PRINCIPLES, PROJECTS, PROGRAMMING

CONTROLLER AREA NETWORK PROJECTS

Dogan Ibrahim

elektor

LEARN | DESIGN | SHARE

Controller Area Network Projects

Dogan Ibrahim

Controller Area Network Projects

Dogan Ibrahim

Elektor International Media
www.elektor.com

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

ISBN 978-1-907920-04-2

Prepress production: Jack Jamar | Graphic Design, Maastricht
First published in the United Kingdom 2011
Printed in the Netherlands by Wilco, Amersfoort

119009-1/UK

Contents

Preface

The Controller Area Network (CAN) was originally developed to be used in passenger cars. Today, CAN controllers are available from over 20 manufacturers, and CAN is finding applications in other fields, such as medical, aerospace, process control, automation, and so on.

With the establishment of the Can in Automation (CiA) association in 1992, the manufacturers and users have come together to exchange ideas, and develop the CAN standards and specifications.

This book is written for students, for practising engineers, for hobbyists, and for everyone else who may need to learn more about the CAN bus and its applications. The book assumes that the reader has some knowledge on basic electronics. Knowledge of the C programming language will be useful in later chapters of the book, and familiarity with at least one member of the PIC series of microcontrollers will be an advantage, especially if the reader intends to develop microcontroller based projects using the CAN bus.

The book should be useful source of reference to anyone interested in finding an answer to one or more of the following questions:

- What bus systems are available for the automotive industry ?
- What are the principles of the CAN bus ?
- What types of frames (or data packets) are available in a CAN bus system ?
- How can errors be detected in a CAN bus system ?, and how reliable is a CAN bus system ?
- What types of CAN controllers are there ?
- How can one create a CAN based project using a microcontroller ?
- How can one monitor data on the CAN bus ?
- What is an On Board Diagnostic (ODB) tool, and how can I use one ?

The book consists of 12 Chapters:

Chapter 1 presents the basic features of automotive bus systems, and discusses the advantages and disadvantages of various bus systems used in automotive industry.

Chapter 2 provides a brief introduction to the history of CAN bus.

Chapter 3 is about the ISO/OSI standards, and examines the various layers of this standard. The layers covered by the CAN bus are identified and described briefly.

Chapter 4 covers the physical layer structure of the CAN bus. The physical layer standards, and various cabling mechanisms are described in this chapter.

Chapter 5 presents the various frames of the CAN bus. Each frame type has been examined in detail and examples are given where appropriate.

Chapter 6 is about one of the most important topics in CAN bus, the error conditions. The sources of various errors are described with reference to how errors can be detected on the bus, and what actions can be taken once an error condition is detected.

Chapter 7 explores the data exchange mechanisms on the bus. Examples are given to demonstrate how data frames can be transmitted by one node and then received by a number of nodes. The important topic of acceptance filters and acceptance masks are described in this chapter.

Chapter 8 is about one of the most important topics in CAN bus, the timing. This chapter presents the various timing conditions and gives several examples to show how the timing segments can be selected in various CAN bus implementations.

Chapter 9 reviews the various commercially available CAN bus development tools. The hardware as well as software development tools, and bus analyzers are described in this section. Example products are given from different manufacturers.

Chapter 10 presents the basic principles of CAN bus controllers, and gives example controller products from various manufacturers.

Chapter 11 is very important for anyone who wishes to develop a microcontroller based project using the CAN bus. The chapter reviews the basic features of PIC microcontrollers, and makes an introduction to a high level C language that can be used to develop software for microcontrollers. Several working example projects are given in this chapter using a microcontroller with a CAN bus.

Finally, *Chapter 12* is about the On Board Diagnostics (OBD) and presents various commercially available tools that can be used for diagnostics. An example tool with a diagnostic application example is given in this chapter.

Dogan Ibrahim
London, 2011

Acknowledgements

The following material is reproduced in this book with the kind permission of the respective copyright holders and may not be reprinted, or reproduced in anyway, without their prior consent.

Figure 1.7 and 1.8 are taken from the Renesas Electronics Corporation Document "*Introduction to CAN*", Application Note No: REJ05B0804-0100/Rev. 1.00, web-site: csc@resesas.com.

Figure 2.4 is taken from Omitec web-site: www.omitec.com

Figure 2.5 is taken from Amazon web-site: www.amazon.co.uk

Figure 2.10 is taken from the Bosch Engineering GmbH web site: www.bosch-motorsport.com

Figure 8.2 and much of section "*Selecting the Timing Parameters*" in Chapter 8 are "Copyright of Freescale Semiconductor, Inc, 2011," and are used with permission. These items were taken from Application Note AN1798 (CAN Bit Timing Requirements, written by Stuart Robb).

Figure 9.2 and Figure 9.3 are taken from the web-site of Renesas Electronics Corporation: http://am.renesas.com

Figure 9.3 is taken from the web-site of Custom Computer Services: www.ccsinfo.com

Figure 9.4 is taken from the web-site of PEAK-System Technik GmbH: www.peak-system.com

Figure 9.5 is taken from the web-site of Fujitsu Semiconductor Europe GmbH: http://emea.fujitsu.com/semiconductor

Figure 9.6 is taken from the web-site of Wilke Technology GmbH: www.wilke.de

Figures 9.7 – 9.9 are taken from the web-site of mikroElektronika: www.mikroe.com

Figure 9.10, 10.3, 10.4, 11.1, 11.2, 11.18, 11.22 are reprinted with permission of the copyright owner, Microchip Technology Incorporated. All rights reserved. No further reprints or reproductions may be made without Microchip Technology Inc's prior consent. The material above are taken from Microchip Technology Inc. data sheets PIC18FXX2 (DS39564C), PIC18FXX8 (DS41159E), and the web-site: www.microchip.com

Figure 9.11 is taken from the web-site of Softing: www.softing.com

Figure 9.12 – 9.14 are taken from the web-site of Computer Solutions:
www.computer-solutions.co.uk

Figure 9.15 is taken from the web-site of Wuensche: www.ems-wuensche.com

Figure 9.16 is taken from the web-site of IXXAT Automation GmbH: www.ixxat.com

Figure 9.17 is taken from the web-site of LeCroy Europe GmbH: www.lecroy.de

The following are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries: MPLAB, PIC, PICmicro, PICSTART.

Parts of Chapter 11 are taken from one of author's books entitled "*Advanced PIC Microcontroller Projects in C: From USB to RTOS with the PIC18F Series*", Newnes, 2008.

Chapter 1. The automatic Bus Systems

Today's vehicles are highly complex machines incorporating mechanical and electronic parts. The number of electronic components used in vehicles has increased rapidly in recent years. As a result of the increase in safety, comfort, and performance requirements, we see many more electronic components being added to present day vehicles. As a result of this, there has been an increasing demand to connect these electronic components in such a way that they can communicate reliably, safely, and in real-time.

Today automotive electronic systems contain many sensors, actuators, monitoring units, entertainment and navigation systems that are distributed and embedded in different parts of a vehicle. It is estimated that in a typical modern passenger car over 70 electronic control units are used, exchanging over 2500 signals, and this number is increasing with increased complexity.

In the past, the electronic units in a vehicle used to be connected in a complex way with hundreds of wires running at different parts of the vehicle. Consequently, it was very difficult to trace an electronic fault. There was no co-ordination between different parts of the electronics as each electronic part was controlled independent of the others. Maintenance and repair of the vehicle electronics was extremely difficult as in many cases it was not easy to locate and change a faulty component.
Figure 1.1 shows a traditional old style vehicle electronic system.

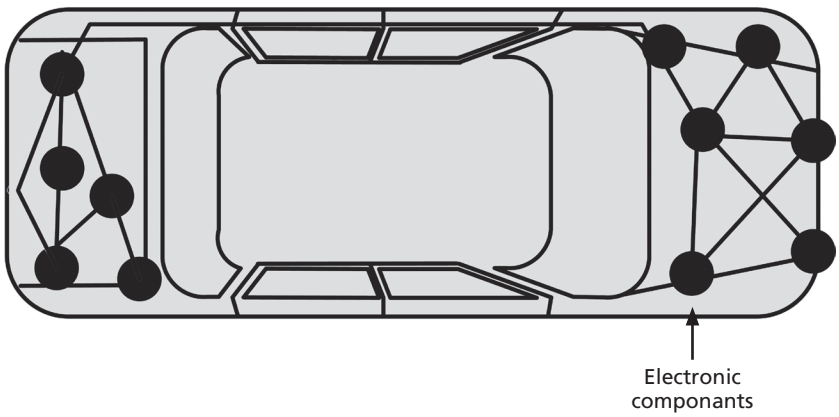


Figure 1.1 Traditional old style vehicle electronics

As the complexity of the vehicle electronics had grown by many factors, it had become difficult for the manufacturers to design safe and reliable electronic systems based on the old traditional methods. The current requirements can not be met with a simple electronic control unit. The solution is to network the various electronic modules with a high performance network. This is why it had become necessary to design a network based electronic system where the electronic modules could easily be attached to a network and then controlled from a central intelligent unit (e.g. the Engine Control Unit). This resulted in an "intelligent" car where many sensors and actuators are used to sense the environment and perform many functions. One example is the automatic turning on of the lights when it becomes dark or when the car goes through a tunnel. Another example is the automatic operation of the wipers when the rain starts, and so on.

One of the advantages of a network based system is that it is relatively easy to trace and detect a faulty module. In addition, the wiring is a lot simpler and easier to maintain. For example, by communicating with the central intelligent unit one can tell whether or not the overall electronics system is healthy, and if not, the faulty modules can easily be detected. A networked system also allows the various modules on the bus to communicate with each other and exchange information if required. For example, the intelligence unit can receive the engine temperature value from the temperature sensor module. This temperature can then be displayed on an electronic dash-board. Should the temperature be too high, appropriate messages can be sent to responsible parts of the engine and corrective measures can easily be taken. Figure 1.2 shows a modern vehicle where a bus system is used to connect and control the electronic modules.

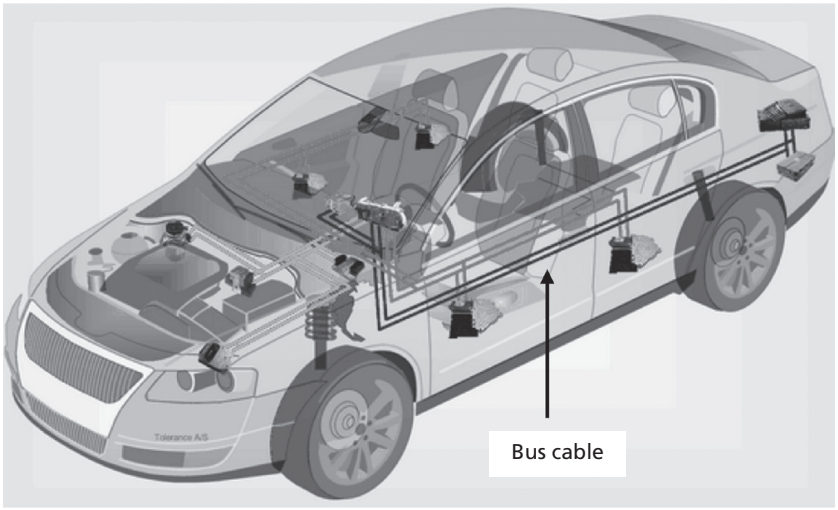


Figure 1.2 Modern vehicle where a bus system is used to connect the electronic modules together

This Chapter provides an overview of the most important vehicle bus systems currently used in vehicles, and provides a table to compare the advantages and disadvantages of each system.

1.1 Vehicle Bus Systems

The vehicle bus systems (or networks) were classified in 1994 by the Society of Automotive Engineers (SAE). According to this classification, bus systems were classified based on their bandwidth (i.e. data rate) and functions of the bus system. The classification divides bus networks into four: Class A, Class B, Class C, and Class D.

Class A networks are low-speed low-cost networks with data rates less than 10 kbps. These systems are mainly used in body of the car.

Class B networks operate between 10 and 125 kbps and are used for information exchange. e.g. instrument cluster, vehicle speed and so on.

Class C networks operate between 125 and 1 Mbps, and are used in a wide range of applications, such as engine control.

Class D networks operate above 1 Mbps and they are used mainly for telematics applications.

There are many automotive bus systems, some developed by vehicle manufacturers on their own, and some developed jointly with semiconductor manufacturers. A list of the vehicle bus systems is:

- CAN bus
- LIN bus
- FlexRay
- MOST
- Byteflight
- DSI bus
- Intellibus
- SAE J1850
- BST bus
- NML bus
- And others…

In this section we shall be looking at the basic properties of the most commonly used automotive bus systems, namely:

- CAN bus
- LIN Bus
- FlexRay
- MOST
- Byteflight
- Intellibus

1.1.1 CAN Bus

The Controller Area Network (CAN) bus is the main topic of this book. At this section we shall be looking at the basic properties of this bus together with the other automotive busses.

CAN is a serial two-wire multi-master bus that was developed by Robert Bosch GmbH in 1980s. It is one of the most widely used automotive busses today. The physical layer of CAN consists of a pair of twisted cables. CAN provides reliable, robust, and fast communication up to 1 Mbps (with 40 m bus length). CAN 2.0A is the original CAN, consisting of the fields: Start of Frame bit, 18-bits header (having 11-bits message identifier), 0-8 bytes data, 15-bits Cyclic redundancy Check (CRC), 3-bits acknowledgement slot, and 7-bits of End of Frame.

CAN bus is based on the CSMA/CR (Carrier Sense Multiple Access/Collision Resolution) mechanism to prevent frame collisions during transmissions on the bus. Each CAN node monitors the bus and when the node detects that the bus is idle, it may start transmitting data. If other nodes on the bus attempt to send data at the same time, arbitration will take place and the node with the highest priority (lowest message identifier) will win the arbitration and send its own data. CAN bus has a simple error detection and recovery mechanism. Receiving nodes check the integrity of the messages by looking at the CRC fields. If an error is detected, the other nodes on the bus are informed by error flag messages. Figure 1.3 shows a typical CAN bus implementation with two nodes A and B. CAN is Class A/B type network.

1.1.2 LIN Bus

The Local Interconnect (LIN) bus is a low-cost bus, operating at 20 Kbps. This bus is mainly used for body/comfort functions. LIN is a single wire, single master/multiple slave type bus system where the vehicle chassis is used as the return path. In a typical application, the master broadcasts a message with a message header, asking for data and the slave that has the correct message header sends the requested data, including a checksum for error checking. Typical LIN bus applications include the control of small motors for wipers, sun-roof control, heating control, rain sensor control etc. where a wide bandwidth is not required. LIN bus is used in applications where the implementation of CAN

bus would be too expensive. The initial LIN specification was defined by a consortium consisting of BMW, Audi, Volvo, VW, Motorola, Volcano, and DaimlerChrysler.

LIN bus is based on Serial Communications Interface (UART) with 8-bit data. Figure 1.4 shows a typical LIN bus implementation with a master and two slaves. LIN is Class A type network.

1.1.3 FlexRay

FlexRay was initially developed by MBW and DaimlerCrysler in 1999 as a fast, efficient, and error free automotive bus system. FlexRay is suited to real-time high-speed applications as it supports a bandwidth of up to 10 Mbps. Both electrical and optical transmission medium can be used. FlexRay is mainly used in safety critical applications and in real-time high-speed engine control.

