and there is no way to prevent signals not intended for the receiving node being accepted as well, thus generating false events. Although this system can work for a small single purpose network, e.g. for a brake/brake light system, such a system would be unlikely to be used outside of a learning environment. The CAN system as described here is simply not robust enough for practical use. Some method must be used to make the nodes more selective of what messages they send or receive. This will be covered in the next section, dealing with Message ID's.

6 CAN Demonstrations

The following three demonstrations illustrate CAN in action, and some of the main uses of CAN. These demos can be used to give students a basic understanding of CAN systems prior to the CAN programming examples.

Also the demos can used to illustrate concepts such as message monitoring, start-up scans and diagnostic tests for students who may be working with CAN systems, but are not required to create or program CAN systems.

- DM01 (6.1) illustrates a start-up scan with basic system error checking. Checking that a node is present on the system is the first step in examining the problem.
- DM02 (6.2) is a basic message monitoring example. Message monitoring is a standard method of fault diagnosis. Monitoring the message traffic can help to identify which node or nodes have faults in them.

DM03 (6.3) is a node specific monitoring and test program. This demo illustrates the use of diagnostic tests to examine a particular node to diagnose the fault.

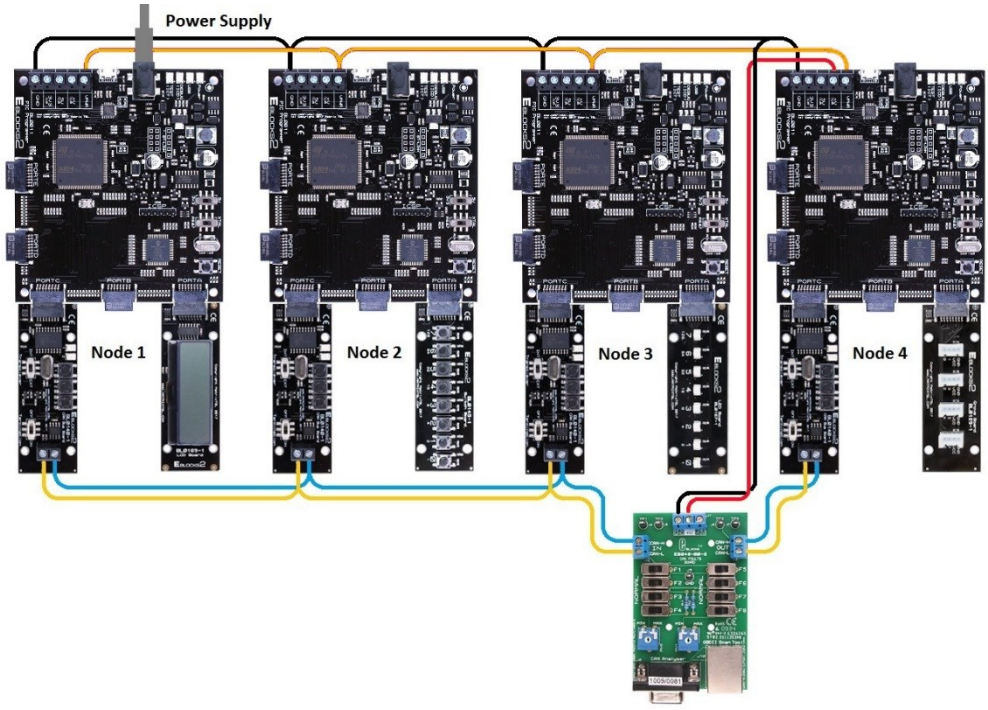


Figure 6.1 *CAN System – System overview*

CAN Node 1 – Main control panel
CAN Node 2 – Switch panel
CAN Node 3 – LED panel
CAN Node 4 – Sensor panel

Flowcode FCFX files are provided so that advanced users can see and work the code if required.
For simple demonstrations the Nodes can be pre-programmed to allow students to examine the nodes straight away.

Note:
For these Demos only the Message ID HI byte is used in order to simplify the mathematics involved.
It is assumed that all Message ID LO bytes are identical as so can be safely ignored for these demos.

6.1 DM01 – Start-up scan

6.1.1 Aim

To demonstrate an initial start-up scan that checks the system components are all connected and working, and how this can help in automotive diagnostics.

6.1.2 Resources:

This task uses 5 programs:
DM01_N1 – Node 1 program (Main control panel program)
DM01_N2 – Node 2 program (Brake switch)
DM01_N3 – Node 3 program (Brake lights)
DM01_N4 – Node 4 program (Fuel sensor)
DM01_NX – program for non-functioning node

Programs are supplied in FCFX file format so that users can view and modify the flowchart program.

(Note: For the purposes of this exercise the programs only contain code for the start-up scan procedure.)

6.1.3 Part 1: Running the scan

1. Load programs DM01_N1 – DM01_N4 into nodes 1-4 respectively.
2. Upon reset of the Node 1 control panel node the Scan program will commence.
3. The LCD will display “Starting scan”
4. Next Node N2 (Brake switches etc.) will be checked.
5. Once Node N2 has been detected “N2 – Brake Sw” and “Present” will be displayed.
6. Next Node N3 (Brake Lights) will be checked.
7. Next Node N3 (Fuel Sensor) will be checked.
8. Finally the LCD will display the overall scan result. In this case “All systems on” to show that all systems responded correctly.

6.1.4 What is happening?

The scan works on a simple message/response system.

The whole system is broken up into 4 separate units, called Nodes, which function independently. They are all however connected to the CAN bus, which allows them to listen for signals on the CAN bus and to send signals as well.

The Control panel node - N1 - sends a sequence of messages with a pre-set ID number, to the other nodes in the system.

The messages are not sent to a specific node. They are simply put on the CAN bus for all the nodes to listen for.

The other nodes on the system are listening for messages. They have been set up to look at the ID of the message and only respond to one specific ID message. When a node spots a message it can respond to it sends a message of its own i.e. it responds to the initial message.

The control node N1 node then waits for this response. If it does not get one in a set time it records the node as not present and moves on to the next item to check. If node N1 receives a message it then knows that that particular node is present on the system.

6.1.5 Part 2: Picking up errors.

1. Send the program DM01_NX to one or more of nodes 2-4 (e.g. Node 2).
2. Press reset on Node N1 to rerun the scan.
3. Note what happens when the scan reaches the node(s) with DM01_NX in them.
4. The program will pause for a bit whilst it is looking for the node. After a short while it will assume the lack of a response means that the node is not connected or not working and will report the node as “Not present”.
5. Once all nodes have been scanned for the LCD will display a “Warning” message and then list the Nodes that it did not find e.g. “N2”.
6. Next, send the DM01_NX program to nodes N2-N4.
7. Press reset to run the scan again.
8. Once it is complete it will display an “Error” message and “No systems found” to inform you of the problem.

6.1.6 What is happening?

The program DM01_NX effectively disables the node mimicking a non-functioning ECU which does not respond to the ‘scan’ from Node N1.

6.1.7 Limitations

The start-up scan simply checks that a node is present: it does not prove that all parts of the node circuit are working correctly. For example a fuel node with a broken sensor will be present, but not working. However just knowing that a node is or isn’t present can help solve some problems in automotive networks.

Note that we have made separate programs here to illustrate how the start up scan in a car can help check the functionality of the car before a journey starts. In practice each sensor would have a routine like this incorporated into its main program.

6.1.8 How does this help me?

If a system (node) is not attached or not working it can’t be used. A start-up scan is useful for this basic level of diagnostics. Are all the systems present and accounted for? Finding out that Node N2 is not responding to your scan means that you can then start checking Node N2 for faults.
Whilst we have only four simple nodes here imagine a more complex system with a hundred or more nodes. One simple start-up diagnostic program can save you hours checking with a multi-meter.

6.2 DM02 – CAN monitor

6.2.1 Aim

- To monitor the messages and data being sent on the CAN bus.

To display the Node and function, the Message ID and the Data.

6.2.2 Resources:

This task contains 5 programs:
DM02_N1 – Node 1 program (Main control panel program)
DM02_N2 – Node 2 program (Switches node)
DM02_N3 – Node 3 program (Lights node)
DM02_N4 – Node 4 program (sensor node)
DM02_NX – program for unknown messages

Programs are supplied in FCFX file format so that users can view and modify the code.

6.2.3 Part 1: Message monitoring

1. Load programs DM02_N1 – DM02_N4 into nodes 1-4 as appropriate.
2. Upon reset of the Node 1 control panel node the Monitoring program will commence.
3. When a message is received the ID is checked against a list of known ID's and the Node and function displayed on the top line of the LCD.

On the next line of the LCD the Message ID and the Data is displayed.

6.2.4 Signals sent

- The Fuel sensor on Node 4 automatically sends out fuel level signals every few seconds.
- The switches on Node 2 send messages when pressed: data value 255 when pressed and data value 0 when released.

Node 3 lights up the LEDs when appropriate, but does not send out any signals.

6.2.5 What is happening?

The control panel listens for any message and displays the Message ID and data value on the LCD display. The Message ID is also checked against a list of known Message ID's to get the Node and function information, which is also displayed. Messages which are not on the list of known nodes and functions are listed as "NX – Unknown".

Some messages are sent automatically, such as the Fuel level reading – this ECU has a program inside it which sends the messages at regular intervals. Others are sent as the response to an action – such as pressing or releasing the brake switch. Some nodes – e.g. Node 3, may never actually send messages: it only listens for messages and processes them. Note that this is an example of a fully functioning Node which does not create any messages on the CAN bus.

Note that when a switch is pressed a CAN message is generated, and when the switch is released a different message is generated. We could send a continuous signal until the switch was released, but that would swamp the CAN bus with messages. If a signal was sent just for pressing a switch we would never know when it was released. If the same data value was sent we could get confused as to when it was turned on and when it was turned off. So we send different data values to distinguish the 'switch on' and 'switch off' transitions. You could also use different messages with different Message IDs instead. The important thing here is being to be able to differentiate the different signal states.

6.2.6 Part 2: Unknown messages.

1. Send the program DM02_NX to one of the nodes 2-4 (e.g. Node 2).
2. When a message is received from this node it does not match any of the known nodes and functions.
3. In this case the node is reported as “NX – Unknown”

The Message ID and any data is shown on the second line of the LCD.

Unknown messages would be very rare in a new automotive system but can occur under certain circumstances. For example where a vehicle has been damaged, a new ECU may be fitted which has a later version of software than the original vehicle. This new CAN ECU may have slightly different messages, or may generate new messages. Because of this automotive technicians may be asked to download new software to some parts of the vehicle to cope with engineering changes that have occurred as the design has changed.

6.2.7 Limitations

The Message IDs are checked against a list of known nodes and their functions contained in Node N1. Each message only contains an ID and some data – a programmer has made some kind of look up table in Node N1 that relates this to the function of the node the message is from. A message does not tell us where it came from. It only contains the Message ID and the data. If a message arrives with a Message ID that is in the list it will be reported as being that node and function regardless of whether it is or not. This is a very important point to appreciate. For example if you programmed a temperature sensor ECU with the program for a fuel sensor, then it is possible that you could end up with a fuels sensor level indicator which is actually governed by the engine block temperature!

Due to LCD display size constraints we can only display the first item of Data from the message. However the CAN nodes can send up to eight items of data with each message.

6.2.8 How does this help me?

The CAN bus is a message system. It carries the messages produced by nodes along the system for other nodes to listen in to and react to when needed. The CAN monitor program allows us to see this information visually.

What these messages are and what data they carry is of prime importance to us. Wrong or missing data and messages are prime indicators of problems. If the brake node is not sending messages when pressed we know there is something wrong with the brake node. If it is sending data but the brake light is not coming on, then we know it is a problem with the brake light node. Just monitoring for brake messages can tell us where the problem lies: at the source of the messages, or at the destination.

By monitoring the messages sent and the data values, and what causes signals and what reacts we can start to diagnose problems, and narrow down what individual nodes need further diagnostic tests. If a unit is upgraded or modified we can also monitor its messages and data to ensure it is working correctly.

6.3 DM03 – Sensor Diagnostic program

6.3.1 Aim

- To monitor the messages and data being sent on the CAN bus.
- To display the Node and function, the Message ID and the Data.

6.3.2Resources:

	PIC BL0011			Arduino BL0055		
	Port A	Port B	Port C	A0-5	D0-7	D8-13
Node 1	BL0169	BL0145	BL0140	BL0169	BL0145	BL0140
Node 2	BL0145		BL0140	BL0145		BL0140
Node 3	BL0167		BL0140	BL0167		BL0140
Node 4	BL0129		BL0140	BL0129		BL0140

Node 4 BL0129 to be fitted with sensors: Light (socket 1, pins 0,1), Rotary (socket 2, pins 2,3) and Temperature (socket 3, pins 4,5)

Programs are supplied in both HEX file format for immediate use and FCFX file format so that users can view and modify the code.

6.3.3 Part 1: Sensor monitoring

1. Load programs DM03_N1 to DM03_N4 into nodes 1-4 respectively.
2. Upon reset of the Node 1 control panel node the Start-up scan program will commence. This is detailed in DM01
3. The Start-up scan is used to verify that the Sensor node is present.
4. Once the scan has been completed the program then monitors and displays the sensor values. You should be able to see a change in the value displayed by altering the light level, temperature, and also by altering the variable resistor on the sensor board which is used to mimic fuel level.
5. Temperature, Fuel level and Light levels are monitored, with the sensor values displayed on the LCD below the appropriate heading.

The data assumes an initial value of zero, and is updated every time a message arrives bearing data from the sensor node on the status of the sensors.

6.3.4 What is happening?

The control panel listens for sensor node messages and records the data received from them. The data in this case is a number between 0 and 255. In practice a separate program on the instrument console would be used to convert this data into a meaningful quantity for human beings; for example fuel remaining.

In this set of programs the data received is displayed to allow a technician to monitor the sensor values. Various problems will cause distinct readings that can be used to predict what faults have occurred.

6.3.5 Limitations

The sensor monitor program only shows the data values and does not provide diagnostic error messages for all situations. Given the small system here it is not a problem, but more complex systems may require better internal error checking procedures and diagnostic messages to be incorporated. However as you will see in the next few exercises, the CAN system can greatly assist in finding faults with nodes.

6.3.6 Part 2: Potential problems

The following set of Errors show various diagnostic tests in action.

6.3.7 Error 1: Node 4 has no power.

1. Set up the programs as in part 1
2. Unplug the power to the sensor node.
3. Run the diagnostic program
4. The start-up scan will report that N4, the Sensor node, was not present indicating a problem with that unit.

The monitoring values will all be zero and will not change.

This is the first test – checking that the node is actually there. If this test indicates the sensor node is not present then we know that the fault is affecting the whole node and the error is likely to be failure of the main ECU, loss of power or ground.

6.3.8 Error 2: The sensors have no power.

1. Set up the programs as in part 1
2. Remove power from the sensor board (not the whole node, just the sensor board) by unscrewing the power screw terminal and removing the red wire.
3. Run the diagnostic program
4. The start-up scan will report that N4, the Sensor node, is present.
5. The monitoring values will all be stuck at whatever default value they have and will not change indicating that the sensors are not updating.

Stimulating the sensors produces no change in the reading.

Here the operation of the sensor node, N4, is fine. However there is a fault with part of the sensor circuit that the sensor node cannot detect. We know the node is working because we are getting readings: because none of the sensor readings change when we stimulate the sensors, we know the fault must be with the sensor board as a whole.

6.3.9 Error 3: Breakdown mid journey

1. Set up the programs as in part 1
2. Run the diagnostic program
3. Monitor the sensor values to see that they change and update as the sensors are stimulated.
4. Unplug the power to the sensor node.
5. No more messages will be sent meaning that the sensor data values will not get updated.

The readings will appear to jam. Stimulating the sensors will not change the sensor values.

Here we are simulating a breakdown, where the power loss is after start-up so we don't get the "Not present" warning that would show us the problem then. This could be the result where there is an intermittent fault in the power line to a sensor ECU: the system checks out on start up, but fails part way through the journey.

This is a tricky situation as ideally we would like to know when a system failed and is not updating. There are a number of ways we could check for this.
We could for instance have a timeout after which the sensor readings go to "–".
We could also re-run the start-up scan for the entire system every few minutes in the journey to check all nodes are present.

7 Worked Example 1: Brake!!!!

For this first example we will just send a message and see if we can receive it on the other node, reacting in some way to show that it has arrived.

7.1 Objective

Create a CAN network that turns on the brake light when the brake pedal is pressed. The brake pedal is on switch 0 on Node 2, and the brake rear light is on LED 0 on Node 3.

7.2 Part 1: The basic programs

For this program we will use the default settings.

Note: You will need to configure the microcontroller development board for the correct microcontroller device.

7.2.1 The Send signal

1. On the Flowchart add a 'Call Macro' icon and set it to *Initialise*. This is an important macro that is needed for the CAN component to work. This macro is best added at or near the start for ease of reference.
2. Add a loop so that the program will run continuously.
3. Add an input icon and get the value of switch D0, the brake pedal, into a variable called *BRAKE*.
4. Add a decision icon set to *BRAKE > 0*.
5. On the *YES* branch add a further macro. Open the macro Properties Panel and select the *SendBuffer* macro. The macro takes one parameter – *Buffer*. We will discuss buffers in more detail later. For now set *Buffer* to '0'.
6. Add a short delay (Delay icon set to 100ms) just before the end of the loop.

Save the program as CAN_Example_01_Send.fcfx.

Fig.7.1 shows the Flowcode implementation. The program is now ready to compile and download to a CAN node. We now have a working CAN node. When we press switch D0 it will send a message on Buffer 0.

7.2.2 The receive signal

We have a message – now we need a node that can react to it!

1. Start a second program, once again with the appropriate microcontroller.
2. Add a CAN *Initialise* macro and a loop.
3. Inside the loop add the macro *ReadRx* and set the parameter, *Buffer*, to '0' to match that of the message we sent in the first program.
4. Create a MESSAGE variable and use this for the Return value.
5. When a message gets sent to buffer 0 ReadRx will return a non-zero value.
6. So if we follow the Read Rx macro with a decision icon we can then react to the message.
7. Add the decision icon and set it to *MESSAGE > 0*.
8. On the YES loop add an output icon so that we can react to the message.
9. We will set this to turn on LED 0 – the brake light.
10. On the NO loop add an output icon to turn off D0, so that it is not on permanently.
11. Add a short delay (Delay icon set to 100 ms) just before the end of the loop.
12. Save the program as CAN_Example_01_Receive.fcfx.

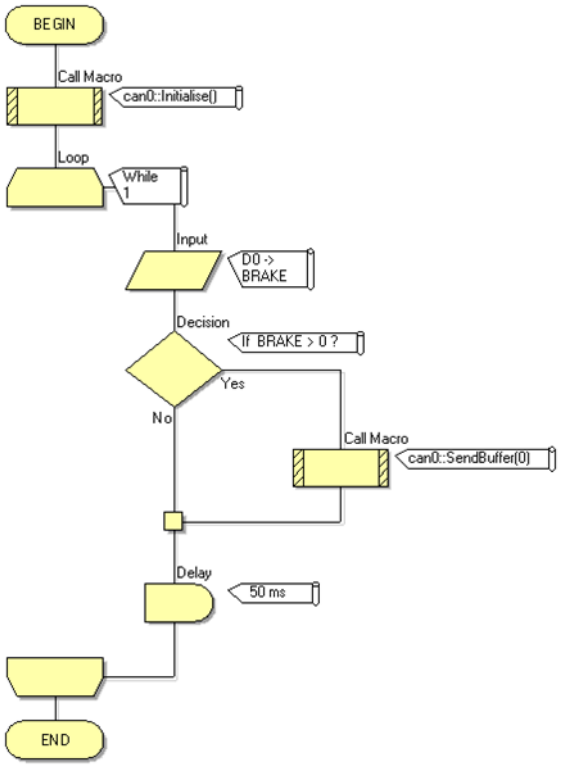


Figure 7.1 *Send signal flowchart*

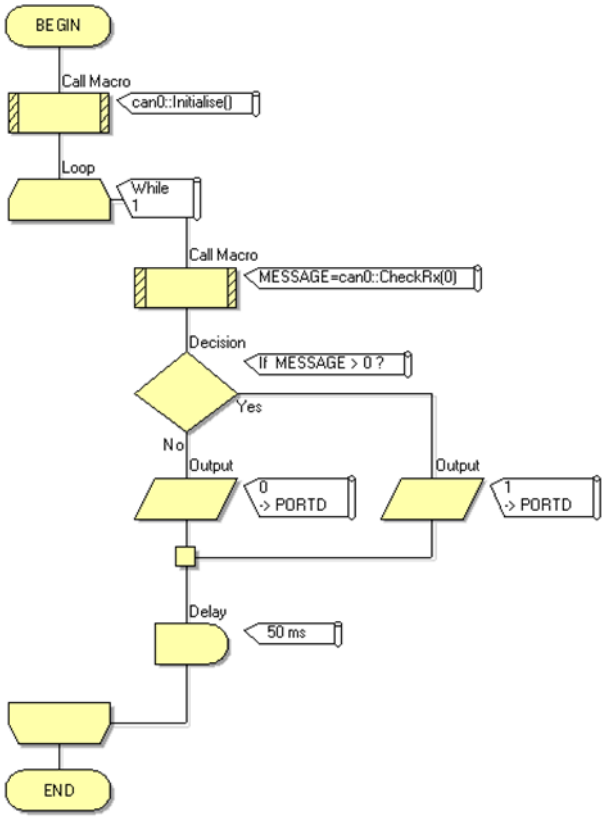


Figure 7.2 *Receive signal flowchart*

7 Worked Example 1: Brake!!!!

Having entered the flowcharts you are now ready to compile and download them.

1. Compile and download CAN_Example_01_Send.fcfx to the CAN node with the attached switches.
2. Compile and download CAN_Example_01_Receive.fcfx to the CAN node with the attached LED's.
3. Press switch 0 and, if all went well, LED 0 should light.

7.2.3 Testing the programs

Pressing switch 0 on Node 2, the switches node, should now light up LED 0 on Node 3, the LED node.

To view the network traffic

1. Connect the CANLeaf analyzer to analyzer node on the network.
2. Open CANKing and select the USB CANLeaf device.

7.3 Part 2: A second receive node

Send the receive program CAN_Example_01_Receive.fcfx to Node 1. Now when switch D0 is pressed the same LED will light on both Node 1 and Node 3.