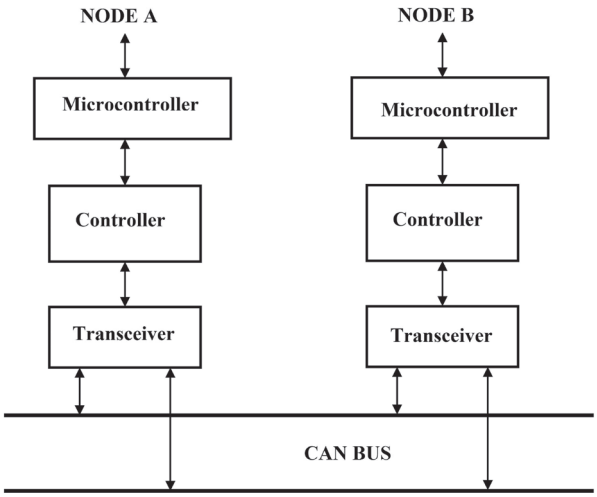


Figure 1.3 CAN bus with two nodes

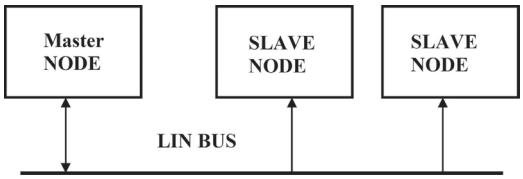


Figure 1.4 LIN bus with master and two slaves

Flexray is based on the TDMA (Time Division Multiple Access) mechanism where each device on the bus has a fixed slot allocated to it. TDMA is a deterministic bus access mechanism as it is known when a device will respond on the bus. If a device has no data to send then its time slot is wasted. If on the other hand a device can not send its data in the allocated time slot it has to wait until its time for the next time slot arrives. In order to increase the efficiency the communication is divided into fixed and dynamic time slots where the time slot of a device can be extended if required. One advantage of FlexRay is that the network configuration can be bus, star, multiple star and so on. FlexRay is Class D type network.

1.1.4 MOST

The Media Oriented Systems Transport (MOST) bus is mainly used in automotive telemetric and multimedia applications, such as audio control, video, navigation, communication and so on. The initial MOST network was developed by BMW and DaimlerChrysler in 1998.

MOST supports a very high bandwidth: 28.8 Mbps synchronous, and 14.4 Mbps asynchronous. An optical medium is used for data transmission which is free of any electromagnetic radiation or interference.

In MOST networks one device is the master (called the timing master) and the others are slaves. Data is sent in frames where each frame is 512 bits. MOST communicates using the TDM/CSMA (Time Division Multiplex/Carrier Sense Multiple Access) protocol. MOST is a Class D type network.

1.1.5 Byteflight

Byteflight has been developed by BMW and it offers 10 Mbps bandwidth. Byteflight is mainly used in highly safety related networks, such as automotive and avionic systems (e.g. in vehicle airbags, body electronics, and so on). To make it possible to be highly safe, the data protocol must be fault-tolerant and deterministic. The data control mechanisms before Byteflight had been either event-controlled, or time-controlled. Event controlled (e.g. CAN) only transmits data when data is ready or when a data request arrives. Time-controlled data protocols grant time to each node in accordance with a pre-defined sequence. The number of messages to be transmitted can not generally be changed during operation as the number of allocated time slots are fixed. Byteflight protocol combines both the event-controlled and time-controlled protocols, and guarantees deterministic latencies for a specific number of high-priority messages, and flexible us of the bandwidth for lower priority messages. Byteflight is based on message oriented transmission with FTDMA (Flexible Time Division Multiple Access) mechanism, and mainly using a star topology. Similar to CAN, each message in Byteflight possesses a unique identifier to avoid collisions on the bus. Byteflight mes-

sages consist of 6-bit starting sequence, an identifier byte, a length byte, up to 12 data bytes, and two CRC bytes for high level error checking.

The individual nodes in a Byteflight network are connected together via fiber-optic cables and an active star coupler. The lines are operated bi-directionally using half-duplex transmission. Byteflight network can easily be expanded by incorporating more nodes. New messages can be added to the system without having to change the existing software. The high transmission rate of 10 Mbps and high level of immunity to electromagnetic interference made possible by fiber-optic transmission offers great advantages to Byteflight bus system in automotive applications. Figure 1.5 shows the typical bus topology of the Byteflight network.

The first use of the Byteflight in mass production was by BMW in a networked passive safety. Byteflight supports various network protocol in a mixed bus environment. For example, CAN bus and Byteflight can co-exist and can communicate in the same environment. Special controllers are used to transfer data between Byteflight and other bus systems. Byteflight is a Class D type network.

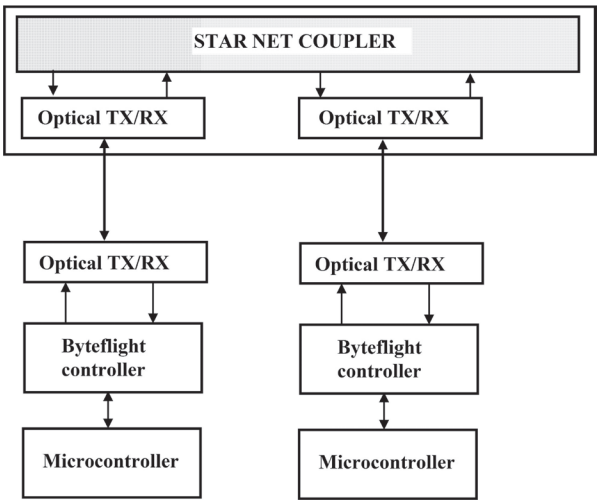


Figure 1.5 Byteflight topology

1.1.6 Intellibus

Intellibus is a high speed bus offering up to 15 Mbps bandwidth. It was initially conceived by Boeing to reduce the wiring complexity associated in distributed systems in aerospace applications. It is a low-cost bus, allowing a large number of sensors to be connected. A typical Intellibus network

consists of a Network Interface Controller (NIC) and 1 to 255 Intellibus Interface Modules (IBIM) that can be connected to sensors. The NIC can be installed in a PC or in some other electronic device. Normally, the NIC is downloaded with software to sample data from various IBIMs in a scheduled manner, at specific intervals. In a complex network, two or more NIC cards can be used to increase the node capacity. Special dedicated software is available to program the NIC and IBIMs.

Intellibus is used in automotive electronics, process control, automation, avionics, medical fields, and in several other fields. Figure 1.6 shows a typical Intellibus network. The network is of type Class D.

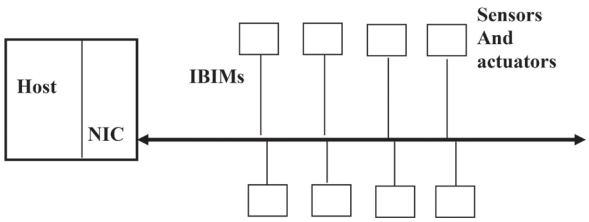


Figure 1.6 Intellibus

Another automotive bus standard that should be mentioned in this Chapter is the SAE J1850 (or simply J1850) which was developed in 1994. This standard was widely used in cars such as GM, Chrysler, and Ford. J1850 bus is used for diagnostics and data sharing applications. There are two versions of this standard: PWM (Pulse Width Modulation) with 41.6 Kbps using two wire differential physical layer, and VPW (Variable Pulse Width) with 10.4 Kbps using single wire physical layer. The two standards are incompatible with each other. The J1850 protocol is frame based and uses CSMA/CR arbitration where a frame consists of a Start of Frame, a header byte, data bytes, one byte CRC, and End of Data symbol (a 200us low pulse).
Most OBD (On Board Diagnostic) tools support the J1850 protocol for diagnostic purposes. The J1850 standard is old and is being phased out. J1850 standard is a Class B type network.

1.1.7 Comparison of Automotive Bus Systems

Table 1.1 is a summary of the commonly used automotive networks (or bus systems), comparing the various systems. For each system, the Class, General information, Bandwidth, and typical Application areas are given.

In general, LIN bus is used in low-speed automation, such as wiper motor control, rain sensor, etc. CAN bus is used in engine control, clutch control, and so on. FlexRay is used in very high performance and safety critical applications. This is a relatively new bus structure and is hoped to be the

automotive standard in the future. MOST is generally used in automotive multimedia, and navigation applications. This bus is optical based and is tolerant to electrical noise. Byteflight is also a new automotive network system and is well suited to high demand real-time safety critical applications, such as air-bag control. Byteflight is based on fiber-optics and as such it is much more tolerant to electromagnetic radiation and electrical noise. Intellibus offers high bandwidth and is mainly used in aerospace applications. It is not as safe and reliable as the Byteflight.

1.2 Summary

There are many automotive bus systems developed either by individual automobile companies, or jointly by automobile companies and semiconductor manufacturers. Currently one of the most commonly used bus systems is the CAN bus, which can be used in speeds up to 1 Mbps.
In general, in many vehicle electronics more than one bus system is used. For example, for low-speed non time-critical applications the LIN bus seems to be preferred. For higher speed applications, either the CAN bus, FlexRay, or the Byteflight seem to be the preferred network. Byteflight offers the advantage that it is based on fiber-optic medium and is more tolerant to electrical noise. The MOST network is common in automotive multimedia applications.

Table 1.1 Automotive bus systems

	Class	General	Bandwidth	Application
CAN	A, B	-Twisted pair -Low-cost -Widely used -Multiple master	1 Mbps	-Body -Engine
LIN	A	-Low-cost -Low-speed -Master/slave -Single wire	10 Kbps	-Body -Comfort
FlexRay	D	-Very high speed -Future standard -Master/slave (for synchronization)	10 Mbps	-Engine -Safety
MOST	D	-Data efficient -Standard for multimedia -Master/slave (for synchronization) -Fiber-optic	25 Mbps	-Multimedia
Byteflight	D	-High speed -Master/slave - Fiber-optic	10Mbps	-Safety critical
Intellibus	D	-High speed	15 Mbps	-Body -Engine -Safety

Figure 1.7 and Figure 1.8 show examples of passenger cars where several bus systems are in use (source: *Introduction to CAN – Application Note REJ05B0804-0100/Rev. 1.00)*). In Figure 1.7, the CAN bus can be seen at the left of the figure, connected to the instruments, climate control, lights, steering wheel, power train, and the lock system. The MOST bus can be seen at the lower-left part of the figure, connected to the multimedia equipment such as speakers, digital radio, vehicle computer, and navigation equipment. In this particular application, the FlexRay can be seen at the lower-right of the figure, connected to the engine, steering system, and the brakes. A diagnostic tool, connected to CAN bus at the bottom-right of the picture is used to check the state of the vehicle. InFigure 1.8, the headlights, air-conditioner, wipers, doors, and windows are controlled by the LIN bus, while the FlexRay controls the brakes. Most of the other parts of the car are controlled by the CAN bus.

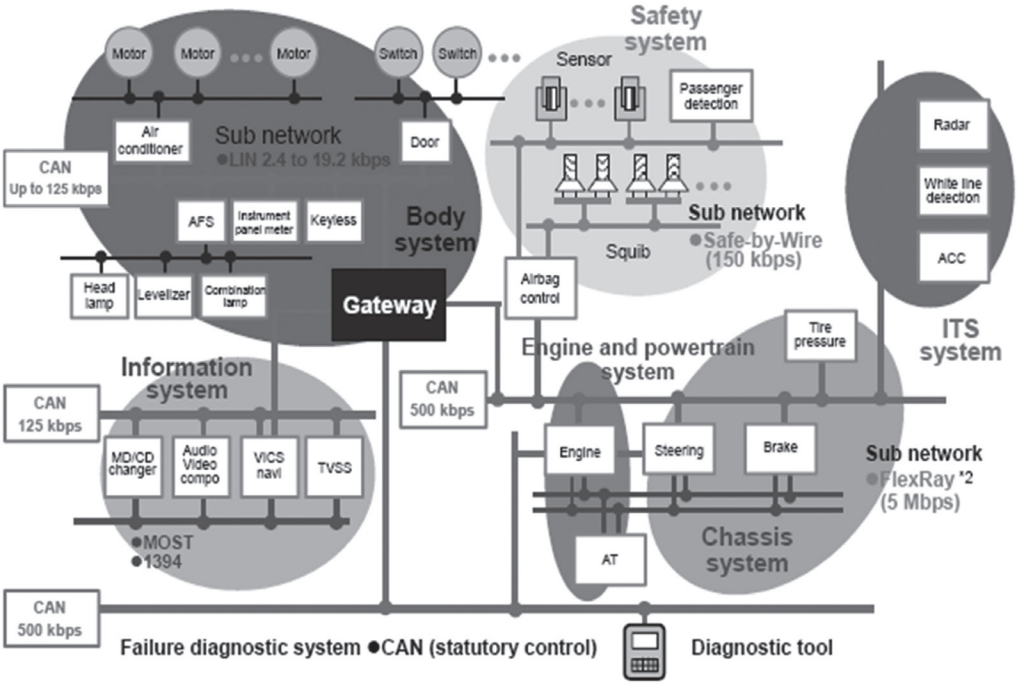


Figure 1.7 A typical passenger car with several bus systems

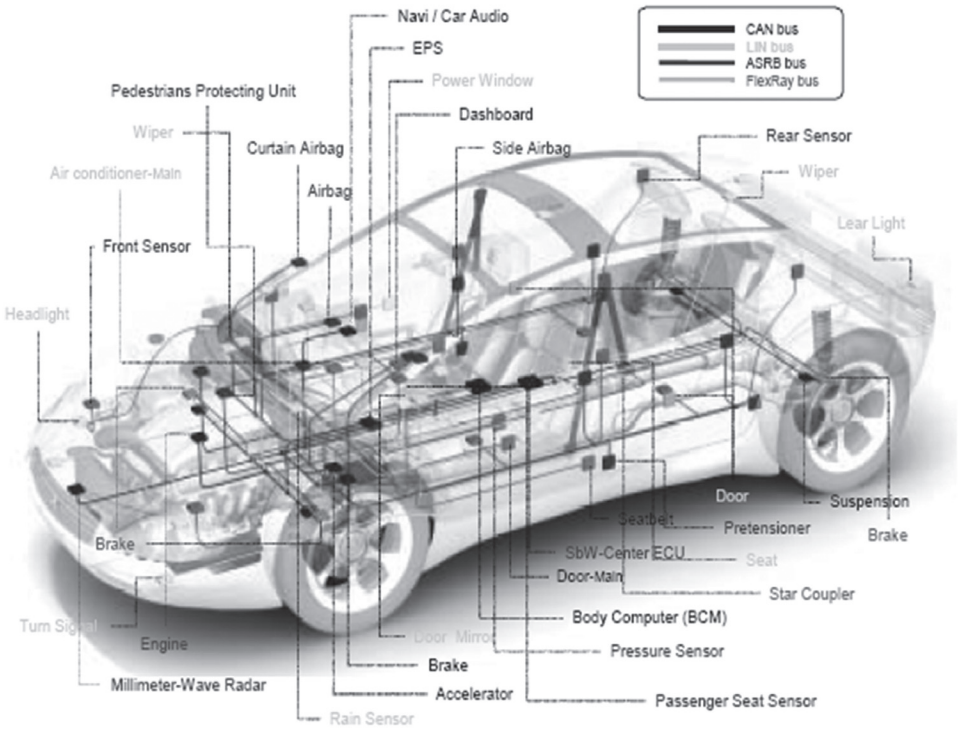


Figure 1.8 Another passenger car with several bus systems

Chapter 2 A brief history of CAN

The CAN (Controller Area Network) serial bus system was first introduced in February of 1986 by the engineers at the Robert Bosch GmbH in Germany. At the time, the engineers were looking for and investigating a bus system to use in automobiles that would simplify and at the same time improve the functionality of the complex automotive electronics. It was evident that the use of a serial bus system would reduce the wiring complexity, enable new and enhanced functionalities to be added easily, and at the same time make it considerably easier to maintain the large number of wiring in working order.