Next modify the program to light up LED 1 instead. Save the program as CAN_Example_01_Receive_A.fcfx and send the program to Node 3. Now when switch 0 is pressed different LED's light up on Node 1 and Node 3.

7.4 Conclusions

This example demonstrates that sending and receiving are not only separate acts, but also that they are totally independent. Depending on the programs sent to different receiving nodes each node could respond to the signal in a completely different way.

7.5 Further work

- Consider what would happen if in Part 2 you loaded a program that would send a signal if switch 1 was pressed. How would the other Nodes be affected? Modify the send program to send the signal when a switch on Port A is pressed and send the program to Node 1 to see what happens.

- If you use CANKing to view the network traffic you will notice how many messages there are and how frequent they are sent. Consider if there is a better way of doing this that will cut down on the network traffic. Modify the send and receive programs accordingly to see if you can cut down on the network traffic generated.

8 Demonstration 1: Brake!!!

This example shows a simple signal-response system in action, the most basic CAN system possible.

8.1 Setup

	PIC BL0011			Arduino BL0055		
	Port A	Port B	Port C	A0-5	D0-7	D8-13
Node 1	BL0167		BL0140	BL0167		BL0140
Node 2	BL0145		BL0140	BL0145		BL0140
Node 3	BL0167		BL0140	BL0167		BL0140

- Switch 0 on Node 2 to mimic the brake pedal action.
- LED 0 on Node 3 to mimic the rear brake light action.

LED 0 on Node 1 to mimic the dashboard brake light signal action.

Open up file CAN_Example_01_Send.fcfx in Flowcode and download it to Node 2 (The Switches node).
Open up file CAN_Example_01_Recieve.fcfx in Flowcode and download it to Node 3 (The LED node).

8.2 Viewing the messages

If you open up CANKing and view the network traffic you will see a message being sent whenever the brake pedal is pressed.

8.3 Part 1 – The brake light

When the brake pedal is pressed (Switch 0 on Node 2) the Brake light (LED 0 on Node 3) lights up.
The signal generated by Node 2 (the switches) is picked up by Node 3 (the LED's) and, as all messages are accepted, the message is acted upon lighting the LED.

8.4 Part 2 – The dashboard display

Next download the CAN_Example_01_Recieve.fcfx program to Node 1.
Now when the brake pedal is pressed the LED's on both Node 1 and Node 3 will light up.
We have not altered the signal sent in any way, but both the receiving nodes receive and act upon the signal.

Next download the CAN_Example_01_Recieve_A.fcfx program to Node 3.
This time the same LED lights on Node 1, but a different LED (1) lights on Node 3.
The same signal is sent but acted upon differently by the two receiving nodes.

8.5 Conclusions

This example demonstrates that sending and receiving are not only separate acts, but also that they are totally independent. Depending on the programs sent to different receiving nodes each node could respond to the signal in a completely different way.

8.6 Further work

- Consider what would happen if in Part 2 you loaded a program that would send a signal if switch A1 was pressed. How would the other Nodes be affected?
- If you use CANKing to view the network traffic you will notice how many messages there are and how frequent they are sent. You may want to consider if there is a better way of doing this that will cut down on the network traffic?

9 Fault finding in CAN systems

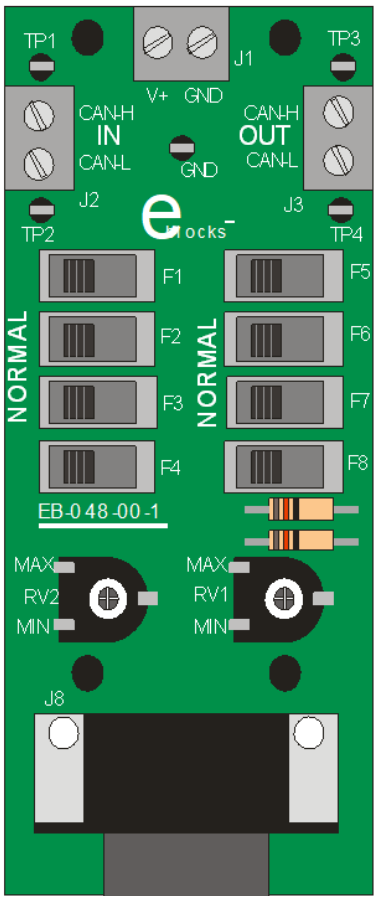
This exercise will allow you to understand how to find faults in CAN bus systems

9.1 Setup

Connect and power up the CAN solution. We will be using Node 1, the display node, to display the data. We will be using Node 4, the Sensor Node, to send the data.

Open the file CAN_EXAMPLE_04_RECEIVE.FCFX in Flowcode and download it to Node 1 (The Display node).
Open the file CAN_EXAMPLE_04_SEND.FCFX in Flowcode and download it to Node 4 (The Sensor node).

The CAN faults board is positioned between Nodes 3 and 4 in the system. Any fault in the CAN bus will result in the data not being transmitted.



Here is a graphic of the faults board. The screw terminals on the top are market IN and OUT. Of course CAN bus signals flow in both directions but we will use IN and OUT for convenience. Note that the CAN analyser socket is connected to the CAN IN lines. Make sure that all the switches are in the NORMAL position. In this position no faults are inserted. You will be inserting faults into the system between Nodes 3 and 4. In the example programs signals are being sent from Node 4 to node 1: this means that the Analyser is on the non-fault side of the CAN bus.

TP1 and TP2 are connected to the CAN-H and CAN-L signals ‘before’ a fault. TP3 and TP4 are connected to the CAN-H and CAN-L signals ‘after’ a fault.

9.2 Viewing the messages

If you open up CANKing and view the network traffic you will see messages being sent at regular intervals. If you vary the potentiometer on the Sensor board on Node 4 you will see on the analyzer readout that the data portion of the message relates to the potentiometer setting. In this case the potentiometer value is mimicking the fuel sensor on a petrol tank. Varying the potentiometer will result in a change in the display on the LCD on Node 1 by indicating how much fuel is in the tank.

- Connect a storage oscilloscope between GND and TP1. You should be able to see the CAN bus signal.
- Connect a storage oscilloscope between GND and TP2. How does it differ from the first measurement?

Make a drawing (approximate) of each signal or print it out. Record timing and voltage levels so that you can refer to these drawings later.

Remove the CAN-L and CAN-H wires from the 'IN' screw terminal connector. The LCD display should go blank. This is the fault condition: when there is no display then there is a fault on the system. Note that under partial open circuit fault conditions your CAN bus analyzer may still show data on the bus even though the LCD is blank: the reason for this is that the CAN interface on Node 1 has a different interface circuit to the CAN analyzer which is more sensitive.

9.3 Part 1 – F1

- Move switch F1 into the right hand position.
- What happens to the LCD?
- Is this a fault?
- Use the oscilloscope to view the signals on TP 1 to TP4. What can you see?
- Remove power from all nodes. Use a multimeter to measure the resistance between nodes and complete the table below.

What kind of fault does F1 represent?

	TP1	**TP2**	**TP3**	**TP4**
Resistance to ground				
Resistance to +V				
TP1	-			
TP2		-		
TP3			-	
TP4				-

9.4 Part 2 – F2, F3, F5, F6, F7

Move switch F1 back into the NORMAL position. Repeat the exercise in the section above individually for fault switches F2, F3, F5, F6, F7. When you do this make sure all switches are in the NORMAL position except one switch. You should now have a clear understanding of what fault each switch inserts into the system.

9.5 Partial open circuits

F4 and F8 are used to insert partial open circuits into each CAN bus line. Place all switches in the NORMAL position. Place switch F4 in the right hand fault position. Using the potentiometer vary the fault resistance until the system just stops working – the point at which the LCD on Node 1goes blank. Remove power. Complete the table above again and note the fault resistance. Complete this exercise again for switch F8. Does the resistance at which the CAN-H and CAN-L lines introduce a fault vary?

10 Intermediate CAN Networking

Here we will discuss and explain the core features of CAN networking, and provide examples and suggested projects for you to train with. By the end of this section you will be able to set up and run a multiple node CAN network that can respond to a variety of messages. There is a lot to learn here so you may need to come back to various parts of this section some times to review what you have learned.

11 The CAN component

Here we will look at Message ID's and how we can be selective about which messages to respond to. We will also look at sending and receiving data. We will examine the general settings, and how they affect the network.

To be able to work with Message ID's we need to understand the CAN component Properties Panels.
The various tabs on the CAN component properties detail general settings, and the default TX Transmit and RX Receive buffer settings.

11.1 General Settings

The general settings properties tab contains the main CAN network settings properties.

Bus rate Bus rate is the connection speed for the node. The node will expect all messages to arrive at that rate. Messages arriving at too fast a rate or too slow a rate may be sampled incorrectly leading to erroneous or jumbled messages. All nodes on a network need to be set to the same Bus rate otherwise communication problems will arise.

Sample Point Sample point is the point of the expected signal pulse at which the signal is measured to determine if it is a 1 or a 0. This is normally set to 50%-80% of the signal pulse period.

Synchronization Jump Width - SJW Synchronization Jump Width is used to help synchronize CAN nodes. As CAN does not use a clock it needs to synchronize itself with the transmitter nodes. SJW is a variable that helps set the maximum amount of timing leeway allowed for synchronization. This is used to help data transfer between data nodes on unusually long CAN cables and can be left as is for most node networks.
The default settings, of the CAN component are as follows: Channel = External, Bus Rate = 500, Sample Point = 60%, Sync Jump Width = 1, ID Type = Standard Only. In order to use the CAN analyzer shipped with some of our CAN systems, ensure that the settings are the same for both the CAN component and the CAN analyzer, as it is required for them to function correctly.

Note: Change the 'Bus Rate' from '500' to '125' to match the settings required for the exercises.

11.2 Transmit Buffers

Our implementation of the CAN physical layer has three transmit buffers – TX Buffer 0, TX Buffer 1 and TX Buffer 2. Each buffer contains a Message ID and a batch of Data. All three buffers work the same way. The buffers are used to store the CAN message until it is ready to send. The buffer can be modified and changed and is not fixed. The default values on the Properties Panels are used unless programmatic changes are made in which case the changed values are used.

Use the properties panel to set the Message ID's and data for each of the three buffers, this gives you the ability to send three predefined messages. Later on we will look at ways to modify these properties on the fly, but for now we will use the properties panel to set up the messages.

11.2.1 Message ID's

When a message is put onto the network, the nodes need to know whether to react to the message or not. In the basic example we simply reacted to the presence of a message regardless. However we can be selective. Each message sent onto the network has a Message ID number. The nodes can check this Message ID number to see if it matches with the list of ID's it should accept. If it matches the node can react to the message, if not the message can be disregarded.

If you look at the Properties Panel (see Fig. 10.1) and find the TX Buffer 0 section you will see a Message ID property. This is the Message ID value sent along with Buffer 0.

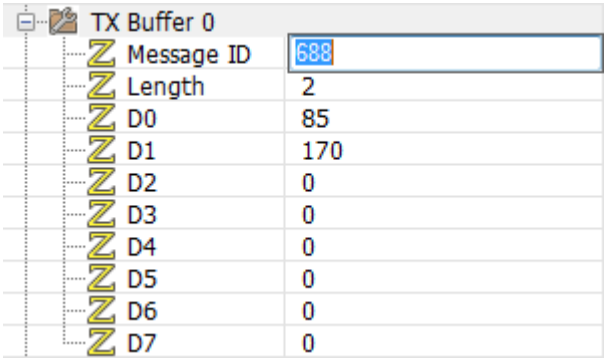


Figure 11.1 *Setting the Message ID in the Properties Panel*

TX Buffer 1 and TX Buffer 2 also have identical boxes for you to put their Message ID's in. This means that we can generate messages with three separate Message ID's from any particular node by using just the default buffer properties alone. However, we can have more than one node on a network, each of which could transmit signals with different Message ID's – or even the same Message ID's depending on the system. From this you can start to grasp the enormous amount of potential messages, each with their own Message ID that can be sent. Later we will look at ways to change the buffer Message ID programmatically, giving us even more flexibility than the three default values.

11.3 Receive Buffers

There are two receive buffers RX Buffer 0 and RX buffer 1 in our implementation of the CAN physical layer, which store received data. These can be checked to see if they have received any messages.

Receive buffers can be one of the most complex parts of our implementation of CAN to use and understand. The Filter properties are similar to Message ID's as the Buffers will react to them. The advanced mode will be dealt with later in the advanced CAN networking section.

There are 6 Filters in total split between two RX Buffers (0 and 1). RX Buffer has 2 filters available (Filter 0 to 1) and RX Buffer has 4 filters available (Filter 2 to 5).

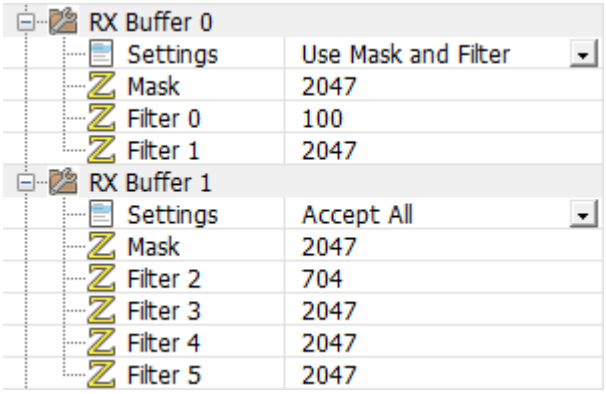


Figure 11.2 *RX Buffer Filters and properties*

You can set up the simple mode by clicking on the 'Simple settings' check box. You can then enter Message ID values into the boxes. Messages with these ID's will be accepted by the RX Buffer. Once a message has arrived we can check for it with the *CheckRX(Buffer)* macro. The buffer parameter is used to indicate which RX Buffer is checked (0 for RX Buffer 0, and 1 for RX Buffer 1). A non-zero return value indicates that a message has arrived.

12 Working with Message ID's

Message ID's are complex to get to grips with due to the mathematics involved, although careful selection of Message ID's can simplify the problem greatly. There are two parts to consider – how to read in and differentiate Message ID's and how to change pre-set ID's for outgoing messages.

12.1 Checking Message ID's

You can check for messages with *CheckRx(Buffer)*, which returns a non-zero value when a message is received. You can interrogate a message to discover what the Message ID is. However Message ID's can be up to a value of 2047 (hex 0x7FF), whilst Flowcode and the microcontroller only use values up to 255 (0xFF). The range of possible Message ID values is greater than the largest number the microcontroller can handle. To get around this the message is broken up into two bytes, a *hi* byte and a *lo* byte. These are accessed via the two macro functions *GetRxIDHi* and *GetRxIDLo*. Both functions take the parameter *Buffer* to select which RX Buffer to get the Message ID data from.

The problem is further complicated by the fact that the Message ID is an 11 bit number with the first 8 Most Significant Bits forming the hi byte, and the 3 Least Significant Bits forming the first three Most Significant Bits of the lo byte (see Fig. 11.1). This can make the mathematics for manually checking Message ID values quite complex. The simplest way is to compare the retrieved values with known values for hi and lo to see if they match.

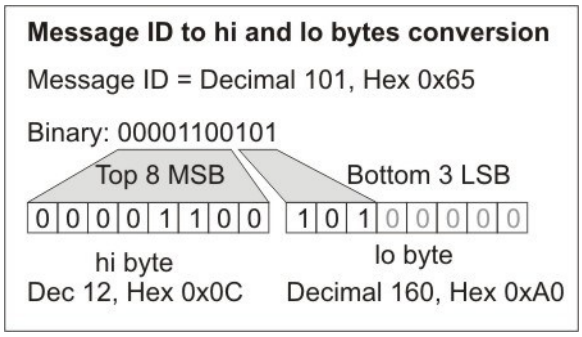


Figure 12.1 *Converting the hi and lo bytes of the Message ID*

12.1.1 Converting from 8-bit values to 11-bit

At some stage you may need to know how to get the full 11-bit value from the two 8 -bit High and Low bytes. The mathematics is as follows:

Message ID = (hi x 0x08) + (lo / 0x20)

12.2 Manual Message ID's – a recommendation

Because of the added complexity of working with the *hi* and *lo* Message ID values we suggest that for educational use, especially when just beginning to work with CAN, that you use a system where one of the bytes (probably the lo byte) remains the same, whilst the other byte (the hi byte) changes. This simplifies the mathematics considerably whilst still allowing you access to 256 different Messages ID's.

Note: Because we are creating both sending and receiving nodes we can pick Message ID's that make life easy for us. However, if you are adding a node to an existing system you may need to work with Message ID's that have already been set up. This may well mean that both hi and lo bytes need checking. You may also need to test the network to ensure that your Message ID's do not affect other nodes inadvertently.

13 Exercise 2: Rear Light cluster

13.1 Part A: Sending

13.1.1 Objective

Set up serious of switches to activate a brake light, an indicator light, and a rear light.

13.1.2 Instructions

The three lights have been assigned the following ID numbers:

Brake	= ID 8
Rear light	= ID 16
Indicators	= ID 32

The activation switches are as follows:

Brake	= Switch 0 - Brake
Rear light	= Switch 1 – Rear Light
Indicators	= Switch 2 – Left indicator

Node 2 (Switches Node) will be used to send the signals.

13.2 Part B: Receiving

13.2.1 Objective

Set up a basic car rear light cluster display containing a brake light, an indicator light, and a rear light.

13.2.2 Instructions

The three lights have been assigned the following ID numbers:

Brake	= ID 8
Rear light	= ID 16
Indicators	= ID 32

The Display lights are as follows:

Brake	= LED 0 - Brake
Rear light	= LED 1 – Rear Light
Indicators	= LED 2 – Left indicator

Node 3 (LED’s Node) will be used to display the signals.

The indicators need to be made to flash if possible at about 1 second on, 1 second off.

13.3 Further work

We have set the program up as a LEFT rear light cluster. Consider what changes we would need to be able to make in order to create a RIGHT rear light cluster.

Could we set up a front light cluster? If so what messages would be the same? What extra messages would we need?

Ideally we would want both a left and a right rear cluster. What changes would we need to the CAN system in order to enable us to set this up?

14 Notes for Exercise 2

14.1 Part A: Sending

We have three signals to send, and just by coincidence we have three buffers with which to send signals. If we set a light up for each buffer then we can send all three signals using the three buffers.

The basic program is the same as the brake light one in example 1, but with three check switch/send signal sections. Set each section up to use a different buffer, and set each buffer up with a different Message ID. These are what we will be dealing with next.

Open up the Properties Panel and set the Message ID for the three TX Buffers as follows:

Buffer	Message ID	Function
TX Buffer 0	8	Brake
TX Buffer 1	16	Rear Lights
TX Buffer 2	32	Indicators

14.2 Part B: Receiving

In Part A we set up a node to send three possible messages with the Message ID's 8, 16 and 32. Now we need to set up a node that can accept these three Message ID's.
If we open up the properties and look at RX Buffer 0 we will see that we can have up to 2 Message ID's to accept for that buffer. If we check Rx Buffer we will see that RX Buffer 1 has 4 message ID's we can set up. We have three possible incoming messages so we use RX Buffer 1 to receive them all.

Set the first three ID boxes on RX Buffer 1 to Message ID 8, Message ID 16 and Message ID 32. Deselect the other boxes so that they don't accept any other Message ID's.
Now we will get a response on RX Buffer 1 whenever one of those 3 Message ID's is sent.

We can use a simple *CheckRx* macro to see if a message has arrived. However there are three messages that could trigger RX Buffer 1, so we need to distinguish between them.
At this point you would probably need to get out the pen and paper and work out what the hi and lo byte values are for the two Message ID's. Fortunately we have done this for you.

Message ID	Hi byte	Lo byte
8	1	0
16	2	0
32	4	0

Now you can see why we picked such a strange Message ID sequence. The lo byte is the same for each, so we can simply test the hi byte to find out which message it is.
A more sophisticated approach would be to test the lo byte first to make sure it is 0, as expected, and then test the hi byte to see which message it is.

If we had picked values such as 101, 149, 150 the values would have been:

Message ID	Hi byte	Lo byte
101	12	160
149	18	160
150	18	192

And we would need to check both hi and lo bytes for the values involved.

Notes for Exercise 2

In our send program we set up the Message ID's 8, 16 and 32. Now we can check to see which message was sent and deal with it accordingly.

Message ID	Hi byte value	Message	Output
8	1	Brakes	D0
16	2	Lights	D1
32	4	Indicators	D2

So if we receive Message ID 16 for instance we turn on output D1.

The code fragment in Fig. 13.1 shows how we can check for a message and then retrieve the message ID identifier values (in this case just the Message ID Hi byte is needed) and check that to see which message has been sent.

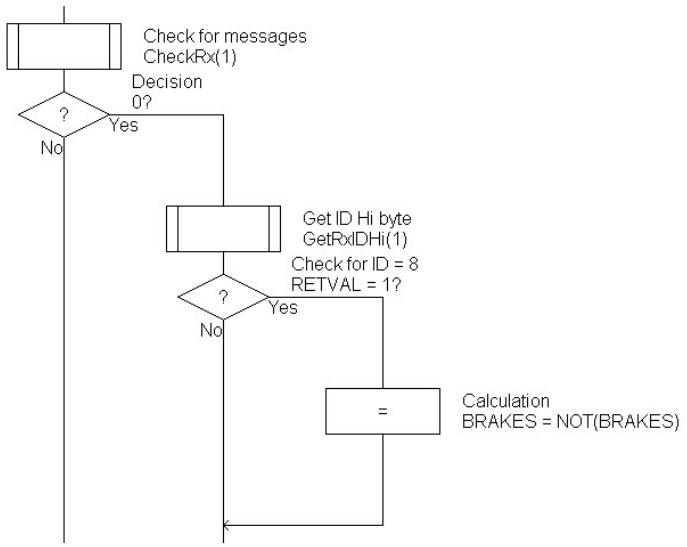


Figure 14.1 *Code fragment showing how to retrieve the message ID*

14.2.1 Which buffer to use?

We need to accept three Message ID values, but there are only 2 Message ID slots on RX Buffer 0. However RX Buffer 1 has 4 slots for Message ID values. In the example above we have set up all three messages on RX Buffer 1 and just used that buffer. However we could have used a combination of both buffers if we wished.

14.3 Indicators

Having flashing indicators is an optional extra as it does not concern CAN networking, but is a useful exercise and improves the visual appeal of the finished display.