The initial investigation by the engineers revealed that at the time none of the existing serial bus systems offered the required communication reliability, safety, real-time response, and data correction. It was then agreed by the engineers that the best option was to develop a new bus standard, specifically to meet the demands of the real-time automotive engineering requirements.

Table 2.1 gives a list of the main historic milestones (source: http://www.can-cia.org) in the development of the CAN bus. A brief history of the CAN is described in this section.

Table 2.1 Historic CAN milestones

1983:	Start of the Bosch internal project to develop an in-vehicle network
1986:	Official introduction of CAN protocol
1987:	First CAN controller chips from Intel and Philips Semiconductors
1991:	Bosch's CAN specification 2.0 published
1991:	CAN Kingdom CAN-based higher-layer protocol introduced by Kvaser
1992:	CAN in Automation (CiA) international users and manufacturers group established
1992:	CAN Application Layer (CAL) protocol published by CiA
1992:	First cars from Mercedes-Benz used CAN network
1993:	ISO 11898 standard published
1994:	1st international CAN Conference (iCC) organized by CiA
1994:	DeviceNet protocol introduction by Allen-Bradley
1995:	ISO 11898 amendment (extended frame format) published
1995:	CANopen protocol published by CiA
2000:	Development of the time-triggered communication protocol for CAN (TTCAN)

After several years of internal research by the engineers at Bosch, they officially announced the CAN protocol and demonstrated its superiority to the existing serial bus systems in 1986, at the SAE (Society of Automotive Engineers) congress in Detroit, Michigan. Later in the next year, the collaboration of several German Universities, the vehicle manufacturer Mercedes-Benz, and the semiconductor manufacturer Intel produced the first CAN controller chip Intel 82526, followed shortly

thereafter by the CAN chip 82C200 manufactured by Philips Semiconductors. Since then, many semiconductor companies have been manufacturing CAN controller chips and the CAN protocol has been accepted almost by every car manufacturer in Europe, and have been used in hundreds of passenger cars.

In the year 1992, the users and manufacturers of the CAN bus jointly formed the non-profit making organization CiA (CAN-in-Automation). The aim of this organization has been to promote the interests of its members by providing technical, product, and marketing information. The CiA is registered in Nuremberg (Germany) and over 500 companies are members of this organization. CiA organizes joint marketing activities including stands at fairs and exhibitions and joint seminars. To get access to the members only area of the CiA and to become a member one has to register by filling an application form at their web site (http://www.can-cia.org). The membership is to be renewed every year and as a CiA member the following benefits are offered (see the CiA site for more details):

- Initiate and influence specifications, which will be published by CiA
- Receive access to most current CiA work drafts and CiA draft standard proposals (even if not participating in the standardization)
- Be assigned free-of-charge CANopen vendor-IDs
- Receive exclusive information on new CAN technology and market trends
- Participate in joint marketing activities
- Develop partnerships with other CiA members
- Get credits on CiA events such as the international CAN Conference (iCC), CAN schools, etc.
- Get credits on some CiA publications
- Get credits on CANopen product certifications

The CAN protocol is protected by patents held by Robert Bosch GmbH. Licenses are normally granted to research organizations such as universities and manufacturers such as automobile and chip manufacturers. Bosch holds patents on the technology, and manufacturers of CAN-compatible microprocessors pay license fees to Bosch, which are normally passed on to the customer in the price of the chip.

The first production application of the CAN bus was in 1992 on several upper-class Mercedes-Benz passenger cars. Today, we see the CAN bus on all new models of this manufacturer. After Mercedes-Benz, automobile manufacturers Volvo, Saab, Volkswagen and BMW, now also Renault and Fiat use CAN networks in their vehicles. Far Eastern semiconductor vendors have also offered CAN controllers since the late 1990s. One of the first tasks of the CiA was the specification of the CAL (CAN Application Layer). This was necessary because CAN is a pure data link layer implementation and there were no standards to show how data could be exchanged using this protocol. Although the CAL approach was academically correct and usable in industrial applications, it was a true application layer where every user has to design a new communication mechanism.

The ISO (International Standards Organization) published the CAN standard in 1993 (http://www.iso.org) with the aim of defining an international standard where all the manufacturers can follow and produce compatible products. The standard, named ISO 11898:1993 entitled "Road vehicles – Interchange of digital information – Controller area network (CAN) for high speed communication" describes the general architecture of CAN in terms of hierarchical layers. The document contains detailed specifications of aspects of CAN belonging to the physical layer and the data link layer. The ISO 11898:1993 standard was later revised in 2003 and also in subsequent years. The complete CAN specification can be downloaded from the ISO's web site (http://www.iso.org) after paying the appropriate fee. These documents basically consist of the following parts:

- Data link layer
- Physical signalling
- High-speed medium access unit
- Low-speed fault tolerance interface
- Time triggered communication

The CiA organizes international annual conferences where experts from all over the world and from the most diversified application areas have met for years at this international event. The conference is unique in its target group and offers visitors the possibilities to become acquainted with the latest developments in CAN. Lectures are also offered in the conferences on various topics of the CAN bus. The first international CAN conference (iCC) was held in 1994.

In the year 1994, the American company Allen-Bradley (now owned by Rockwell Automation) developed the DeviceNet protocol. DeviceNet is a network system used in the automation industry to interconnect control devices for data exchange. It is based on the CAN bus as the backbone technology and defines an application layer to cover a range of device profiles. DeviceNet is an open standard, managed by the ODVA (Open DeviceNet Vendors Association) in North America. DeviceNet is mainly used in industrial automation, in programmable controllers, and to interconnect computers and industrial controllers.

In the year 1995, the CANopen proocol has been published as a higher level protocol for embedded systems. CANopen protocol has been developed not only for the automobile industry but for a very broad range of applications, including machine control, medical applications, maritime electronics, building automation, and so on. CANopen was developed and is currently being maintained by the CiA user group.

In the beginning of 2000, an ISO task force involving several companies defined an extension to the existing CAN protocol for a time-triggered transmission of CAN messages. Some Bosch employees, together with experts from the semiconductor industry and from academic research defined this new protocol as the "Time-triggered communication on CAN (or TTCAN)". This CAN extension will allow the time-equidistant transmission of messages and the implementation of closed loop control

via CAN. It will also enable the use of CAN in "by-wire" applications. Because the CAN protocol has not been altered, it is possible to transmit time-triggered as well as event-triggered messages via the same physical bus system.

2.1 CAN In Automotive Industry

Today, more than 20 semiconductor manufacturers produce devices with CAN interfaces and almost every new passenger car manufactured in Europe and Far East are equipped with at least one CAN network. CAN is one of the most dominating bus protocol used in passenger cars. Figure 2.1 shows the CAN usage in the world (source: http://www.can-cia.org) since the year 2000 where there were approximately 120 million CAN devices used in the world. By the year 2007 this number has increased to 800 million devices.

CAN and CAN based higher level protocols such as CANopen and DeviceNet are becoming more widely accepted by the North American manufacturers and it is expected that these protocols will have very high penetrations in most of the industrial and commercial automation markets. In particular, CAN will be used in the following diverse applications within the next five to ten years:

- Passenger cars
- Buses
- Trains
- Maritime electronics
- Aviation electronics
- Factory automation
- Lifts
- Medical equipment
- Programmable machine controllers
- Home entertainment systems
- Domestic appliances
- Military applications
- Space applications

An interesting and important application of the CAN bus will certainly be in the medical industry. Medical equipment and devices such as X-ray machines, ultrasound, radiotheraphy machines, CAT scanners, MRI machines and so on could be designed and developed to make use of the CAN bus for their internal communication structures. This approach will simplify the design of the machine as well as increase its safety and reliability, and also help to improve the maintenance.

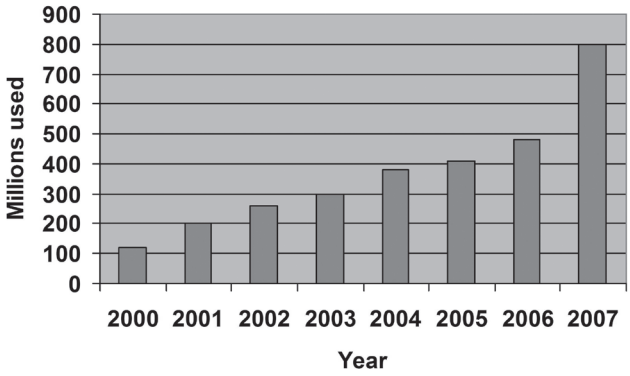


Figure 2.1 CAN usage over time

2.2 The Basic Structure of a CAN Automotive System

It is worthwhile to have a look at the basic structure of a CAN based automotive system before going into the theory and details of the CAN protocol and CAN electronics.

Before the development of the CAN bus the wiring of the automotive electronics was very complicated. There were hundreds of wires connected from one unit to another one in a complicated way. The wiring and maintenance of such a system was extremely complex and it was very difficult to locate and correct an error. In addition, there were many local controllers used to control various units of the car in an isolated manner. For example, isolated controllers were used to control the brake system or the lighting system. There was no co-ordination between the various controllers, the reliability and safety were poor, and as such it was hard to maintain the overall system.

With the development of the CAN bus there has been major advances both in reliability and safety of automotive electronics. As shown in Figure 2.2, in a CAN bus based automobile all of the units are connected to each other over a two-wire bus system. As we shall see in later chapters in full detail, the two wires of the bus are named as CAN_LO and CAN_HI. There is full co-ordination between all the units and the ECU (Engine Control Unit) is responsible to control each unit and to make sure that all the units operate as expected. In Figure 2.2, the units at the upper part of the figure are time-critical units requiring high priority. Similarly, the units at the lower part of the figure are slow speed non-critical units requiring lower priorities.

The complete wiring and the state of the vehicle can be monitored and interrogated by connecting a diagnostic tool through the diagnostic connector located on the engine. This tool can be a laptop running special software (see Figure 2.3) developed by the manufacturer, or a dedicated hand-held

diagnostic device (see Figure 2.4 and Figure 2.5) can be used to communicate directly with the ECU. Using this tool and with the help of the ECU it is possible to get information about the state of each unit of the vehicle. For example, we can easily interrogate the Airbag Module and check that it is working properly.

CAN is a two-wire bus system (see Figure 2.6) with resistive connectors at each end of the bus, as shown in Figure 2.7. All the units (or nodes, or CAN-stations) are connected rigidly to the bus using CAN connectors (e.g. T-connectors, End-connectors and so on). Figure 2.8 shows a typical section of the BUS where several units are connected and can communicate over the bus. The trunk cable runs along and sensors are connected to this cable via the drop cables.

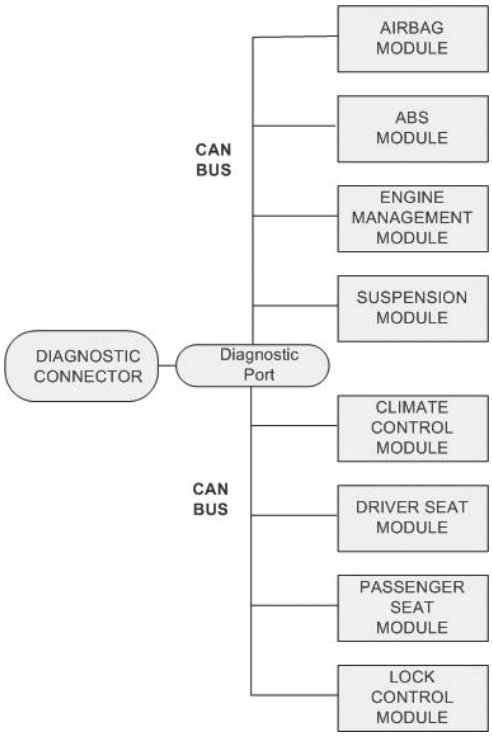


Figure 2.2 CAN Bus based automotive wiring system

In general, information is received from the various sensors (e.g. engine temperature sensor) located at different parts of the vehicle. As shown in Figure 2.9, the sensor information is then passed to a microcontroller, and then to the CAN bus via a CAN controller and a CAN transceiver. The function of the CAN transceiver is to make physical connection to the actual CAN bus for communication over the bus. The CAN controller is under the control of the microcontroller and performs the CAN protocol specific functions.

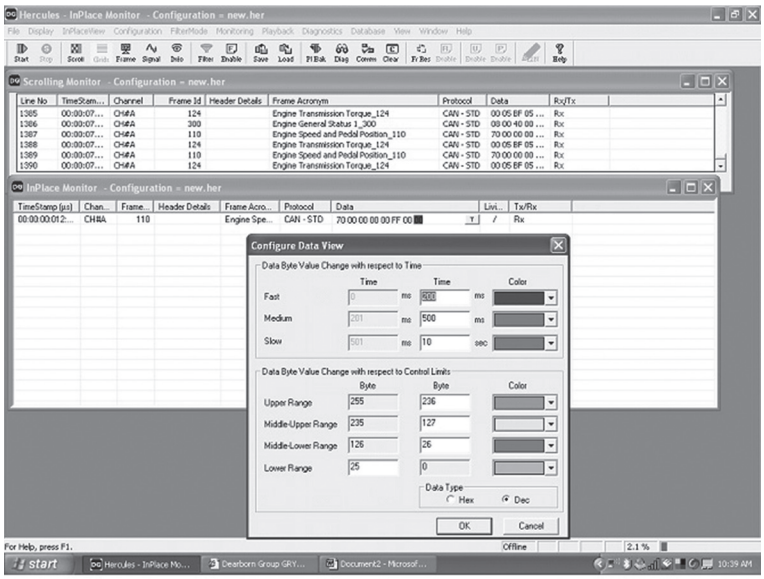


Figure 2.3 CAN Bus software protocol analyzer (Model: Hercules)

Figure 2.4 A typical CAN Bus diagnostic tool (Model: OmniScan)

Figure 2.5 Another CAN Bus diagnostic tool (Model: VAG5053)

Figure 2.6 CAN is a two-wire bus

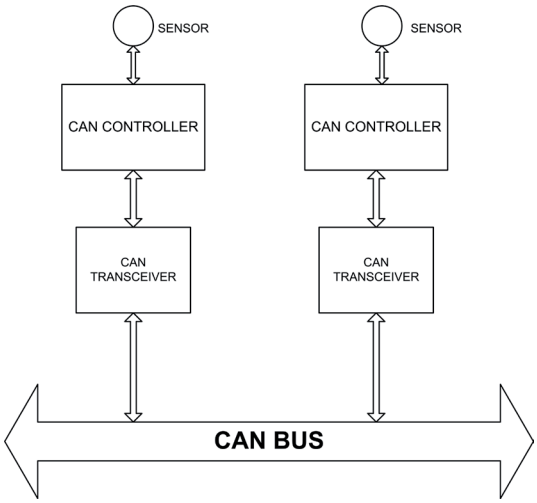


Figure 2.7 A typical two node CAN bus

A typical ECU is shown in Figure 2.10. This ECU (Model: ECU MS3 Sport, Source: Bosch Motorsport, Equipment For High Performance Vehicles, Edition 2011/11) is suitable for car engines up to 6 cylinders and has the following electrical characteristics (see the manufacturers' manual for more information):

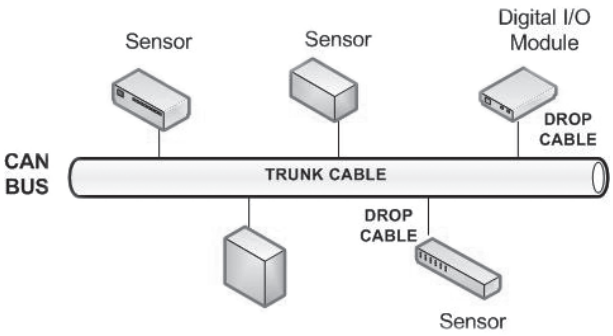


Figure 2.8 Section of a CAN Bus

- Max power consumption 10W (at 14V)

INPUTS:
- 2 lambda sensor interfaces LSU
- 4 inputs for Hall Effect wheel speed sensors
- 1 input for inductive crankshaft sensor
- 1 input for Hall Effect camshaft sensor
- 17 analog inputs (0 to 5V)
- 2 knock sensor inputs
- 6 digital inputs

OUTPUTS:
- 8 injection power stages
- 8 ignition power stages
- 16 power stages
- 2 power stages for lambda heater
- 1 H-bridge (5A)
- 2 sensor supply

Figure 2.11 shows diagramatically how the ECU (in the center of the figure) can be connected to other parts of a car. The port labelled "Diagnosis" is used for OBD (On Board Diagnostics) where a hand-held terminal (a scantool) or a laptop computer can be connected using a special connector and the health of the overall vehicle can easily be examined.