14.4 Conclusion

Now we have a working system. It can send different signal and can differentiate between those signals. This enables us to build quite complex systems that are capable of sending many different messages from many different nodes, and has the capability to be selective and to react to some all or even none of the potential messages on the system.

14.5 MAJOR ERROR!!! – Is the Brake on?

There is a problem with the system. We have a brake signal that we are using to turn the brake light on or off. But what would happen if the system missed a signal, or started with the brake on when it should be off? The signal would become reversed. The brake light would be lit when the brake is off not on. Not a good situation.

We are stuck in this situation because we can only send three signals and all three are needed for different jobs. Also the signal sent for the brake does not tell us if the brake is on or off, simply that a Brake signal was sent. What we would like to do is either send separate Brake On and Brake Off signals, which would require more Message ID's than we have at the moment, or the ability to send data with the message to say whether the brake is on or off. And these are exactly the problems we will be dealing with next.

Further work

Further work includes some simple practical questions, and one theoretical one.

To make a right hand light cluster would just involve a different Message ID and activation switch to replace the left hand indicator. In a similar way the front light cluster would need to ignore the brake light message as there is no brake light there. But it would need a Dip signal sent instead.

Setting up a left and right system (or a full front and rear system if this train of thought is taken to its conclusion) is theoretical, as it requires macros not yet introduced. To set up a left and right system though would require either a fourth Message ID, or some kind of data to say which indicator to use.

15 Demonstration 2: Rear light cluster

This example shows a CAN message system in action.

15.1 Setup

	PIC BL0011			Arduino BL0055		
	Port A	Port B	Port C	A0-5	D0-7	D8-13
Node 1	BL0145		BL0140	BL0145		BL0140
Node 2	BL0145		BL0140	BL0145		BL0140
Node 3	BL0167		BL0140	BL0167		BL0140

Connect and power up the CAN solution. We will be using switches 0-2 on Node 2 to mimic activation action signals. We will be using LED's 0-2 on Node 3 to mimic the rear light cluster.

Open up file CAN_EXAMPLE_02_SEND.FCFX in Flowcode and download it to Node 2 (The Switches node).
Open up file CAN_EXAMPLE_02_RECEIVE.FCFX in Flowcode and download it to Node 3 (The LED node).

15.2 Viewing the messages

If you open up CANKing and view the network traffic you will see a message being sent whenever one of the switches is pressed.

15.3 The light cluster

When the brake pedal is pressed (Switch 0 on Node 2) the Brake light (LED 0 on Node 3) lights up.
The signal generated by Node 2 (the switches) is picked up by Node 3 (the LED's) and, as all messages are accepted, the message is acted upon lighting the LED.
In a similar way the Lights switch (Switch 1) activates the rear light (LED 1), and the Left indicator switch (2) activates the left indicator (LED 2).

15.4 The messages

The three lights have been assigned the following ID numbers:

Brake = ID 8
Rear light = ID 16
Indicators = ID 32

Watch in CANKing as messages are sent, and how the Message ID tells you which light will be activated.

15.5 Other network traffic

Open up file CAN_EXAMPLE_02_RANDOM_SEND.FCFX in Flowcode and download it to Node 1 (The Display node).
The switches on the Display node have been set to send various messages. Press on them to see what happens. If anything significant happens check with CANKing to see if you can work out why.

15.6 Conclusions

This demonstration shows Message ID's in action. Events can be linked to specific Message ID's so that they can be on a network with lots of traffic and will only respond to the correct signal, not random traffic as in demonstration 1.

16 Notes for Demonstration 2

The first demonstration files CAN_EXAMPLE_02_SEND.FCFX and CAN_EXAMPLE_02_RECEIVE.FCFX are based on those created for Exercise 2.

The final file CAN_EXAMPLE_02_RANDOM_SEND.FCFX that is sent to the Display node sends out various signals. Of note is the signal for switch D4 that has the message ID 32. This is the same Message ID as the Indicator signal. As it has the same Message ID it will trigger the Indicators on the receiving node.
This illustrates a potential problem on CAN networks – that of conflicting Message ID's. Where more than one Node can generate a message with the same Message ID thus inadvertently triggering the receiving Node. This can be used at the end of Exercise 2 to demonstrate the same potential problem.

17 Changing Message ID's

Up to now we have used the Message ID's as set in the Properties Panels. However, you can change the ID using the *SetTxID* macro. This opens up a whole new realm of possibilities. By being able to change message ID's on the fly you can react to inputs by sending different messages depending on the data received. For instance the message that operates a flashing warning light could be changed to one that also produces a warning noise after a certain point.

However there is a problem: Message ID's are 11-bits in length, but microcontrollers are only able to handle 8-bit numbers. So the Message ID value needs to be divided into two separate 8-bit bytes, a hi and a lo byte, as shown in Fig. 16.1.

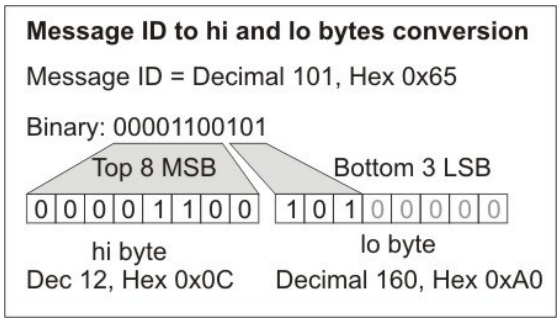


Figure 17.1 *Converting the hi and lo bytes of the Message ID*

17.1.1 The SetTxID macro

The SetTxID macro takes three parameters: buffer, hi and lo. Buffer is the TX Buffer number, 0-2.
The other two parameters, hi and lo, set the Message ID value.

The problem here is that whilst the Message ID value can be up to 2047 (hex 0x7FF) Flowcode, and the microcontroller can only handle numbers up to 255 (hex 0xFF). So we need to split the Message ID value into two separate bytes. The Message ID is 11 bits in length, and we need to convert this into two 8 bit bytes.

17.1.2 Converting from 11-bit to 8-bit values

The two bytes are laid out as follows:
The 8 Most Significant Bits (the first 8) of the ID are put into the hi byte.
The 3 Least Significant Bits (the final 3 bits) are put into the lo byte - but in the 3 Most Significant Bit positions (i.e. the first 3)

The mathematics for converting from the full Message ID to the two hi and lo bytes is:

hi = (Message_ID AND 0x7F8) / 8
lo = (Message_ID AND 0x007) * 0x20

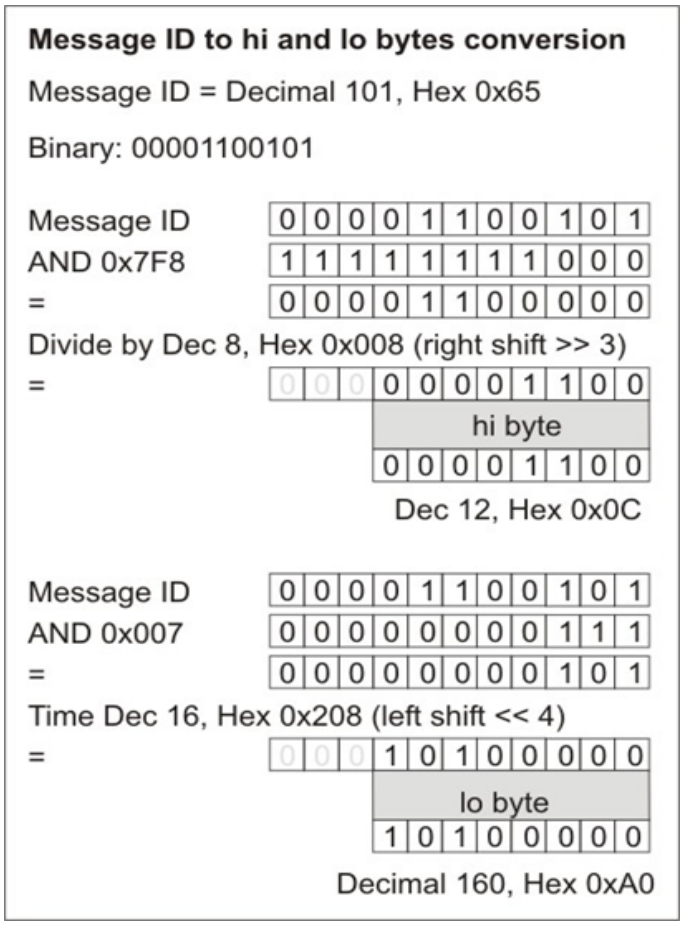


Figure 17.2 *Converting the hi and lo bytes of the Message ID*

The simplest way is to set the hi and lo values to pre-determined values. When choosing Message ID values you will need to ensure that you do not use any of the same Message ID's used by other transmitting node, otherwise you risk chaos on the network. You can aid the designer of the receiving nodes as well by selecting Message ID values that the receiving nodes can identify and differentiate easily.

18 Exercise 3: Rear light system

Exercise 3 extends Exercise 2 to include left and right indicators. The solution to Exercise 2 can be used as a starting point for Exercise 3.

18.1 Part A: Sending

18.1.1 Objective

Set up series of switches to activate a brake light, a left and a right indicator light, and a rear light.

18.1.2 Instructions

The three lights have been assigned the following ID numbers:

Brake = ID 8
Rear light = ID 16
Left indicator = ID 32
Right indicator = ID 64

The activation switches are as follows:

Brake = Switch 0 - Brake
Rear light = Switch 1 – Rear Light
Left indicator = Switch 2 – Left indicator
Right indicator = Switch 3 – Right indicator

Node 2 (Switches Node) will be used to send the signals.

18.2 Part B: Receiving

18.2.1 Objective

Set up a left and a right car rear light cluster display containing a brake light, an indicator light, and a rear light. Note that this will require two separate nodes with separate but similar programs.
Create one node first and use the program as the base for the program for the second node.

18.2.2 Instructions

The three lights have been assigned the following ID numbers:

Brake = ID 8
Rear light = ID 16
Left indicator = ID 32
Right indicator = ID 64

The Display lights are as follows:

Brake = LED 0 – Brake
Rear light = LED 1 – Rear Light
Left indicator = LED 2 – Left indicator
Right indicator = LED 3 – Right indicator

Node 1 (Display Node) will be used to display the Left hand cluster signals.

Brake = LED 0 – Brake
Rear light = LED 1 – Rear Light
Left indicator = LED 2 – Left indicator

Node 3 (LED's Node) will be used to display the Right hand cluster signals.

Brake = LED 0 – Brake
Rear light = LED 1 – Rear Light
Right indicator = LED 3 – Right indicator

The indicators need to be made to flash if possible at about 1 second on, 1 second off.

18.3 Further work

We now have a working rear light set. What changes are needed to create a front light set?

Can a full front and rear light system be created? (Node 1 can be used for the front light cluster).

We can use the same signals to set up a dashboard display. The only difference to a cluster is that the dashboard has signal lights for both indicators.

19 Notes for Exercise 3

19.1 The programs

Apart from having four messages the problem is the same as in Exercise 2. We need a fourth message so we need to modify the Message ID for at least one of the TX Buffers. However we can leave the other two as defaults. The best solution would be to keep the brake and Light messages as is, and modify the Indicator Message ID according to which one it is – left or right.

We can use the *SetTxID* macro to set the Message ID's on the fly. For this macro though we need the hi and lo bytes to be sent as parameters. The hi and lo bytes needed are listed below for convenience. Note how we have maintained the same system as before with only the hi byte changing.

Message ID	Hi byte	Lo byte
8	1	0
16	2	0
32	4	0
64	8	0

Once again though it is worth reiterating that we may not always have the luxury of such a neat set of Message ID's when working with other networks.

The two receive nodes are simple to make. The left hand indicator is exactly the same as the one in Exercise 2, and the right hand one only needs changing to accept the right hand indictor Message ID.

19.2 Conclusion

This may seem a small exercise, but it is of fundamental importance. Now we are free of the default three TX buffer/Message ID limitation. We can alter Message ID's as we see fit and so can create up to 65536 potential Message ID's. Our only worries now are that we ensure that our Message ID's don't clash with other Message ID's on the network, and that our receiving Nodes are set to receive them. But that is just down to good planning coding and documentation.

20 Demonstration 3: Rear light cluster

This example shows a CAN message system in action.

20.1 Setup

	PIC BL0011			Arduino BL0055		
	Port A	Port B	Port C	A0-5	D0-7	D8-13
Node 1	BL0167		BL0140	BL0167		BL0140
Node 2	BL0145		BL0140	BL0145		BL0140
Node 3	BL0167		BL0140	BL0167		BL0140

Connect and power up the CAN solution.
We will be using switches 0-2 on Node 2 to mimic activation action signals.
We will be using LED's 0-2 on Node 3 to mimic the rear light cluster.

Open up file CAN_EXAMPLE_03_SEND.FCFX in Flowcode and download it to Node 2 (The Switches node).
Open up file CAN_EXAMPLE_03 _LEFT_INDICATOR.FCFX in Flowcode and download it to Node 1 (The Display node).
Open up file CAN_EXAMPLE_03 _RIGHT_INDICATOR.FCFX in Flowcode and download it to Node 3 (The LED node).

20.2 Viewing the messages

If you open up CANKing and view the network traffic you will see a message being sent whenever one of the switches is pressed.

20.3 The light cluster

When the brake pedal is pressed (Switch 0 on Node 2) the Brake lights (LED 0 on both Node 1 and Node 3) light up. The signal generated by Node 2 (the switches) is picked up by both Node 1 and Node 3 (the LED's) and, as both nodes are set to accept all messages, the message is acted upon lighting the LEDs.

In a similar way pressing the Lights switch (Switch 1) activates the rear lights (LED 1 on Node 1 and Node 3). Note that whilst development boards often use push to make switches, real life applications would be likely to use toggle switches for items such as light switches.

The Left indicator switch (2) activates the left indicator (LED 2 on Node 1) and the Right indicator switch (3) activates the right indicator (LED 3 on Node 3). However, unlike the Brake and Light signals these two indicator signals are only accepted by specific nodes. So the left indicator signal is only accepted and acted upon by Node 1, the Left light cluster. Similarly the right hand cluster accepts the right hand indicator signal whereas the left hand cluster does not.

20.4 The messages

The four lights have been assigned the following ID numbers:

Brake = ID 8
Rear light = ID 16
Left indicators = ID 32
Right indicators = ID 64

Demonstration 3: Rear light cluster

Watch in CANKing as messages are sent, and how the Message ID tells you which light will be activated.

20.5 Conclusions

This demonstration shows Message ID's in action. Events can be linked to specific Message ID's so that they can be on a network with lots of traffic and will only respond to the correct signal, not random traffic as in the first demonstration.

21 Notes for Demonstration 3

This example is the same as Demonstration 2, but with both rear light clusters allowing the students to see the different indicators light up.

You can use this demonstration as an alternative to demonstration 2.

Sending the CAN_EXAMPLE_02_RANDOM_SEND.FCFX file from Demonstration 2 to Node 1 will allow you to show how random messages with the same Message ID's can cause problems for receiving nodes.

22 Message Data

22.1 Default Data properties

On the Properties Panels there is a set of properties for adding data to a message.
The properties are a Data Length property and up to eight bytes of data. The data length property sets how many bytes of data the message contains, and can be set from 0 to 8. When set to 0, no data will be sent. When set to 1-8 that amount of data bytes will be sent. Depending on the value set some or all of the data boxes may be grayed out. These grayed out boxes will not be used in the message. You can edit the value in the data boxes to set default data values that will be passed with that buffer. All three transmit buffers work the same way.

22.2 Changing Message Data

There is a macro that you can use to modify the data in your program.
The *SetTxData* macro takes the following parameters: *Buffer, Count, d0, d1, d2, d3, d4, d5, d6, d7*.
Buffer refers to the TX Buffer to be modified (0, 1 or 2).
Count sets how many bytes of data to use (0 – 8, with 0 being no data).
d0 – d7 are the individual bytes of data.

Note that values for all 8 bytes must be added, as they are required by the macro. Simply add a value for any unused bytes (traditionally '0' is used in programming for values that need to be supplied but are not acted upon). This needs to be done even if *Count* is set to 0, meaning no actual data is sent.

Example 1

SetTxData(0, 4, 255, 128, 32, 56, 0, 0, 0, 0) will setup TX Buffer 0 to send the 4 bytes of data 255, 128, 32, 56.

Example 2

SetTxData(1, 0, 0, 0, 0, 0, 0, 0, 0, 0) will setup TX Buffer 1 to send 0 bytes of data.

22.3 Keeping track of data

When a program uses *SetTxData* to modify the data it is up to the programmer to keep track of what the data now is. It is also the job of the programmer to ensure that the correct new data is passed to the buffer. The Properties Panel defaults are what is in the buffer when the program starts. If you modify the data it will stay modified until you modify it again.

22.4 Sending data

Whenever a buffer is sent, the Data associated with that buffer is sent automatically. No further action needs to be taken. If *SetTxData* has not been used to modify the default data then that default data specified on the Properties Panel for that buffer will be sent. If *SetTxData* has been used to modify the data then the current data will be sent. Note that the Properties Panel is not altered by *SetTxData*.

When a message is sent the data length is passed with it, along with the corresponding bytes of data. If data length is 5 then five bytes of data will be sent. If it is 3 then only three bytes of data will be sent. This is important as trying to read five bytes of data if only three were sent will cause problems.

22.5 Receiving Message Data

Receiving message data is done in two parts. Firstly you need to find out how much data has been sent, and secondly you need to extract the individual bytes of data.

The macro *GetRxDataCount* is used, along with the parameter buffer, to get the data length for the message in the relevant RX Buffer. Once you have the data length you can see if the message includes any data, and if so how many bytes.
You can then use this information to safely extract the data using the *GetRxData* macro.
GetRxData takes the parameters *Buffer* (the RX Buffer to use) and *Index* (the item to retrieve).

The index is numbered 0-7, in the same way the TX Buffer data items are named D0 to D7 on the Properties Panels. Index 0 is the first item, index 1 the second etc. Starting at 0 rather than 1 is a common feature in programming, and one to be aware of if you are getting erroneous data returns.

22.6 Data order considerations

Care needs to be taken when working with data as changing the order in which the data is stored will require corresponding changes to how the data is retrieved. Given the fact the nodes are independent of each other it is best to decide on a strategy for the data at an early stage of system design. Should more data be required it is often easier to add the new data on as extra items rather than change the order involved, as this will have less impact on any other nodes.

It may also be easier to use a pre-existing data structure and simply read in and ignore items that are not required rather than to reprogram several nodes just to rearrange the data order.

22.6.1 Example: Setting up the data node

We are given the task to send two bytes of information on TX Buffer 0, Message ID 105, when switch A0 is pressed. The two bytes of data have default values but the values can be updated programmatically.

To set up the default values we first open up the Properties Panel and go to the TX Buffer 0 tab and select a data length of 2. Note that boxes D0 and D1 are active, but that the others are grayed out. We would then enter the default data values which would be automatically send when the buffer is sent e.g. 145 in D0, and 12 in D1. Once we have set up the default values they will be sent whenever the buffer is sent unless they are changed programmatically.

We can change the value in the program by using the *SetTxData* macro e.g. *SetTxData (0, 2, MyVar0, MyVar1, 0, 0, 0, 0, 0, 0)* would change the two bytes of information to the values of MyVar0 and MyVar1.

22.6.2 Example: Setting up the receive node

If we wished to receive data from a message, such as two items of data as sent by the example above, we can *GetRxData* to retrieve the data items.

Set up a basic receive node that polls RX Buffer 0 for the Message ID 105.
Once a message is received we can query it.

Remember that the *GetRxData* index parameter is 0-7 not 1-8 matching the D0-D7 data items.
Add two *GetRxData* macros to the program. Set the first one to retrieve RX Buffer 0 data item index 0, *GetRxData(0,0)*, and put the value into DATA_0 (or some other suitably named variable).
Set the second one to retrieve data item index 1 and out this into DATA_1, *GetRxData(0,1)*.
We than have the data in and can then check it, display it or modify it as we wish.

22.6.3 Example: Variable amounts of data

We knew that there would only be two items of data in the program above as we created both nodes, but if we were not sure we could check how much data had arrived with the *GetDataCount* macro.

Once a message has arrived we would use *GetRxDataCount* to check how many items of data have arrived. We can then use this information to go through and read in the items of data.
Once we have the data read in using *GetRxData* we can then work with the data as needed for the program task.

23 Example 4: Fuel gauge and warning light

Set up a basic fuel gauge with a warning light than comes on when 10% or less of the petrol is remaining.

23.1 Part A: Sending

23.1.1 Objective

Set up a fuel level sensor that passes the fuel level as a 0-255 value.
In addition set up a “Fuel low” warning sensor that activates at a preset fuel level.

23.1.2 Instructions

Set up a basic CAN send program with the following default properties:
TX Buffer 0 – Message ID = 160, Data Length = 1, D0 = 0.
TX Buffer 1 – Message ID = 176, Data Length = 0.

TX Buffer 0 will carry the Fuel value in D0, and TX Buffer 1 will be used for the warning signal.

Node 4 (Sensor Node) will be used to send the signals.
Note: The variable potentiometer can be used to represent the fuel level.

23.2 Part B: Receiving

23.2.1 Objective

Set up display panel that shows the fuel level. Also set up a warning indicator that flashes when the fuel level becomes low.

23.2.2 Instructions

The LCD display can be used to display the fuel amount. This can be either raw data 0-255 or in some form of conversion e.g. percentage or a system when the maximum 255 = X number of gallons.

LED 0 will be used for the Fuel low warning light.

Node 1 (Display Node) will be used to display the fuel level and the warning light.

23.3 Further work

The fuel warning light comes on when fuel gets low. However, drivers are notorious for missing or ignoring warning lights. One thing that does grab our attention though is a flashing light.
Modify the program to produce a flashing light once the fuel gets to say half the fuel low level.

24 Notes for Exercise 4

For the program we need to add an Analogue sensor (the thermometer Component) and to monitor its reading. The analogue sensor reading consists of two bytes of data, a High byte and a Low byte. However for this example we will just use the High byte to simplify matters. Add the *SampleADC* macro, and a *ReadHigh* macro to read the fuel level into a suitable variable such as FUEL_LEVEL.
Add a *SetTxData* macro and put in the parameters *Buffer* 0 (for TX Buffer 0), *Count* = 1 (1 data register) and *D0* = FUEL_LEVEL (the data to be sent). You need to add data for the other data registers even though they are not used, so you will need to add a zero for each of them.