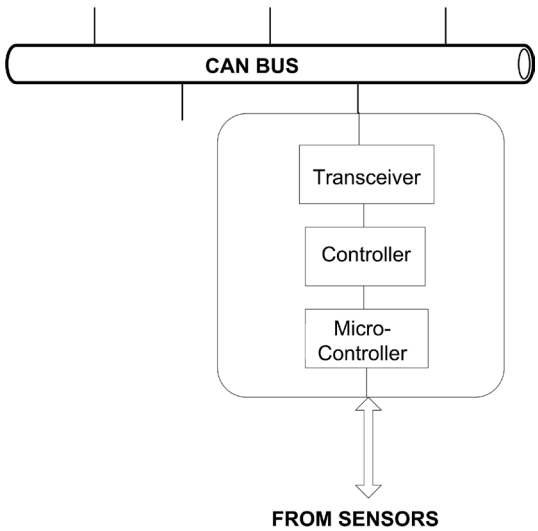


Figure 2.9 A typical CAN Bus sensor interface

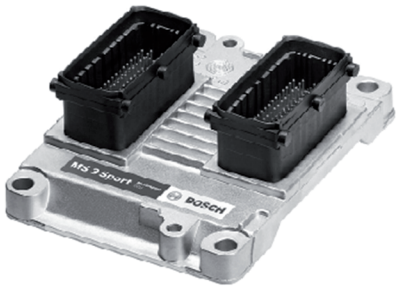


Figure 2.10 ECU MS3 Sport

2.3 Advantages of the CAN Bus

The main benefits of the CAN bus can be summarized as follows:

- Low wiring complexity
- Easy to manage twisted-pair wire bus
- New nodes can easily be added and removed
- Breakdown of a node does not affect the other nodes in the system
- Centralized control
- All devices on the bus can read the same message
- Fail-safe against electromagnetic radiation
- Deals effectively with errors
- Easy diagnostic and maintenance

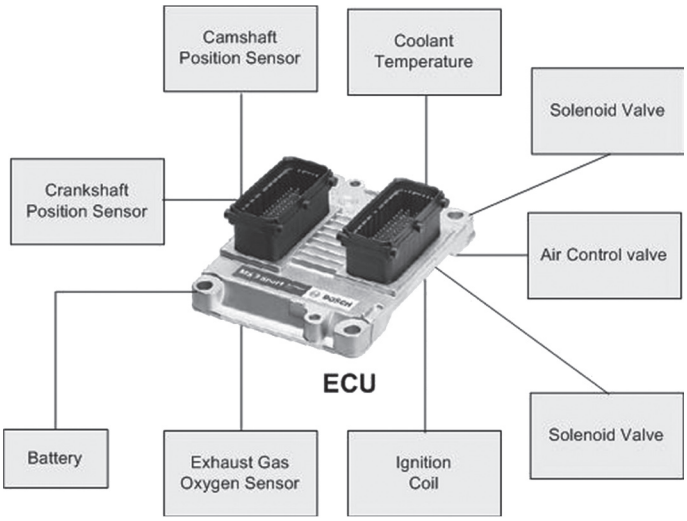


Figure 2.11 ECU and other parts of a car

2.4 Disadvantages of the CAN Bus

Some of the disadvantages of the CAN Bus are given below:

- The data rate can be low in some applications (limited to 1Mbps)
- Expensive to implement as special controllers are required
- Complete system shutdown if the main CAN truck wire is damaged

2.5 Properties of the CAN Bus

The CAN is bus is currently used in many diverse fields, from passenger cars to marine electronics, medical electronics, aviation electronics and so on. The reason for the wide acceptance of the CAN bus is because of its high performance, reliability, robustness, and safety.

The main properties of the CAN bus can be summarized as below (the details of these properties will be examined in the next chapters):

- Two-wire twisted-cable bus
- Bus terminated with resistors at each end
- Up to 1Mbps speed on the bus (for 40m length bus). Higher data rates for shorter bus lengths
- Transfer of up to 8 data bytes at any time
- Messages sent and received as frames under a clearly defined robust protocol
- Multi-master priority based access to the bus
- Any one node can transmit while other nodes listen
- Bus arbitration if two or more nodes attempt to transmit at the same time
- Broadcast type message transfer where all nodes can receive the same message
- No node addresses. Nodes accept or reject data on the bus based on message acceptance filtering
- Remote data request where a node can request data from another node
- Error detection and signalling
- Automatic retransmission of messages if failed to transmit (because of the bus arbitration)
- Automatic deactivation of nodes that present consistent errors

2.6 Summary

This chapter has made an introduction to the CAN bus. The history of the CAN bus was given briefly with the major milestones in its development and success to date. The basic features of the CAN bus are described with reference to its use in passenger cars. An example ECU was given with its electrical specifications. Finally, the main advantages and disadvantages of the CAN bus and its properties are given towards the end of the chapter.

Chapter 3 The ISO/OSI reference model and CAN

The *OSI* (*Open Systems Interconnection model*) *model* is a way of sub-dividing a communications system into smaller manageable parts called layers. Basically, a layer is a collection of similar functions that receives services from the layer below and provides services to the layer above. The OSI model is developed by the *ISO* (*International Organization for Standardization*) and it is a 7-layer reference model as shown in Figure 3.1. The OSI reference model is now considered as a primary standard for internetworking and inter computing. Today many network communication protocols are based on the standards of this model.

The **Physical Layer** defines the physical and electrical specifications of the devices. i.e. it represents the actual hardware and the physical connection between the nodes in the network and the electrical characteristics of the actual voltages and currents involved in the hardware. In particular, it defines the relationships between a device and its transmission medium, such as copper, twisted-cable, coaxial cable, or optical cable. This layer also includes the type of connectors used, such as round, D-type, etc, the layout of pins, voltage and current levels (e.g. how many volts should represent logic 0 and logic 1), and the timing features of the signals (e.g. how many seconds or microseconds should a bit last). Some of the standards that deal with the physical layer are: RS232, RS485, Centronics, X.21, I2C, 1-Wire, IEEE 802.3, and so on.

The **Data Link** Layer provides delivery of information packets between communicating devices. This layer is responsible for sending, receiving, and validating the data. The Data Link Layer assembles bits into frames with the correct error checking bits, making them ready for transmission over the network, through the physical layer. It provides error detection (and sometimes correction) of the transmitted and received data. Encryption can be used to protect the data to be sent between the nodes. The receiving node then decrypts the message. Examples of Data Link Protocols are ARP, ATM, SLIP, PLIP, HDLC, and so on.

The **Network Layer** handles the routing of the data, i.e. sending it in the right direction to the right destination on outgoing transmissions and receiving incoming transmissions. This layer is responsible for routing and forwarding message packets. The decision for the best route is made and the packet is sent accordingly. The layer uses logical addressing for routing a packet. Note that the physical address (like MAC address) keeps changing from hop to hop when a packet travels from source to destination. The logical address doesn't change and it provides continuity between hops. The Network Layer receives data from the Transport Layer and provides data to the Data Link Layer. Routers operate at this layer. Examples of Network Layer protocol are ICMP, ARP, RARP, DLC, IPX, IPsec, and so on.

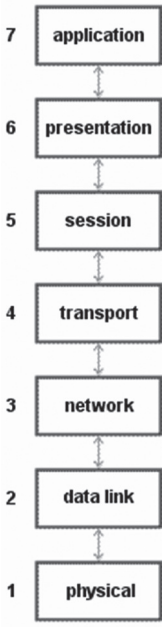


Figure 3.1 The ISO/OSI reference model

The **Transport Layer** provides error checking and end to end message delivery, thus ensuring successful delivery of packets across the network. This layer also provides the acknowledgement of the sent packets and re-transmits the packet if error occur. The Transport Layer manages connection orientated (e.g. TCP) and connectionless (e.g. UDP) packet transfers. Examples of Transport Layer protocols are TCP, UDP, SPX, NETBIOS, ATP, and so on.

The **Session Layer** establishes and manages the session between the two nodes at different ends in a network. This layer sets up (i.e. opens), coordinates, and terminates (i.e. closes) conversations, and exchanges data between the applications at two nodes. Session layer also manages which node can transfer the data in a certain amount of time and for how long, and provides synchronization between communicating computers. Examples of Session Layer protocols are FTP, SNMP, Mail Slots, Names Pipes, RPC, PPTP, and so on.

The **Presentation Layer** presents the data into a uniform format and masks the difference of data format between two dissimilar systems. This layer is usually part of the operating system and converts incoming and outgoing data from one presentation format to another (i.e. it is responsible for protocol conversion). The Presentation Layer is also responsible for the protocol conversion, encryption, decryption and data compression. Examples of Presentation Layer protocols are MIME, TLS, SSL and so on.

The Application Layer is the OSI layer closest to the end user, which means that both the OSI application layer and the user interact directly with the software application. This layer provides a means for the user to access information on the network through an application and the appropriate software libraries (e.g. API). Many user applications that need to communicate over the network interact with the Application layer protocol directly. Examples of Application Layer protocols are Telnet, FTP, SMTP, and so on.

Figure 3.2 shows that the message size gets bigger at every layer of the ISO/OSI reference model.

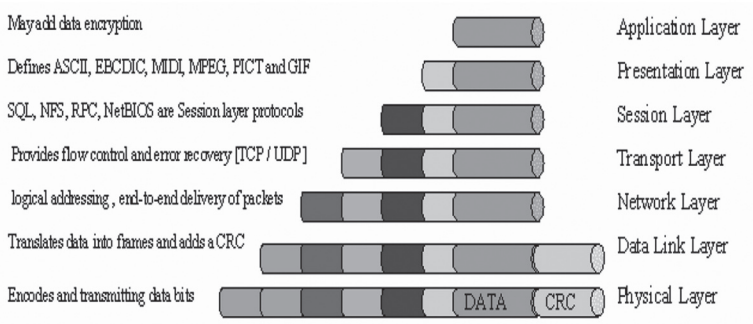


Figure 3.2 The message gets bigger as more data is added at each layer

3.1 CAN Bus ISO/OSI Model

The CAN implementation only uses the Physical Layer and the Data Link Layer of the ISO/OSI reference model, thus bypassing the connection between the Data Link Layer and the Application Layer. This is so that the overhead and the memory usage can be minimized, thus providing higher performance.

ISO/OSI layers 3 (Network Layer) to 7 (Application Layer) are implemented in higher level CAN based protocols such as the DeviceNet and CANopen, which require more resources for their implementation. The higher level protocols are used because they provide network management functions (e.g. node monitoring, node synchronization and so on). In addition, the higher level protocols provide a communications model where all the nodes can follow for compatibility and ease of large and secure data transfer (e.g. the File Transfer Protocol, FTP), and finally the higher level protocols allow larger data bytes to be sent with each message as the standard CAN allows only 8 data bytes to be sent by each message.

The CAN Data Link Layer and most of the Physical Layer are normally implemented on the CAN controller chip (i.e. in silicon) and thus the user does not need to worry about them. This speeds up the development time as the user concentrates only on developing the actual applications code.

Figure 3.3 shows the ISO/OSI model and the layers used by CAN. As can be seen from the figure, the CAN protocol includes the Data Link Layer and the Physical Layer of the reference model.

The Data Link Layer consists of the Logical Link Control (LLC) and Medium Access Control (MAC). LLC manages the overload notification, acceptance filtering, and recovery management. MAC manages the data encapsulation, frame coding, error detection, and serialization/de-serialization of the data. The Physical Layer consists of the Physical Signalling Layer (PSL), Physical Medium Attachment (PMA), and the Medium Dependent Interface (MDI). PSL manages the bit encoding/decoding and bit timing. PMA manages the driver/receiver characteristics, and the MDI is the actual connections and wires.

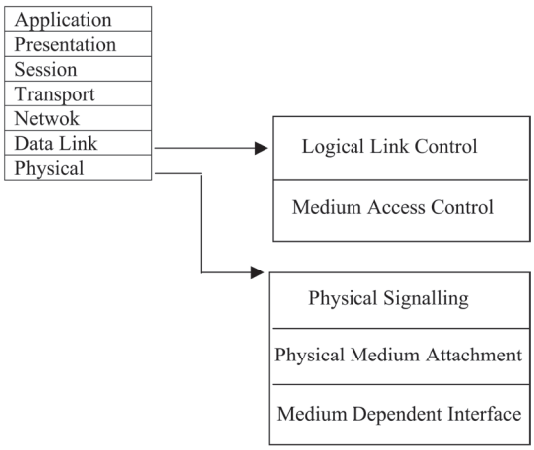


Figure 3.3 CAN and the ISO/OSI Model

3.2 Summary

Many network protocols and communication systems are described using the 7-layer ISO/OSI reference model. A layer receives data from an upper level and transfers this data to a lower level after performing the required formatting operations. This chapter has given the basic structure of the ISO/OSI reference model and also described the tasks performed at each layer.

The CAN bus implements only the lower two layers of the ISO/OSI reference model. The Data Link Layer and most of the Physical Layer are normally implemented in silicon on the CAN controller chip. This makes the development of CAN bus based projects easier.

Chapter 4 CAN physical layer

It is appropriate at this stage of the book to look at the CAN Physical Layer as this layer forms the hardware interface to the actual CAN bus and we can get a better feeling of how various devices can be connected to the CAN bus.

Figure 4.1 shows a CAN bus with three nodes. The bus is made up of a twisted-pair cable and is terminated at both ends with resistors so that the bus has characteristic resistance of 120 ohms. The two wires of the bus are termed as CAN_H and CAN_L.