Follow this with a decision icon to see if the fuel is too low. Here we have opted to check FUEL_LEVEL against another variable called MIN_FUEL, which we will need to initialize at the start of the program.

If the fuel is too low we can send the TX Buffer 1 signal. By using two buffers we can send fuel data updates constantly, but only send the warning signal when needed.

24.1.1 Receiving the data

Firstly set up a program that monitors the CAN network for a signal with the Message ID 100.
Once this is found we need to extract the data, in this case the FUEL_LEVEL data that was sent. We retrieve the data with the *GetRxData* macro. We need to supply the parameters *Buffer* (0 for RX Buffer 0), and the index of the data register we want, in this case the item we want is *Index* = 0 which corresponds to item D0 on the TX Buffer (see Fig. 23.1).

Set the Return value to a convenient variable such as FUEL_LEVEL. Now that you have the data you can process it and output it to the LCD display. Alternatively you could use various output methods such as a light bar graph that falls away as the fuel is used up.

There is another useful macro to use when retrieving data: *GetRxDataCount*, which returns the data length for the specified buffer. This allows you to both check that there is some data there (0 = no data sent), or to find out how much data is there if a variable amount is possible.

24.1.2 Receiving the warning signal

The warning signal is simple. If a message with the Message ID 101 occurs then we need to light the warning LED. We could just deal with it straight after we have read and processed the fuel level data. Or we could deal with it elsewhere. Maybe there is another specific node that handles the warning signals. You don’t need to handle all the messages sent from one node at the same receiving node. The great benefit of CAN is that you don’t have to do everything in one place. We could set up a third node to handle warning signals.

24 Notes for Exercise 4

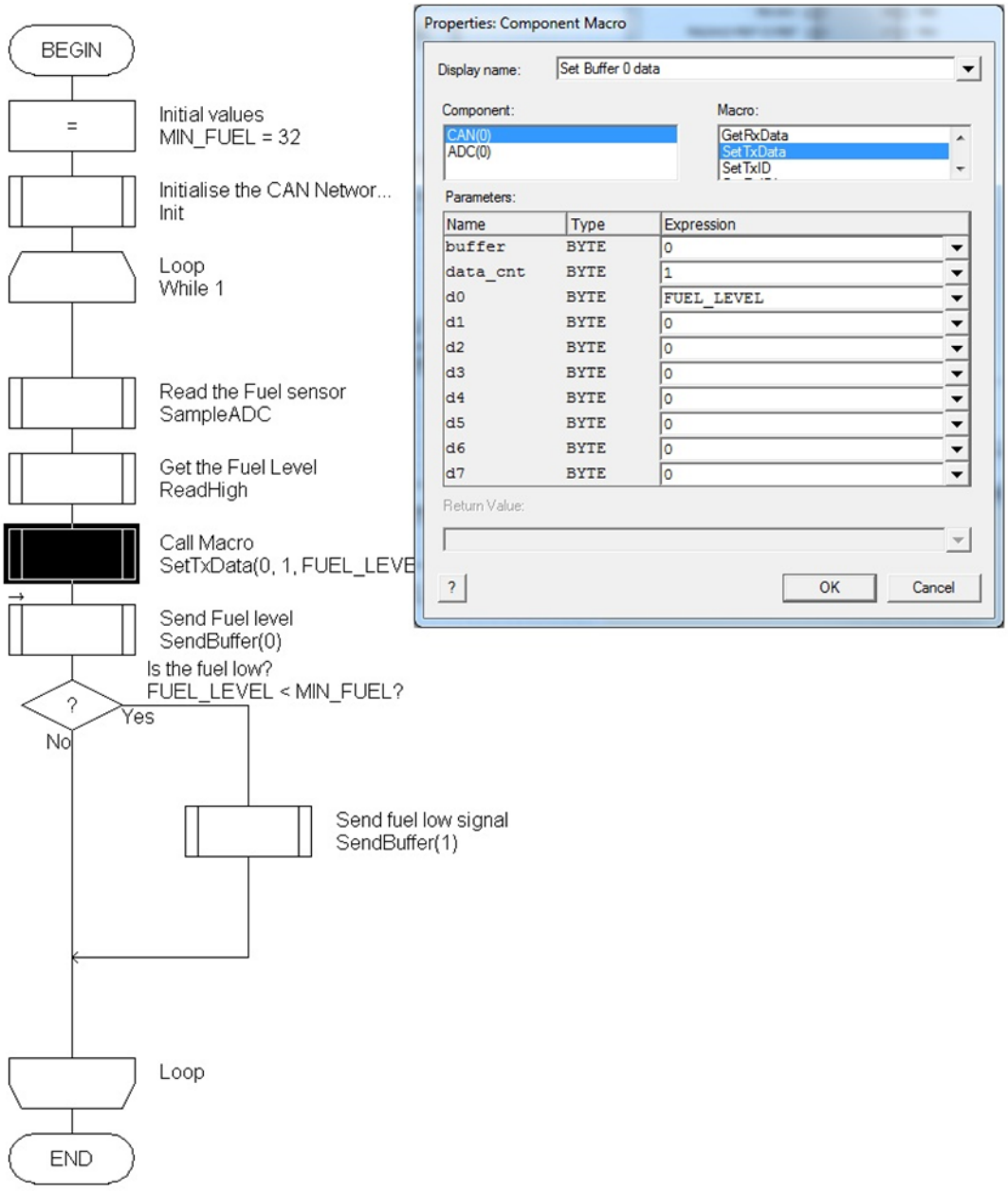


Figure 24.1 *Setting the macro properties*

25 Demonstration 4: Fuel gauge and warning light

This example shows a CAN message system with data in action.

21 Setup

	PIC BL0011			Arduino BL0055		
	Port A	Port B	Port C	A0-5	D0-7	D8-13
Node 1	BL0169		BL0140	BL0169		BL0140
Node 4	BL0129		BL0140	BL0129		BL0140

We will be using Node 1, the display node, to display the data. We will be using Node 4, the Sensor Node, fitted with the rotary potentiometer in socket 1 (Pins 0,1) to send the data.

Open the file CAN_EXAMPLE_04_RECEIVE.FCFX in Flowcode and download it to Node 1 (The Display node).
Open the file CAN_EXAMPLE_04_SEND.FCFX in Flowcode and download it to Node 4 (The Sensor node).

25.2 Viewing the messages

If you open up CANKing and view the network traffic you will see the message being sent periodically for the fuel level, and also for the fuel warning light should the fuel get low.

25.3 The fuel level

When the program starts the fuel level will be displayed on the LCD.
Moving the variable potentiometer on the sensors Node will change the fuel level displayed.

25.4 The warning light

When the fuel level becomes too low a warning light is activated.

25.5 Viewing the data

If you check the network traffic sent in CANKing you will find a stream of messages being sent with Message ID 100. This is the data being sent. Note how the message has a single item of data. Does the data relate directly to the figure displayed? Or is it altered in some way – e.g. raw data to gallons?
Where does the warning light message come from?

25.6 Conclusions

This demonstration shows data being passed. Although in this example it is just one item, it demonstrates the potentials of CAN. Not only can you send specific messages that will be picked up by specific receiving nodes, but you can actually pass data to them as well - fuel, speed, height, pressure, On/Off states, anything. If it can be converted into data it can be sent.

26 Advanced CAN Networking

This section discusses advanced CAN concepts such as setting Message ID's Masks and Filters and the CNF settings. The message structure is examined, and other issues such as network wiring are also looked at.

26.1 Exercises

No exercises or demonstrations are provided for this section. Existing exercises can be adapted to use masks and filtering or exercises can be generated to demonstrate the mask and filter mathematics examples given below.

26.2 Masks and filters

26.2.1 Masks and filters - the general concept

An important but complex part of our implementation of CAN is Masks and Filters.
Masks are used to modify the Message ID values.
Message ID's are checked against the Filters to see if they should be accepted or not.

26.2.2 Masks

Masks modify the Message ID values received by the buffer. They modify the value by removing the mask bits from the incoming Message ID. This can be used to make a number of different Message ID's appear to be the same value. For example a mask could remove the tens digit from a message, so that messages ID 120, 123, 140 and 165 would appear as 100, 103, 100, 105 respectively. For details on the masking process see the examples given below.

If the Filters were set up to accept Message ID 100 then Message ID's 120 and 140, which are both converted to 100 by the mask would be accepted. Such a system could be used to modify a batch of Message ID's that all share a related function – e.g. several warning signals could go to their respective warning lights nodes using their separate Message ID's, and because they all mask to the same value, get picked up and acted upon by a central Master Warning light node.

26.2.3 Filters

Filters are the doormen of the CAN system. They check the incoming Message ID's against the Filters - a list of Message ID's that they can accept. If your name is on the list you will be allowed in. If not… sorry, try somewhere else mate.

Each filter has a check box next to it on the Properties Panel that can be used to enable or disable that particular filter.

There are three general filter settings and either 2 or 4 specific filters depending on the RX Buffer.

- Accept all Messages – all Message ID's are accepted. The node will respond to any Message arriving on this buffer.
- Reject All Messages – effectively an 'Off' switch for the buffer. The node will not respond to any message on this buffer. May seem odd, but can be used to turn off an RX Buffer that is not being used. Why have RX Buffer 1 active if you are only going to use RX Buffer 0?
- Use Masks and Filters – Uses the Masks and filters to modify and check Message ID's to see if they should be accepted or not.

26.2.4 RX Buffer Properties

To enable Masks and Filters, select the 'Use Mask and Filter' setting, the other two settings are 'Accept All' and 'Reject All'. The Filter values can be set in Properties Panel, for example see the RX Buffer 0 properties displayed below.

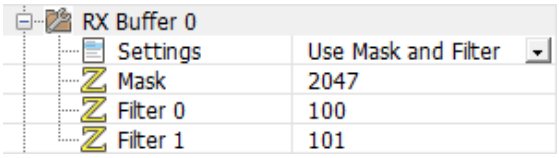


Figure 26.1 *RX Buffer 0 Mask and Filters values, and the Message ID's of each Filter.*

The Buffer RX1 properties are similar to the Buffer RX0 properties shown above; however more filters are available for use.

How to work out which messages will be trapped by a particular mask/filter combination

The best way is to work through some examples:

Note that all values for Message ID's, masks and filters are numbers between 0x000 and 0x7FF.