4.1 CAN Bus Termination

The bus is terminated to minimize signal reflections on the bus. Although usually a single 120 ohm resistor is connected to each end of the bus, in general one of the following methods can be used to terminate the bus:

- Standard termination
- Split termination
- Biased split termination

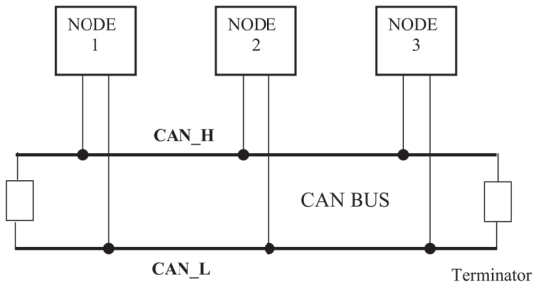


Figure 4.1 CAN bus with three nodes

The most commonly used termination method is the standard termination where a 120 ohm resistor is used at each end of the bus as shown in Figure 4.2.

Figure 4.3 shows the split termination which is gaining popularity. In this method two 60 ohm resistors and a capacitor are used at each end of the bus. The advantage of this method is that it allows for reduced emission.

Figure 4.4 shows the biased split bus termination where a voltage divider circuit and a capacitor are used at each end of the bus. As in the split termination, the biased split termination increases the EMC performance of the bus.

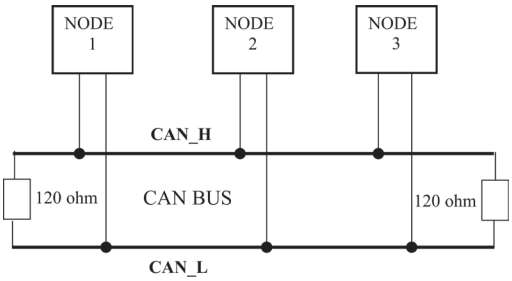


Figure 4.2 *Standard* bus termination

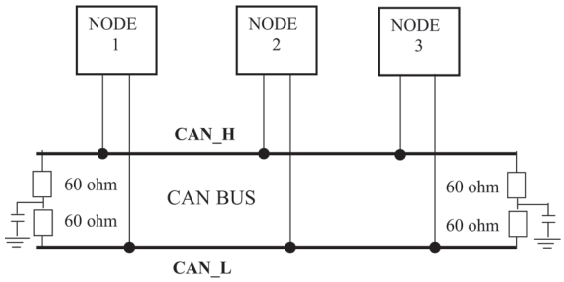


Figure 4.3 *Split* bus termination

4.2 CAN Bus Data Rate

The ISO-11898 CAN specifies that a device on the bus must be able to drive a 40m cable at 1Mb/s. In practise, a much higher bus length is achieved by lowering the bus speed. Table 4.1 shows the bus speed against bus length and the nominal bit time. At 1000kbps (1µs nominal bit time) the maximum allowed bus length is 40m, whereas at 10kbps (100µs nominal bit time) the maximum allowable bus length is increased to 6700m.

A graph of the maximum data rate against the maximum allowed bus length is shown in Figure 4.5.

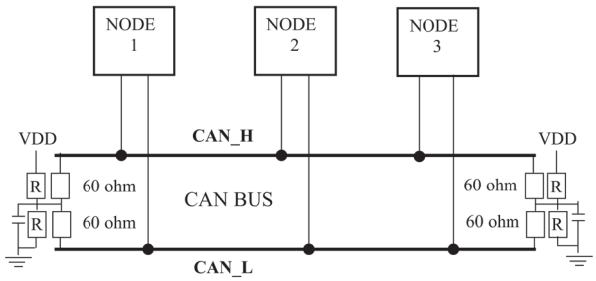


Figure 4.4 Biased split bus termination

Table 4.1 Data rate against nominal bit time and maximum bus length

Data rate (kbps)	Nominal bit time (µs)	Bus length (meters)
10	100	6700
20	50	3300
50	20	1300
125	8	530
250	4	270
500	2	130
1000	1	49

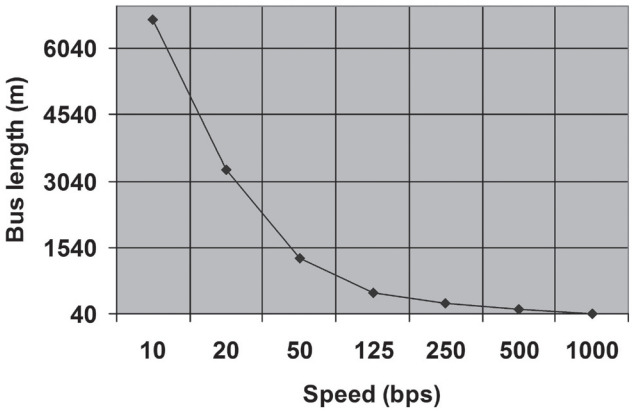


Figure 4.5 CAN bus data rate and bus length

4.3 Cable Stub Length

In high data rate applications the length of the cable stubs and the distance between the nodes become important. At the maximum data rate of 1Mbps, the length of the cable stubs (see Figure 4.6) should not be greater than 0.3m, and the maximum node distance should be 40m.

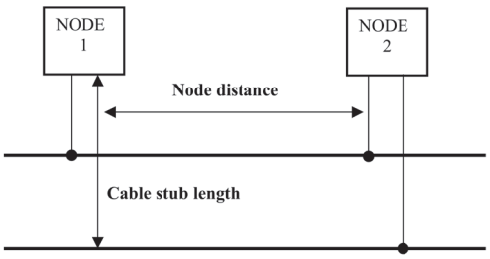


Figure 4.6 Cable stubs and node distances

4.4 CAN Bus Signal Levels

The data on CAN bus is differential and the bus specifies two logical states: dominant and recessive. Figure 4.7 shows the state of signals on the bus (see document ISO-11898-4 for more details).

The recessive state is logic "1" and at this state the differential voltage on the bus (i.e. Vdiff = CAN_H – CAN_L) is ideally 0V (ideally CAN_H = CAN_L = 2.5V). In practise the recessive differential output voltage is less than 0.05V at a bus transmitter output device.

The dominant state is logic "0" and at this state the differential voltage on the bus (i.e. Vdiff = CAN_H – CAN_L) is ideally 2V (ideally CAN_H = 3.5V and CAN_L = 1.5V). In practise the dominant differential output voltage is between 1.5V and 3.0V.

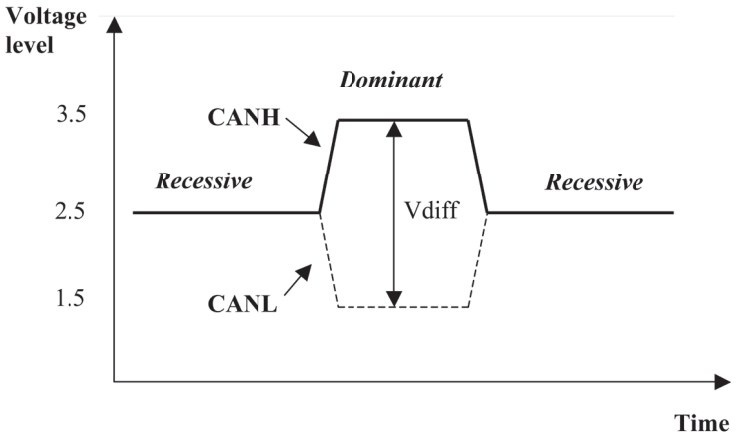


Figure 4.7 CAN bus signal levels

When several nodes on the bus attempt to transmit at the same time, a bus arbitration logic is used to grant access to the bus. When there is arbitration on the bus, a dominant bit state always wins out over a recessive bit state.

The output of a CAN transceiver circuit is usually in open-collector (e.g. TTL logic) or in open-drain (e.g. CMOS logic) format. When several such devices are connected to a bus, the net logic state of the bus is defined by the logical "AND" of the device outputs (also called the "Wired AND"). For example, if three devices are connected to the bus, the state of the bus will be logic "1" if and only if all the outputs of all the three devices are at logic "1", otherwise the bus will be at logic "0".

Figure 4.8 shows the output stage of a typical CAN transceiver (source: MCP2551, Microchip Technology Inc.)

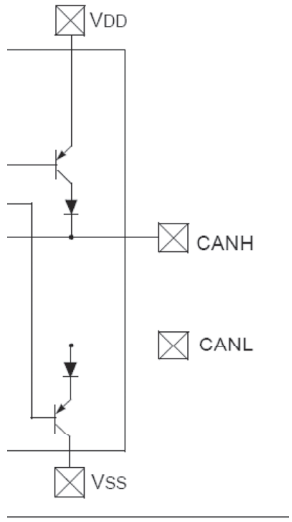
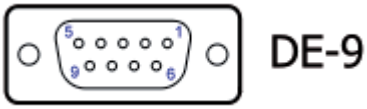


Figure 4.8 Output stage of a typical CAN bus transceiver

4.5 CAN Connectors

Even though CAN is a two-wire network, in many cases a power signal and a ground signal are added to the standard CAN connectors. The actual wires used for the bus can either be unshielded twisted-pair (UTP), or shielded twisted-pair (STP). Shielded cables should be used in electrically noisy environments and when long bus cables are needed.

The standard CAN connector is a 9-pin D-type connector (DE-9) as shown in Figure 4.9. Pin 2 and pin 7 are the CAN_L and CAN_H signals respectively. Pin 3 and pin 9 are used as the signal ground and signal power respectively. The signal ground, and signal power pins can be useful when it is required to power remote devices. Care should however be exercised to make sure that the current capacity of the used cable is not exceeded.



9 Pin (male) D-Sub CAN Bus PinOut		
Pin #	**Signal Names**	**Signal Description**
1	Reserved	Upgrade Path
2	CAN_L	Dominant Low
3	CAN_GND	Ground
4	Reserved	Upgrade Path
5	CAN_SHLD	Shield, Optional
6	GND	Ground, Optional
7	CAN_H	Dominant High
8	Reserved	Upgrade Path
9	CAN_V+	Power, Optional

Figure 4.9 CAN D-connector pin configuration

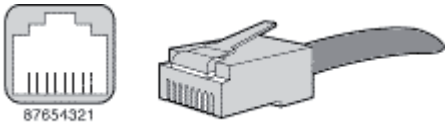
Some companies use a 10-pin header to make connections to the bus. Figure 4.10 gives the standard pin configuration for this kind of connectors.

It is also common to use either RJ10 or RJ45 type connectors in CAN bus applications. Figure 4.11 gives the pin configuration for these type of connectors.

Some companies prefer to use a 5-pin circular connector. Figure 4.12 gives the pin configuration for these connectors.

10-Pin Header CAN Bus PinOut		
Pin #	**Signal Names**	**Signal Description**
1	Reserved	Upgrade Path
2	GND	Ground, Optional
3	CAN_L	Dominant Low
4	CAN_H	Dominant High
5	CAN_GND	Ground
6	Reserved	Upgrade Path
7	Reserved	Upgrade Path
8	CAN_V+	Power, Optional
6	Reserved	Upgrade Path
7	Reserved	Upgrade Path

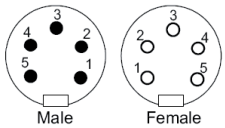
Figure 4.10 CAN 10-pin header pin configuration



RJ-45 connector

RJ10, RJ45 CAN Bus PinOut			
RJ45 Pin #	**RJ10 Pin #**	**Signal Name**	**Signal Description**
1	2	CAN_H	Dominant High
2	3	CAN_L	Dominant Low
3	4	CAN_GND	Ground
4	-	Reserved	Upgrade Path
5	-	Reserved	Upgrade Path
6	-	CAN_SHLD	CAN Shield, Optional
7	-	CAN_GND	Ground
8	1	CAN_V+	Power, Optional

Figure 4.11 CAN RJ10 and RJ45 pin configurations



5-Pin Micro/Mini CAN Bus PinOut		
Pin #	**Signal Names**	**Signal Description**
1	CAN_SHLD	Shield, Optional
2	CAN_V+	Power, Optional
3	CAN_GND	Ground
4	CAN_H	Dominant High
5	CAN_L	Dominant Low

Figure 4.12 CAN 5-pin circular connector

4.6 Summary

This chapter has described the CAN Physical Layer in detail. Various CAN bus termination methods have been explained. In addition, CAN bus voltage levels, and CAN logic states have been described. The pin configurations of the commonly used CAN bus connectors are also given in the chapter.

Chapter 5 CAN bus frames

Messages on the CAN bus are sent and received using *Frames*. A frame is basically like a *Packet* in a TCP/IP type network where the actual data is encapsulated with control data.

CAN bus communication is not like the popular Client-Master type communication. In CAN bus all nodes have the same rights and they can transmit as well as receive data at suitable times.

Some of the important CAN protocol features are:

- CAN bus is multimaster. When the bus is free, any device attached to the bus can start sending a message. When multiple devices attempt to send data at the same time then collision can occur on the bus. Collisions are detected and avoided using an arbitration mechanism.

- CAN bus protocol is flexible. The devices connected to the bus have no addresses (or node IDs), which means that messages are not transmitted from one node to another one based on addresses. Instead, all nodes on the bus receive every message transmitted on the bus, and it is up to each node to decide whether the received message should be kept or discarded. A single message can be destined for a particular device on a particular node, or for many nodes, depending on how the bus system is designed. Messages have message identifiers, and acceptance filters on each node decide whether or not to accept a message being transmitted on the bus. Another advantage of having no addresses is that when a device is added to or removed from the bus, no configuration data needs to be changed (i.e. the bus is "hot pluggable").

- Messages sent on the bus have priorities. A message with a lower message identifier has a higher priority.

- CAN bus communication speed is not fixed. Any communication speed up to the allowed maximum can be set for the devices attached to the bus.

- CAN bus offers remote transmit request (RTR), which means that a node on the bus is able to request data from the other nodes. Thus, instead of waiting for a node to continuously send data, a request for data can be sent to the node. For example, in a vehicle, where the engine temperature is an important parameter, the system can be designed so that the temperature is sent periodically over the bus. However, a more elegant solution is to request the temperature as needed. This second approach will minimize the bus traffic and increase the performance, while maintaining the integrity.

- All devices on the bus can detect an error. The device that has detected an error immediately notifies all other devices. Nodes that transmit faulty data, or nodes that are always receiving data in error will remove themselves from the bus, thus allowing the normal bus operations to continue.

- Receiving nodes on the bus check the validity of the received frame and acknowledge the consistency. The transmitting node monitors the bus during the acknowledgement slot.

- Multiple devices can be connected to the bus at the same time, and there are no logical limits to the number of devices that can be connected. In practice, the number of nodes that can be attached to a bus is limited by the bus's delay time and electrical load on the bus.

There are four message frames in CAN:

- **Data Frame**: Defines the data transfer between nodes.

- **Remote Frame**: Used by a node to request transmission of a message (i.e. data) from another node.

- **Error Frame**: Any node on the bus can send an error frame to signal an error condition.

- **Overload Frame**: This frame is used by a receiving node to indicate that it is not yet ready to receive frames.

We shall now look at each frame in greater detail.

There are basically two types of CAN protocols: 2.0A and 2.0B. CAN 2.0A is the earlier standard with 11 bits of identifier (see next section), while CAN 2.0B is the new extended protocol with 29 bits of identifier. 2.0B controllers are completely backward compatible with the 2.0A controllers and can receive and transmit messages in either format.

We shall firstly look at the standard CAN 2.0A frames and then investigate the CAN 2.0B frames in later sections.

There are two types of 2.0A controllers. The first is capable of sending and receiving 2.0A messages only, and reception of a 2.0B message will flag an error. The second type of 2.0A controller (known as 2.0B passive) sends and receives 2.0A messages but will also acknowledge receipt of 2.0B messages and then ignore them

5.1 Data Frame

The data frame is used by the transmitting device to send data to receiving devices on the bus, and the data frame is the most important frame handled by the user. The data frame can be sent in response to a request, or it can be sent whenever it is required to send the value of some parameter to other nodes on the bus (e.g. the temperature can be sent at periodic intervals).

Figure 5.1 shows the structure of a data frame. The bus is normally idle. Then, a standard data frame starts with the *start of frame* (SOF) bit, which is followed by an 11-bit *identifier* and the *remote transmission request* (RTR) bit. The *control* field is 6-bits wide and indicates how many bytes of data are in the data field. The *data* field can be 0 to 8 bytes and it contains the actual data to be sent. The data field is followed by the 16-bit checksum (*CRC)* field which checks whether or not the received bit sequence is corrupted. The *ACK* field is 2-bits wide and is used by the transmitting node to receive acknowledgement of a valid frame from any receiver. The end of message is indicated by a 7-bit *end of frame* (EOF) field. Successive frames must be separated by at least 3-bit times, called the *interframe space (ITM)*.

The total number of bits required by the data frame are (assuming successive frames are to be sent):

SOF	1 bit
Identifier	11 bits
RTR	1 bit
Control	6 bits
Data	0 to 64 bits (0 to 8 bytes)
CRC	16 bits
ACK	2 bits
EOF	7 bits
ITM	3 bits

In total, 47 bits (no data) to 111 bits (8 bytes of data) are required by the data frame.

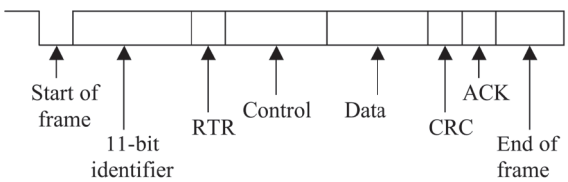


Figure 5.1 Standard data frame

Figure 5.2 shows the standard data frame in more detail.

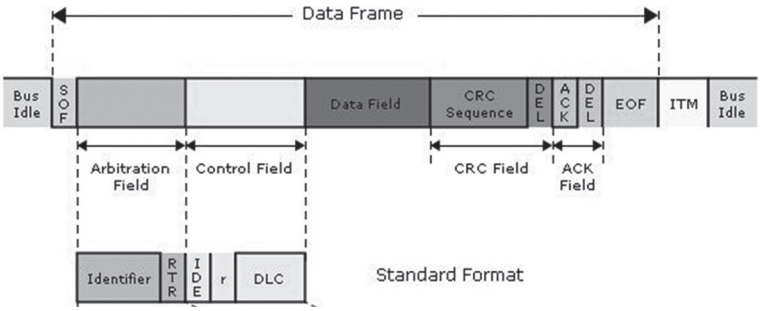


Figure 5.2 Detailed standard data frame

The fields of the data frame are explained beow in more detail.

5.1.1 Start Of Frame (SOF)

The start of frame field is 1-bit and it indicates the beginning of a data frame, sent while the bus is in idle state. The bus is said to be in idle state if a sequence of 11 recessive bits are present on the bus (comprising 1-bit ACK delimeter, 7-bits EOF, and 3-bits ITM). The SOF bit is in dominant state.

As we shall see later, the SOF field also starts the arbitration sequence on the bus when multiple devices attempt to send data at the same time.

5.1.2 Arbitration Field

The arbitration field of a standard data frame is 12-bits wide and it consists of the following two components:

- 11-bit Identifier
- 1-bit Remote transmission request (RTR)

The 11-bit identifier (MSB sent first) is used to identify messages on the bus. Different devices can send messages with different identifiers. For example, a temperature sensor device can send a message with an identifier of 20, while a pressure sensor can send a message with an identifier of 25. The receiving nodes have *Acceptance Filters* and by programming these filters we can accept or reject

messages with given identifier numbers. For example, if we program the acceptance filters of two nodes to accept messages with identifier numbers of 20, then whenever the temperature data is sent by the above temperature sensor node, our two nodes will accept and use this temperature data. With 11-bit identifier up to 2048 (useable 2032) unique identifier numbers can be set.

A data frame with a lower identifier has a higher message priority and as we shall see shortly, such a message is granted the bus access by the arbitration mechanism.

Arbitration is used to resolve bus conflicts that occur when several devices at once start sending messages on the bus. During the arbitration phase, each transmitting device transmits its identifier and compares it with the existing level on the bus. If the levels are equal, the device continues to transmit its identifier. If the device detects a dominant level on the bus while it is trying to transmit a recessive level, it quits transmitting and becomes a receiving device. After arbitration only one transmitter is left on the bus and this transmitter continues to send the remainder of its frame bits. The process of arbitration is illustrated in Figure 5.3 by an example consisting of three nodes with the following identifiers and with RTR = 0:

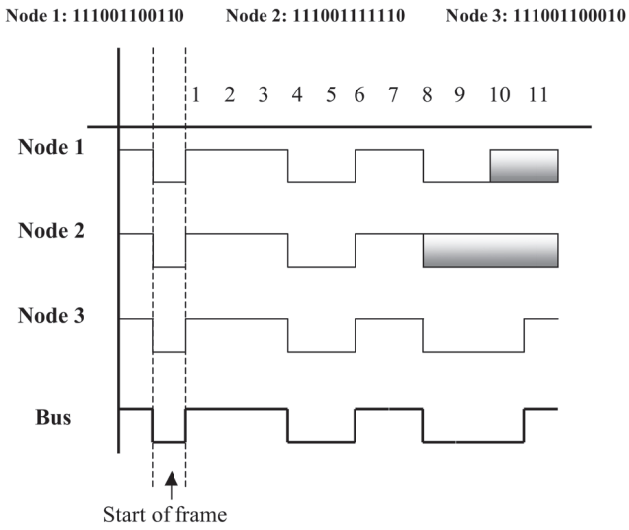


Figure 5.3 Example CAN bus arbitration

Reminding ourselves that the recessive level corresponds to 1 and the dominant level to 0, and that the dominant level has a higher precedence than the recessive level, the arbitration in Figure 5.3 is performed as follows:

- Bits are transmitted starting from the MSB. Numbering the MSB bit as bit 1:
- All nodes start transmitting simultaneously, first sending their SOF bits.
- Then they start sending their identifier bits. Up to the 8th bit all nodes send the same identifier bit. The 8th bit of Node 2 is in recessive state, while the corresponding bits of Nodes 1 and 3 are in dominant state. Therefore, Node 2 stops transmitting and returns to receive mode. The receiving mode is indicated in the figure by a gray colour field.
- The 9th bits of Nodes 1 and 3 are the same, so they continue sending their identifier bits.
- The 10th bit of Node 1 is in the recessive state, while the same bit of Node 3 is in dominant state. Thus, Node 1 stops sending and returns to receive mode (gray colour field).
- The bus is now left to Node 3, which can send the remainder of its identifier, and remainder of the frame bits (RTR bit, control field, data field, CRC, and the ACK bits).

If we look at the *identifiers* of each node again with their hexadecimal equivalents:

Node 1:	**11100110011**	**Node 2: 1110011111**	**Node 3: 11100110001**
Hex:	**733**	**73F**	**731**

The node with the smallest identifier number is Node 3, and thus this node has the highest priority. This is compatible with the example given above where only Node 3 is given transmit right on the bus.

Note that the nodes on the bus had no addresses in the above example. Instead, all the receiving nodes (Nodes 1 and 2 above) pick up all the data sent by Node 3, and they use their acceptance filters to determine whether or not they want to accept the data transmitted by Node 3.

It is important to note that, after Node 3 sends its data, the situation may become different, and for example, Node 2 can become the bus master by sending a data frame with a lower identifier. This is the beauty of the CAN bus, allowing a multi-master operation of the bus, enabling each node to become either a master or a client whenever they wish.

5.1.3 RTR Field

The 1-bit RTR field indicates the transmission of a data frame (RTR = 0), or a Remote Request Frame (RTR = 1). This field, together with the identifier bits form the arbitration field. In the example above, RTR = 0.

5.1.4 Control Field

The control field is 6-bits wide, and as shown in Figure 5.4, consists of the IDE (Identifier Extension) bit, a reserved bit (r0), and 3 DLC (Data Length Code) bits.

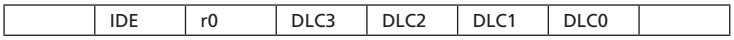


Figure 5.4 CAN control field bits

The IDE bit indicates the CAN format used. IDE = 0 for the standard CAN 2.0A format, and IDE = 1 for the extended CAN 2.0B format.

r0 is a reserved bit.

DLC3 – DLC0 determine the number of bytes in the data field (0 to 8 bytes). Table 5.1 shows how bits DLC3 – DCL0 are used to indicate size of the data field. For example, if 4 bytes are to be sent in standard CAN 2.0A format then the control field has the following bits:

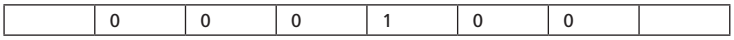


Table 5.1 Determining size of the data field

No of data bytes	DLC3	DLC3	DLC1	DLC0
0	D	D	D	D
1	D	D	D	R
2	D	D	R	D
3	D	D	R	R
4	D	R	D	D
5	D	R	D	R
6	D	R	R	D
7	D	R	R	R
8	R	D or R	D or R	D or R

D = Dominant level (0), R = Recessive level (1)

5.1.5 Data Field

The data field contains the actual data of the message. The data size can vary from 0 to 8 bytes. The data is transmitted with the MSB byte first.

5.1.6 CRC Field

The CRC (Cyclic Redundancy Check) field consists of the following bits:

- 15-bit CRC sequence
- 1-bit CRC delimiter

The CRC field is used to check the frame for a possible transmission error. The CRC calculation includes the SOF bit, arbitration field, control field, and data fields (see Figure 5.5). The receiving node calculates the CRC based on the received data and compares it with the CRC sent with the frame. If the two CRCs do not match, an error is assumed.

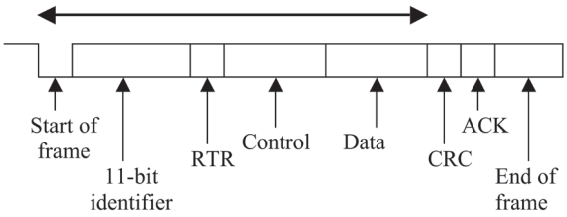


Figure 5.5 Fields used in the CRC calculation

The receiving node calculates the CRC in the same way as the transmitting node. There are many CRC generator polynomials, but the one used by CAN bus is as follows:

- The message is regarded as a polynomial and is divided modulo-2 by the CRC generator polynomial. The CRC is generated by hardware and uses the following polynomial:

  $x^{15} + x14 + x10 + x8 + x7 + x4 + x3 + 1$
  i.e. 16-bit, bit pattern "1100010110011001"
- The remainder of this modulo-2 division is the CRC sequence which is transmitted together with the message.
- The receiver divides the message using the same generator polynomial.
- If a CRC error is detected the receiver discards the message and transmits an Error Frame to request re-transmission.

A delimiter bit is sent at the end of the 15-bit CRC field. The delimiter is always recessive (i.e. 1). The reason for sending the delimiter bit is to allow more time for the CRC to be calculated.

5.1.7 ACK Field

The ACK field consists of the following bits:
- 1-bit ACK
- 1-bit delimiter

The ACK bit (sometimes called the ACK slot) is a confirmation that the frame has been received normally with no errors. i.e. it is a confirmation that the CRC check is successful by the receiving nodes.

The transmitting node sends the ACK bit in recessive (i.e. 1) mode. During the ACK slot, the transmitting node switches to receive mode by sending a recessive signal to the bus. All the receiving nodes which have received a data frame with a *correct* CRC report this to the transmitter by sending a dominant bit during the ACK time slot. If the transmitting node detects a positive acknowledge, that is a dominant ACK, the transmitting node knows that at least one receiving node has got the message correctly.

When there are several receiving nodes, the various possibilities are as follows:

- If the transmitting node detects a dominant bit during the ACK slot, the transmitting node knows that at least one receiving node has got the message correctly (remember the bus "Wired AND" logic). In actual fact, this means that all the receiving nodes received the message correctly. This is because if one node receives a CRC error, it will send an error frame which will destroy the *faulty* message for every node (i.e. every node, including the transmitting node will receive the error frame and discard the message). In this case, the transmitting node will re-transmit the message after a specified timeout.
- If the transmitting node detects a recessive bit during the ACK slot, the transmitting node knows that all the receiving nodes have reported error. In this case, it is probable that the transmitting node calculated the CRC wrongly, or there are no receivers, or there was data corruption on the bus. In this case, the transmitting node will re-transmit the message after a specified timeout.

The 1-bit ACK delimiter is always in recessive state.

5.1.8 End of Frame Field

This is a 7-bit field consisting of 7 recessive bits. The data frame is always terminated by this field. When several data frames are to be sent sequentially it is important that a 3-bit interframe gap is left between each frame. The interframe gap is always in recessive state.

5.2 Remote Frame

The remote frame is used by a node to request transmission of a message from another node. Remote frames are seldom used and there use is not recommended by the CiA.

The remote frame is identical to the data frame except the following differences:
- The remote frame does not have a data field (see Figure 5.6). i.e. it consists of the SOF, identifier field, RTR, control field, CRC field, ACK field and the EOF field.
- The RTR bit of the remote frame is set to recessive (i.e. 1) state.

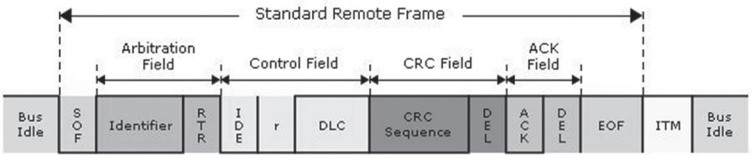


Figure 5.6 The remote frame

Nodes receiving remote frames and accepting them will send a data frame with the requested data, but only one node can send the data.

The data request actually happens after sending two frames. The first frame is the remote frame (with RTR = 1) telling the other nodes that data is being requested. The second frame is a data frame with the same identifier, telling other nodes what the requested data is. The DLC of the remote frame must be same as the DLC of the data frame sent just after the remote frame. If a remote frame and a data frame with the same identifier attempt to access the bus at the same time, the data frame will be given access as its RTR is set to 0 and hence has a higher priority (remember that RTR bit is part of the bus arbitration).

5.3 Error Frame

Error frames are generated and transmitted by the CAN hardware and are used to indicate when an error has occurred during transmission. Generation of an error frame terminates the faulty data on the bus and all the nodes on the bus discard the current data.
As shown in Figure 5.7, an error frame consists of a 6-bit Error Flag and an 8-bit Error Delimiter. A 3-bit interframe gap should also be used at the end of the error frame.

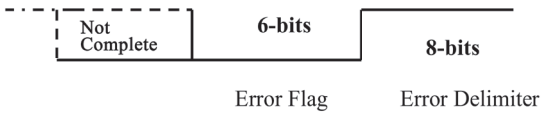


Figure 5.7 CAN Error frame

According to the CAN standard any 5 consecutive bits of the same polarity (5 consecutive dominant bits, or 5 consecutive recessive bits) is a violation of the bus standard and is considered an error condition. When the CAN standard is violated the data currently on the bus is considered to be faulty and is discarded by all the nodes. After the interframe gap the transmitting node re-transmits the same frame. In the error frame 6 dominant bits are sent consecutively and as this violates the CAN standards the complete frame is discarded (as we shall see later, it is permissible to send more than 5 bits with the same polarity as actual data by using a method called "bit stuffing").