26.3.1 Using masks and filters: Example 1

Mask 0 = 0x0FF Filter 0 = 0x100 Filter 1 = 0x050

Mask 0 =	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
Filter 0 =	0	0	1	0	0	0	0	0	0	0	0
Filter 1 =	0	0	0	0	1	0	1	0	0	0	0

In binary, this looks like:

For the mask, a '1' signifies 'check this bit' and a '0' means 'ignore this bit'

So, these filters will accept the following messages ("x" = don't care)

Mask 1 =	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
Filter 2 =	x	x	x	0	0	0	0	0	0	0	0
Filter 3 =	x	x	x	0	1	0	1	0	0	0	0

i.e.:

Filter 0 accepts 0x000, 0x100, 0x200, 0x300, 0x400, 0x500, 0x600, 0x700
Filter 1 accepts 0x050, 0x150, 0x250, 0x350, 0x450, 0x550, 0x650, 0x750

26.3.2 Using masks and filters: Example 2

Mask 1 = 0x350 Filter 2 = 0x200 Filter 3 = 0x123 Filter 4 = 0x3FF

Rewriting in binary:

Mask 1 =	**1**	**1**	**1**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**
Filter 2 =	0	1	0	0	0	0	0	0	0	0	0
Filter 3 =	0	0	1	0	0	1	0	0	0	1	1
Filter 4 =	1	1	1	1	1	1	1	1	1	1	1

Here, the mask will only check 4 bits and ignore the other 6. Here's what the filters will accept:

Mask 1 =	**1**	**1**	**1**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**
Filter 2 =	0	1	0	0	x	0	x	x	x	x	x
Filter 3 =	0	0	1	0	x	1	x	x	x	x	x
Filter 4 =	1	1	1	1	x	1	x	x	x	x	x

They will actually trap a lot of messages (64 each!):
Filter 2 = 0x200, 0x201, 0x202, ... 0x220, 0x221, ... 0x280, 0x281, ... 0x2A0, 0x2A1, ... 0x2AF
Filter 3 = 0x100, 0x101, 0x102, ... 0x120, 0x121, ... 0x180, 0x181, ... 0x1A0, 0x1A1, ... 0x1AF
Filter 4 = 0x750, 0x751, 0x752, ... 0x770, 0x771, ... 0x7D0, 0x7D1, ... 0x7F0, 0x7F1, ... 0x7FF

This second example is not very practical. In general, it is more logical to set the mask so that each filter accepts a consecutive range of messages.

As you can see, the mask determines which bits of the filters are actually looked at. Setting the mask to 0x000 will effectively mean that the filter will accept any incoming message. Also, the value of the mask directly relates to how many messages each filter will trap - i.e. 2^(number of '0' bits in the mask).
A useful way to use the mask would be to ignore the least significant bits. Let's say that you wanted the filters to accept 16 messages each - setting the Mask 0 to 0x7F0 would achieve this. Then, setting the filters to the following…:
Filter 0 = 0x100
Filter 1 = 0x110
…would mean that the following messages are accepted:

Filter 0 = 0x100, 0x101, 0x102, 0x103, 0x104, ... 0x10D, 0x10E, 0x10F
Filter 1 = 0x110, 0x111, 0x112, 0x113, 0x114, ... 0x11D, 0x11E, 0x11F

Of course, for simple CAN applications you may wish to only accept one or two messages. Setting the mask to 0x7FF in this instance would mean that only the message ID specified by each filter would be accepted, e.g.

Mask 1 = 0x7FF
Filter 2 = 0x100
Filter 5 = 0x200
This would mean that only messages 0x100 and 0x200 would be accepted into RX Buffer 1.

26.4 CNF settings

The 'Properties' section on the 'Properties Panel' includes options which determine the CNF settings.
These properties s modify the CNF values automatically, and should suffice for most situations. However there may be a situation where you need to manually set these values. Chances are, in such a case, you will have been given the values to be set. If so you can simply enter the values directly.
If not you may need to consult the CAN documentation and tables to determine the CNF values to set. Links to CAN documentation can be found at the end of this document.
Remember - generally, it is best to make sure that these settings are the same for every CAN node on the bus, although only the bus rate value must be consistent - adjustments for the sample point and SJW are only ever required when using unusually long cables between the nodes.

26.5 Message details

When a message is sent not only does the Message ID and any associated data get sent, but a number of markers and wrapper elements are added as well. Fig. 25.2 shows what a message looks like.

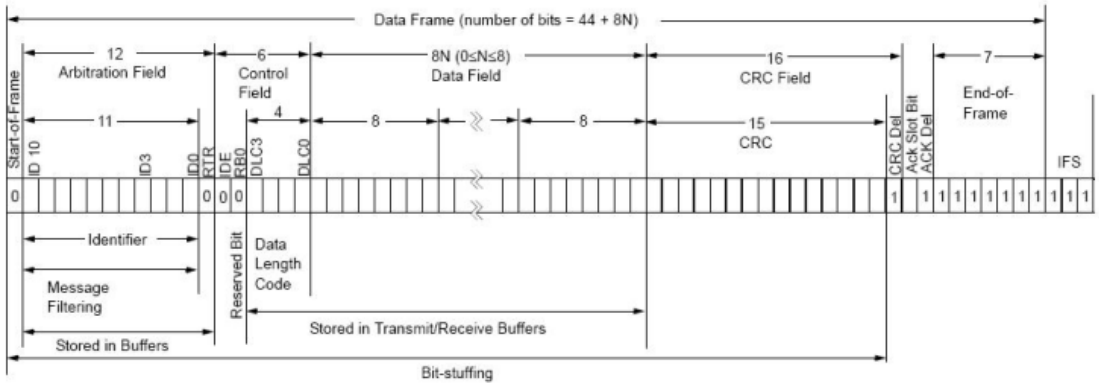


Figure 26.2 *Message format*

Note that the 1's and 0's refer to dominant and recessive voltage levels and not CMOS or TTL logic levels. Basically CAN is able to use different voltage levels so that it can be adapted to different electrical environments.

26.6 Error detection

CAN has a number of automatic error detection systems that flag errors should they occur. If an error is detected an error signal is sent destroying network traffic and the nodes on the network take the appropriate action e.g. discarding the message with the error in it.

Error tracking is complex, and for the full details you will need to check the CAN specifications, but can be summarized briefly as follows: Nodes track how many transmit errors and receive errors flagged on two counters. Transmit errors increment the transmit error counter by 8 per error, and receive errors increment the receive error counter by 1. Successful transmit and receives decrement the values. This means the transmit error counter will rise at a faster rate, appropriate as faulty transmitters are more likely to be problem. It helps ensure that a faulty transmitter will hit the switch off point before the nodes receiving its signal and gaining receive errors.

If either value hits 127 then the node becomes error passive. In error passive the node still transmits errors, but they no longer destroy network traffic i.e. the node is declaring itself to be error prone, and potentially the problem so it stops destroying the network traffic as the messages may not be the problem. However its error counts continue to increment as normal.

If either of the error counters hits 255 then the node goes Bus Off and stops transmitting. The node has identified a problem with itself and removed itself from the network.

In summary the Node goes from screaming 'major error – all stop', to shouting about errors but being ignored, to being switched off.

26.6.1 Bit monitoring

As signals are transmitted they are also check by the receiver part of the CAN system for signal level. If the level detected is not what it should be a *Bit Error* is flagged.

26.6.2 Bit Stuffing.

CAN needs to know if a long signal is a fault rather than part of the message. To help with this CAN performs Bit Stuffing. If 5 bits of the same value are sent (0 or 1) then a 6th bit is inserted with the opposite value to let the network know that it is not a fault. This extra bit is automatically removed by the nodes. Should this bit stuffing not occur then CAN will realize that there has been a problem somewhere and fire a *Stuff Error*.

This is required as there is no separate clock signal in CAN. Instead the data rate is synchronized using the throughput of data. Bit stuffing helps the CAN system synchronize this data clock rate.

26.6.3 ACK bit

When a node sends a message the ACK or Acknowledgment bit is set to 0 (recessive). When a node receives a message acknowledges the message by returning the signal with the ACK bit set to 1 (dominant). This does not mean that the message got through to its intended destination; merely that it was recognized by that particular receiving node as a legitimate message. However by checking the returned signal has the ACK bit set allows the sending node to signal an *Acknowledgment Error* if it is not.

26.6.4 Frame check

Certain parts of the CAN message have set formats and set signals. The message is monitored for errors in these parts. If an error is detected a *Form Error* is generated.

26.6.5 Cyclic Redundancy Check (CRC)

Messages contain a 15 bit Checksum that can be checked by receiving nodes. If the values do not match then a *CRC Error* signal can be fired.

26.7 Wiring and other practical issues

A common implementation of the CAN physical interface utilizes a twisted wire pair, which helps minimize errors due to voltage spikes and EMC interference. Networks are terminated by a resistor across the wires to help counter electrical interference. Nodes are added by connecting the node to the twisted wire pair. All nodes are interconnected via the network; no nodes are isolated from any other on the network.

Voltages are either *dominant* (signified by a 1in written format) or *recessive* (signified by a 0 in written format). There are no absolute voltages. The only requirement is for the system to be able to *distinguish between* the dominant and recessive signals. This frees us considerably as we can then work with signals appropriate to the physical interface most suitable for the system, rather than having to design the system with specific voltages in mind. However the particular IC's and other hardware used in your CAN system may have their own tolerances and expected levels which would need to be taken into consideration.

CAN only specifies the message format, not the physical layer. Whilst this gives us greater flexibility in wiring up a CAN system, it also means that wiring details can vary considerable between even similar systems. Other physical interfaces, such as single line fiber optics, are perfectly acceptable wiring solutions.

27 Reference data

This document was designed to teach you about the basics of CAN, the theory and concepts behind it, and practical exercises to increase your knowledge and skill in working with CAN. It should also provide you with the basis of developing your own CAN teaching program, with plenty of scope for demonstrations and practical work.

But that's not the whole story of CAN. Should you wish to take your study of CAN further, or to move into areas of industry that use CAN, there is much more to know, and much more to study!

27.1 CAN standards

CAN is an evolving specification. It has already advanced from the original Bosch specification and is currently available in Standard (Version 2.0A) and Extended (Version 2.0B) versions.

The system represented here is a Standard CAN system. Extended CAN has a number of differences, particularly with message format. The main practical difference is that Extended CAN uses 29 bit Message ID's in place of the 11 bit values used by Standard CAN. This allows Extended CAN to address something like 500 million nodes.
(Given the pain in the neck 11-8 bit conversions are, aren't you glad we didn't go for 29 bits!)

The CAN specifications can be obtained from the Bosch web site at:
www.semiconductors.bosch.de

There are also two ISO standards used for CAN transmissions.
ISO1159 is used for low speed networks (up to 125kbit/second).
ISO11898 is used for high speed networks (up to 1Mbit/second)
There are differences between the two standards with regard to wiring and voltage tolerances etc.
Should you require them the ISO standards can be purchased from the ISO web site at:
www.iso.org

27.2 Higher level protocols

Higher level protocols (HLP) are out of the scope of this course, but may be of interest for those who will be taking CAN further. Higher level protocols refer to the system languages that pass the data around and present it in a format that applications using that protocol can understand.

CAN only specifies the message format and leaves the higher-level protocol open. This allows different areas of industry, or different companies to develop or adopt their higher level protocols that best suit their needs, such as CANopen from Kvaser (www.kvaser.com).

For us though this means that there is no one higher-level protocol set to study. There are a number of higher-level CAN protocols on the market. Which one to learn and use may simply depend on whom you go to work for. CAN system designers may need to be flexible enough to accommodate design work in one or more of these higher-level protocols.

27.3 Acronyms and abbreviations

ACK Acknowledge
ADC Analogue-to-digital converter
CAN Controller area network
CANH CAN high
CANL CAN low
CMOS Complementary metal oxide semiconductor
CRC Cyclic redundancy check
DAC Digital-to-analogue converter
ECU Electronic control unit
EMC Electromagnetic compatibility
HLP High level protocol
IC Integrated circuit
ID Identifier (or identification)
ISO International Standards Organization
LCD Liquid crystal display
LED Light emitting diode
RPM Revolutions per minute
RX Receive
SJW Synchronization jump width
TTL Transistor-transistor logic
TX Transmit
USB Universal serial bus