There are two types of error flags: active error flags, and passive error flags. An active error flag consists of 6 dominant bits, and passive error flag consists of 6 recessive bits.

Error frames are sent immediately after an error has been detected. In the case of CRC type errors, the error frame is sent after 2 bit time of detecting the error. This is so that there is no conflict with the ACK field.

It is important to realize that when an error frame is sent by a node, all the other nodes on the bus recognize this error condition and they too send error frames. It is possible that some of these nodes may also start sending error frames at the same time, but some nodes may start sending error frames after detecting the error condition on the bus, i.e. after 6 bits. Since these nodes will also start sending their error frames, it is possible that the error flag can be extended to a maximum of 12 bits on the bus. Figure 5.8 shows the case where the error frame can be extended to 12 dominant bits of error flag, and 8 recessive bits of error delimiter.

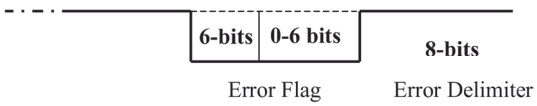


Figure 5.8 Extended error frame

The error delimiter field consists of 8 recessive bits and is sent after the error flag. In the case when the error flag on the bus is more than 6-bits, all the nodes on the bus will send the first bits of their error delimiter bits and keep sending recessive bits until the bus turns into recessive state (the bus will become recessive when all the nodes sent the first bit of their error delimiter bits). When the bus is in recessive state all the nodes will send the remaining 7-bits of their error delimiter bits.

It is important to realize that when the worst case is considered, the error frame consists of 20 bits (12 error flag bits + 8 error delimiter bits). Adding the 3-bit interframe gap we have 23 bits. With a bus data rate of 1Mbps (i.e 1μs bit time), the worst case total error recovery time will be 23μs. This is the worst case time that the bus can recover and re-transmission of frames can start on the bus. This is rather a short time and is one of the advantages of the CAN bus.

The CAN bus error conditions will be described in detail in the next chapter.

5.4 Overload Frame

The overload frame is used by a receiving node to indicate that it is not yet ready to receive frames. As shown in Figure 5.9, the overload frame is similar to the error frame and consists of:

- 6-bits Overload Flag
- 8-bits Overload Delimiter

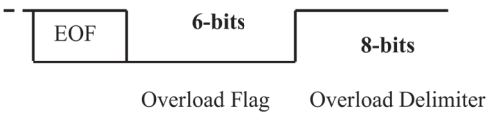


Figure 5.9 CAN Overload frame

The overload frame can be sent between two data or remote frames. There are two kinds of overload conditions that can lead to the transmission of an overload frame:

1. The internal conditions of a receiver, which requires a delay of the next data frame or remote frame. i.e. a receiving node is temporarily overloaded and will not be able to receive any data, requiring a delay on the bus.
2. Detection of a dominant bit during the interframe gap or during the EOF frame.

The start of an overload frame due to case 1 is only allowed to be started at the first bit time of an expected interframe gap. i.e the 6-bit overload flag will override the 3 interframe gap bits. This is important and is the main timing difference between the error frame and the overload frame (remember that the error frame starts as soon as the error is detected where the frame is not completed and as such causes the frame to be discarded by all the nodes).

The overload frames due to case 2 start one bit after detecting the dominant bit. Overload Flag consists of six dominant bits. The overload flag violates the CAN bus rules (more than 5 bits of the same polarity on the bus) and as a consequence, all other nodes also detect an overload condition and on their part start transmission of an overload flag. As in the error flag, the overload flag can be extended to up to 12 bits. No more than two overload frames can be generated to delay a data or remote frame. Overload Delimiter consists of eight recessive bits. The overload delimiter is of the same form as the error delimiter.

5.5 Extended CAN Frames

Now that we have examined all the standard CAN 2.0A frames, we can look at the extended frames and see how the extended protocol differs from the standard protocol and also what additional benefits are available with the extended protocol.

The CAN bus was originally developed for the automobile industry and as such the standard message identifier is 11-bits. With 11-bits it is possible to have 2048 different identifiers in a system. In order to improve the functionality of the bus and also increase the identifier capacity the extended CAN (also called the CAN 2.0B) was introduced in 1985.

Figure 5.10 shows the data frame of the extended CAN.

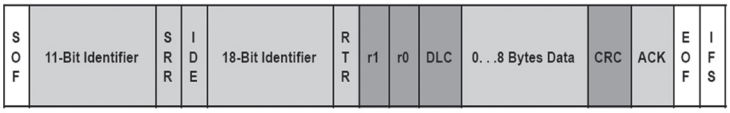


Figure 5.10 Extended CAN data frame

The extended CAN is based on a 29-bit message identifier. The message identifier is in two parts: the standard 11-bit identifier, and an 18-bit extended identifier, making a total of 29-bits.

As before, SOF bit indicates the start of frame.

SRR replaces the RTR bit and is always recessive.

The IDE bit is transmitted as dominant for a CAN 2.0A bus, and as recessive for CAN 2.0B bus.

The RTR bit is as in the standard protocol.

The other bits of the CAN 2.0B protocol are same as the CAN 2.0A protocol.

In CAN 2.0B systems the arbitration fields consist of all the fields from the 11-bit identifier to the RTR field. i.e. total 32 bits (see Figure 5.11). Since the SRR and IDE are recessive bits in CAN 2.0B systems, in a mixed system where both the standard and the extended systems are used, a standard data frame will have a higher priority than an extended data frame.

The extended format has some trade-offs:
- The bus latency time is longer.
- Messages in extended format require about 20% more bandwidth.
- The error detection performance is lower because there are more bits in a frame and the CRC calculation takes longer time.

5.6 Summary

This chapter has described all the frames of the CAN bus. It is very important to understand the structure of all the frames before trying to analyze a frame, or before trying to develop a CAN bus based program.

Both the standard CAN 2.0A and the extended CAN 2.0B data frames have been explained.

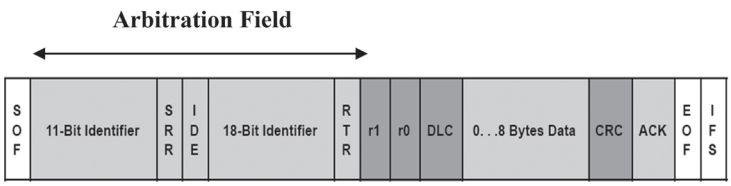


Figure 5.11 The arbitration field in extended CAN

Chapter 6 CAN Bus Error Conditions

Before going into the details of CAN bus error types, it is worthwhile to look at the Bit Stuffing mechanisms on the bus.

6.1 Bit Stuffing

The CAN standard specifies that any bits of the same polarity (recessive or dominant) on the bus that is longer than 5 bits long is a violation of the standard. In fact, this standard was used to send error frames on the bus consisting of 6 dominant bits in sequence.

In some applications it may be required to send more than 5 bits of the same polarity (e.g. the data bits may contain more than 5 bits of the same polarity). This type of situation is handled on the bus by the transmitting node inserting a bit of opposite polarity after the 5th bit. The receiving node then removes this bit. This mechanism is called Bit Stuffing and it allows to synchronize the transmit and receive operations to prevent timing errors Note that the error frames and overload frames are transmitted without Bit Stuffing. Also, during a reception, if the 6th bit is same as the 5th bit then a Stuffing Error occurs on the bus.

Bit stuffing is allowed from the SOF field to the CRC field (see Figure 6.1). Bit stuffing is not allowed in the static fields of a frame. i.e. it is not allowed in the following fields:

- CRC delimiter
- ACK field
- EOF field
- Interframe gap

Figure 6.2 shows an example bit stuffing where the transmitting node added a recessive bit after the 5th dominant bit. The receiving node removed this bit and thus more than 5 dominant bits have successfully been transmitted on the bus.

Similarly, in Figure 6.3, the transmitting node added a dominant bit after the 5th recessive bit. The receiving node again removed the stuffed bit and thus more than 5 recessive bits have successfully been transmitted on the bus..

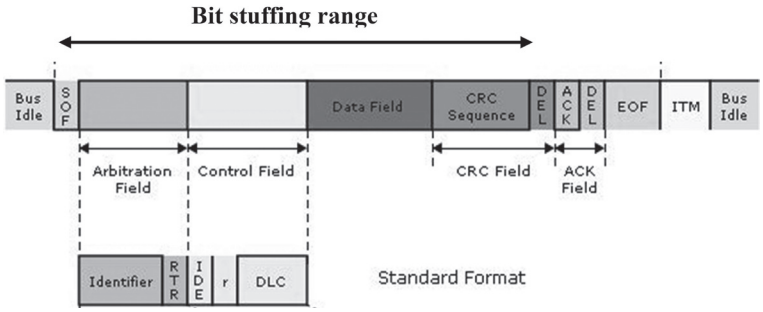


Figure 6.1 Fields where bit stuffing can be used

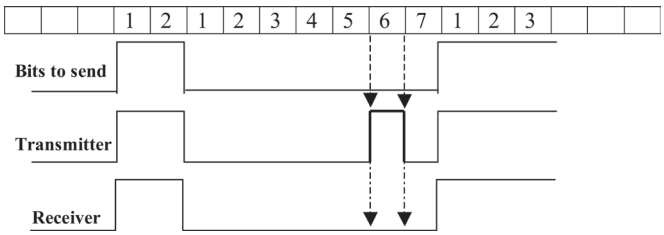


Figure 6.2 Bit stuffing example (adding a recessive bit)

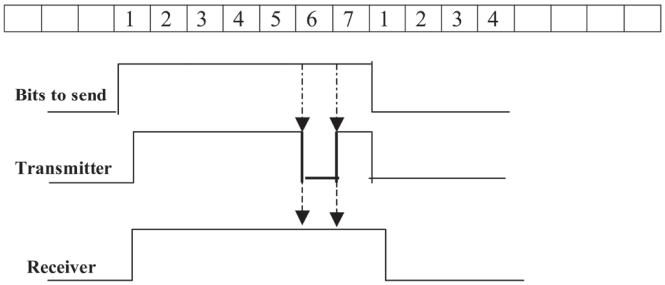


Figure 6.3 Bit stuffing example (adding a dominant bit)

6.2 CAN Bus Error Detection

There are five types of errors that can occur on the bus. If any of these errors is observed, the error frame is transmitted which causes the current frame on the bus to be declared as *faulty* and discarded by all nodes on the bus. The error type are:

- Bit error
- Bit stuffing error
- CRC error
- Frame error
- ACK error

Now lets us have a look at these error conditions in more detail.

6.2.1 Bit Error

When a node transmits a bit on the bus it also monitors the bus and compares the transmitted bit with the actual level on the bus. A bit error is said to happen when the transmitted bit is not same as the bit level on the bus.

Note that a bit error will not happen during the arbitration phase where a node transmits a recessive bit while another node transmits a dominant bit. Also, while a node is transmitting an error frame with 6 consecutive recessive bits (passive error flag, see section 5.3) if a dominant bit is detected on the bus this will not create a bit error.

6.2.2 Bit Stuffing Error

A bit stuffing error will occur if the 6^{th} bit on the bus is same as the 5^{th} bit. Note that during the transmission of an error frame or an overload frame no bit stuffing errors will occur (even though 6 consecutive dominant bits are sent).

6.2.3 CRC Error

The transmitting node calculates the CRC (cyclic redundancy check) using all the bits from the SOF to the end of data field, and then inserts the calculated value into the frame just after the data field. Any receiving nodes also calculate the CRC and expect to find the same value as the received value inside the frame. A CRC error will happen if the CRC value sent by the transmitting node (inside the

frame) is not same as the CRC value calculated by any receiving node (calculated using the received bits from SOF to the end of the data field).

When a CRC error is observed the error frame is sent after a delay of 2 bit times so that there is no confusion with the ACK slot.

6.2.4 Frame Error

Frame errors relate to errors in the format of a frame. A frame error will be observed when the static field of a frame is not as expected. For example, if the ACK delimiter or the CRC delimiter is missing, or if the EOF field contains dominant bits, or if the interframe gap contains dominant bits.

6.2.5 ACK Error

The transmitting node monitors the bus and expects a dominant bit during the ACK slot. An ACK error will happen if the transmitting node detects a recessive bit during the ACK slot.

6.3 CAN Bus Fault Confinement

CAN Bus is a very reliable bus and have the ability to detect errors, and also to correct itself if any abnormalities are detected on the bus. Each node has an error signalling ability which depends on the historic behaviour of that node. For example, if a node keeps generating errors continuously then that node can voluntarily remove itself from the bus so that it does not cause a dead-lock on the bus.

There are three fundamental states that define the error signalling state of each node:

- Error Active
- Error Passive
- Bus Off

The normal state of a node is the Error Active state. When a node is in Error Active state, it can send all frames, including error frames.

When a node is in Error Passive state, the node can send all frames except the error frame. i.e. the node can not participate in error determination on the bus.

A node is isolated from the bus and stops communicating when it is in Bus Off state.

Two internal error counts are maintained by the CAN controller hardware of each node in order to determine the state of the node at any time. These counters are:

- Transmit Error Counter (TEC)
- Receive Error Counter (REC)

The TEC counter increments whenever an error is detected within the node while sending a frame. Also, the TEC counter is decremented whenever a frame is sent successfully. In a similar way, the REC counter is incremented if an error occurs while receiving a frame, and is decremented whenever a successful receive operation is performed.

If any of the two counters in a node become greater than 127, the node goes into the Error Passive mode. In this mode the node can still send and receive frames, but can not destroy frames on the bus (i.e. it can not send error frames). Because the error counters get decremented it is possible that a node which is in Error Passive mode can return to the normal Error Active mode if *both* of its counter are equal to or less than 127.

If on the other hand, the error counters increment further, the node stays in Error Passive mode until the TEC counter becomes greater than 255. At this point the node moves to the Bus Off mode. At this mode the node shuts down and stops sending or receiving frames on the bus. A node which is in Bus Off mode stays in this mode and self recovery is not possible. Re-initializing the controller, or re-initializing the overall bus system should move the node back to the normal Error Active mode. Figure 6.4 shows the CAN bus error states.

The CAN bus is developed such that one of the most stringent requirements set by automotive applications is fulfilled by its standards. The CAN bus is so reliable that, it has been calculated that if a network based on 250 kbps operates for 5 hours a day for a year at an average bus load of 25%, an undetected error occurs only once per 1000 years!.

In the above example, an undetected error means that bits in a message get corrupted in such a way that the CRC algorithm does not detect it and a *faulty* message frame is transmitted on the bus. What this means in automotive terms is that, if a car is driven for 5 hours a day, every day for a year, an error will be undetected in the car electronics once every 1000 years, which is well above the lifetime of the car!.

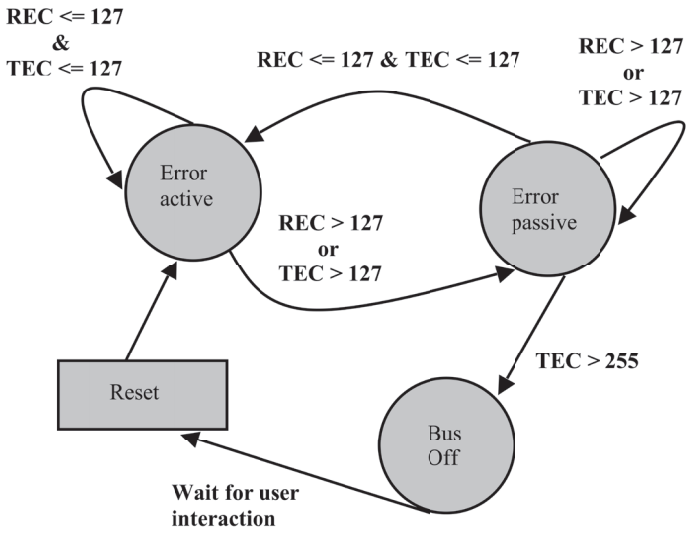


Figure 6.4 CAN bus error states

6.4 Summary

CAN bus is a highly reliable bus structure. This chapter has described all the error conditions that can happen on the CAN bus. In addition, the fault confinement methods have been explained. It is shown that a node that is reporting an error continuously can voluntarily disconnect itself from the bus, thus making the bus available to other nodes in the system.

Chapter 7 Data Exchange on CAN Bus

In this chapter we shall be looking at how data is exchanged on the CAN bus. CAN bus is a multi-master bus where any node can be a transmitting or a receiving node at any one time.

When a node transmits on the bus all other nodes listen and they receive the transmitted message. Although all the receiving nodes receive all the messages they may decide not to act on the message contents as the message may not be relevant to the node.

The nodes on the CAN bus do not have any node addresses. Thus, messages are not transmitted to addressed nodes, but rather they are broadcast on the bus. Also, a node does not need to know where a message comes from. Similarly, a transmitting node may not necessarily know which nodes have actually acted on the message sent. The nodes only receive the messages they are programmed to receive.

Although the internal workings of different CAN controllers may be different, as far as data exchange mechanisms are concerned, the basic principles of data exchange are described briefly in this chapter.

7.1 Data Exchange With Data Frames

Data frames are the most information frames in CAN bus as they enable data to be sent from a transmitting node to other nodes on the bus. A data frame has a dominant RTR bit.

When a node wants to send data to other nodes on the bus it forms a data frame. This data frame basically includes a message identifier, actual data bytes, and error checking bits. The message identifier is very important as it is used by the receiving nodes to decide whether or not to accept this message. The CAN bus controllers in receiving nodes have built in Acceptance Filters (or Receive Filters). These filters can be programmed and loaded with values by the programmer or system user. The message identifiers of data frames on the bus are compared with the filter values, and the message is accepted by the controller if the filter value is same as the message identifier. If the message identifier and the filter values are not the same then the message is not accepted by the controller. Thus, by programming the acceptance filters we can enable a node to accept or reject a message (note that all the receiving nodes **receive** all the messages but they may not *accept* these messages).

Most CAN bus controllers also have built in *Filter Masks*. These masks are used to determine which bits in the message identifier are to be compared with the values in the acceptance filters. For example, setting all the filter mask values to 1s will make sure that all the bits of the message identifiers are to be compared with all the bits of acceptance filters.

Figure 7.1 shows an example data exchange on the bus. In this example there are three nodes on the bus, named Node A, Node B, and Node C. Assume that the acceptance filters in each node's controller, at the time of the data transmission are set to the following values:

Node A: Acceptance filter value = "00000000011"
Node B: Acceptance filter value = "00000011111"
Node C: Acceptance filter value = "00000000111"

Assume that Node A has high priority and transmits a data frame with the message identifier set to bit pattern "00000000111". The other two nodes are in receive mode and compare the message identifier on the bus with their acceptance filter values. Node C has the same acceptance filter value as the message identifier and thus its controller accepts the data frame sent by Node A. The controller in Node B compares its acceptance filter with the message identifier and ignores the data frame as the two are not same.

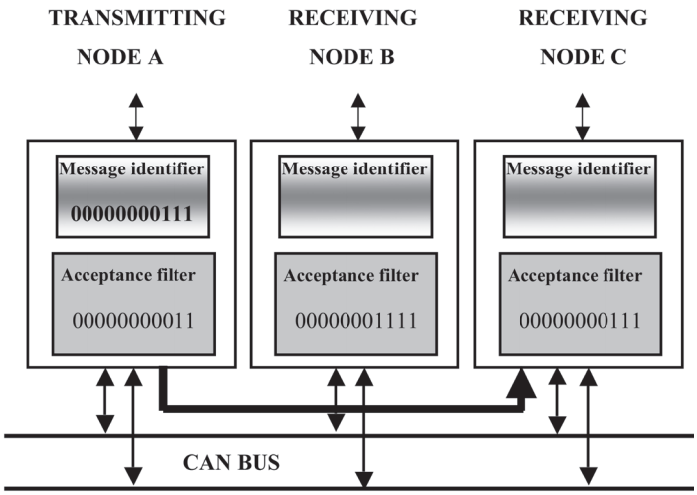


Figure 7.1 Data frame exchange on the bus

Note that when a message is sent on the bus, the controllers of all the nodes *receive* this message, but they may not accept the message. Also, although the controller may accept a message, this message usually stays in a receive buffer of the controller until the applications software makes a request to pull the message out of this buffer and copy it to its internal data structures.

Figure 7.2 shows another example where Node A transmits a data frame and both Node B and Node C accept this data frame. In this example the acceptance filter of Node B is changed to "00000000111".

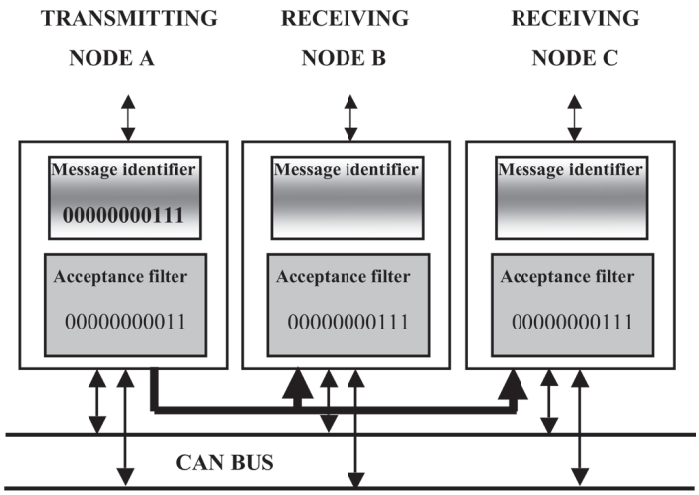


Figure 7.2 Example data frame exchange

In the example in Figure 7.3, Node A transmits a data frame but none of the other two nodes accept this data frame. Here, the acceptance filters of Node B and Node C are changed to "00000111111" and "11110000000" respectively.

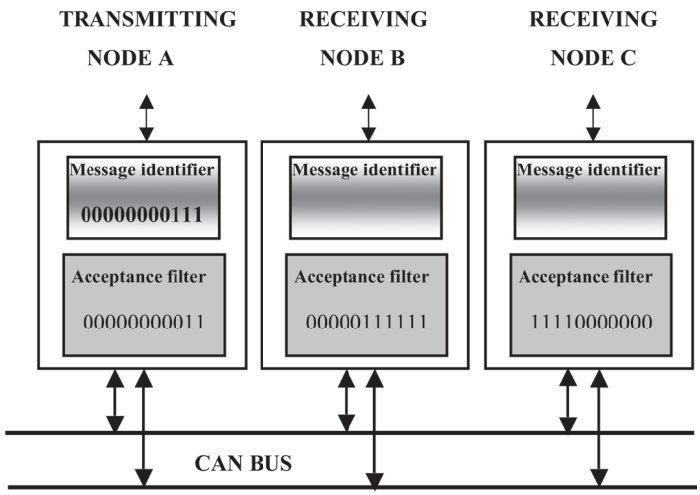


Figure 7.3 Example data frame exchange

The data exchange mechanism when a data frame is sent on the bus is summarized below:

- A node transmits a data frame with a certain message identifier
- The controllers of all nodes on the bus receive this message
- The controllers of the receiving nodes on the bus compare this message identifier with their acceptance filters
- The nodes whose acceptance filters match the message identifier *accept* the data frame. The acceptance filters in these nodes were set such that the message with the specified message identifier is relevant to the application layers of these nodes.
- The nodes whose acceptance filters do not match the message identifier *ignore* the data frame. The acceptance filters in these nodes were set such that the message with the specified message identifier is not relevant to the application layers of these nodes.

Note that the message identifiers and the acceptance filter values are set by the engineers who design and maintain the bus.

7.2 Remote Frames on the Bus

The remote frames are recognized by a recessive RTR (Remote Transmission request) bit in their frames and having no data fields. These frames are used to request data from another node. When a remote frame is sent on the bus, another node sends the requested data in a data frame. Both of these nodes have the same message identifiers.

Figure 7.4 shows an example where three nodes are on the bus with the message identifiers:

Node A: Acceptance filter value = "00000000011"
Node B: Acceptance filter value = "00000011111"
Node C: Acceptance filter value = "00000000111"

Node A sends a remote request by setting the RTR bit to recessive state and the identifier to "00000000111". Nodes B and C receive the remote request, and Node C accepts this remote request.

In Figure 7.5, Node C sends the requested data with a data frame. In this example, Node A receives the requested data frame. Note that Node A sends the requested data frame with the same identifier as the remote frame.

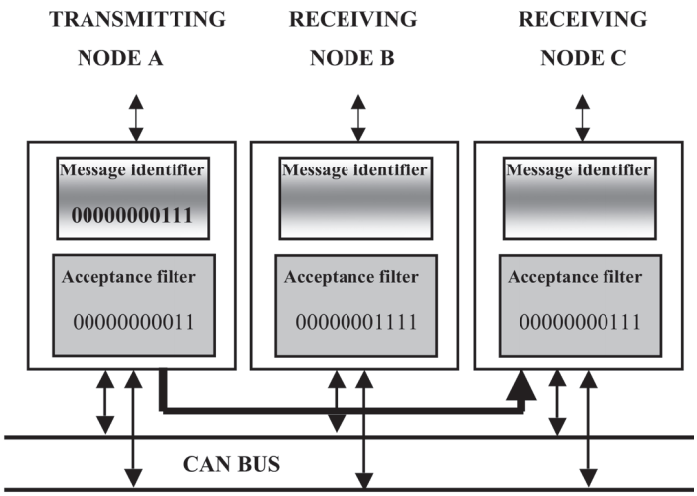


Figure 7.4 Example of remote frame

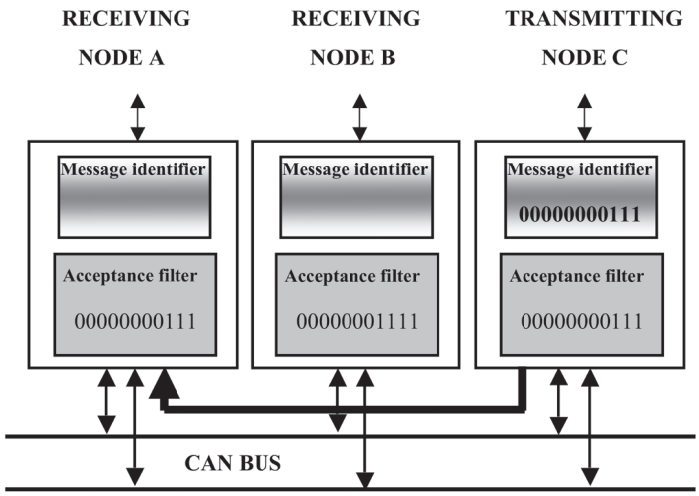


Figure 7.5 Example data frame with requested data

7.3 Summary

This chapter has explained the important topic of data exchange on the CAN bus. Examples are given to show how a node can broadcast data on the bus, and how the receiving nodes make a decision whether or not to accept the transmitted data.

When data is transmitted on the bus, all the receiving nodes receive the data, but they may not accept this received data (i.e. they may not do any action based on the contents of this data).
The condition for the accepting a received data frame is programmed into the acceptance filters (or receive filters) of the receiving nodes. A node will accept a received data if its acceptance filters match the message identifier sent by the transmitting node.

Chapter 8 CAN Bus Timing

Timing and proper synchronization of different nodes on the CAN bus is very important for the proper operation of the bus. Basically, all the nodes on a CAN bus are synchronized by the falling edge (recessive to dominant transition) of the Start-Of-Frame (SOF) bit. This is actually a limited way of making sure that all the nodes are synchronized properly. In this chapter we shall be looking at the CAN bus timing and synchronization methods.

8.1 Bit Timing

Signals on a CAN bus are based on NRZ (Non-Return-to-Zero) signalling scheme where a signal can stay at the same state for long time with no edges. This makes the data synchronization very difficult as there may not be any signal edges to use for synchronization. Luckily the bit stuffing method ensures that a frame can not keep the bus at the same polarity for more than 5 bits, and this feature helps to synchronize the data.

Figure 8.1 shows an example NRZ signal where the same bit pattern is sent for long periods of time (10 bit time in this example), thus making it difficult for a receiver to know the start or end of the data, and thus requiring additional bit synchronization methods.

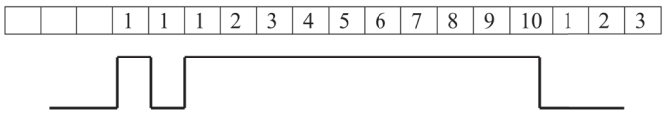


Figure 8.1 Example NRZ signal

CAN bus timing is based on the clock mechanisms employed within the CAN bus controller. Basically, a free running crystal clock provides pulses at a high rate. Inside the controller this clock rate is divided by a user programmable prescaler, BRP, (the prescaler usually has a minimum division rate of 2), and the resulting lower rate clock is used as the *Baud Rate* clock (or the node CAN clock) which determines the timing of the node.

The CAN nominal bit rate is defined as the number of bits transmitted every second without synchronization. The inverse of the nominal bit rate is the nominal bit time. All devices on the CAN bus must use the same bit rate, even though each node can have its own internal clock frequency.

As shown in Figure 8.2, the CAN nominal bit time consists of four non-overlapping time segments named:

- Synchronization segment, *Sync_Seg*
- Propagation segment, *Prop_Seg*
- Phase buffer segment 1, *Phase_Seg1*
- Phase buffer segment 2, *Phase_Seg2*

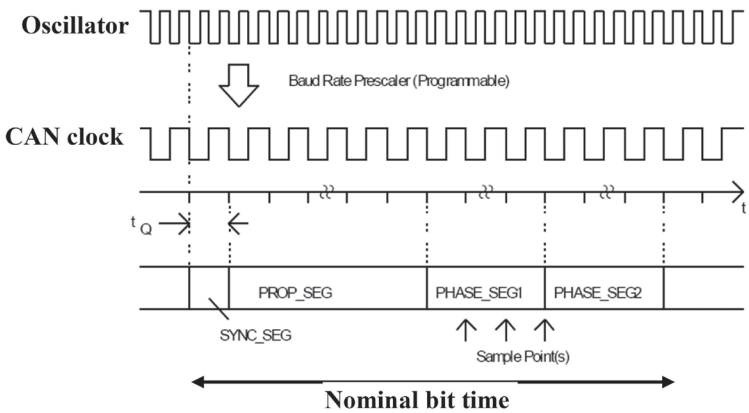


Figure 8.2 CAN controller oscillator and bit time

The *Sync_Seg* segment is used to synchronize various nodes on the bus, and an edge is expected to lie within this segment. Any shift in the *Sync_Seg* will be detected by the nodes and each node will adjust the length of its phase buffer segments accordingly to re-synchronize the node. For a transmitting node, the new bit value is transmitted from the beginning of the *Sync_Seg*. For a receiving node, the start of the received bit is expected to occur during the *Sync_Seg*.

The *Prop_Seg* segment compensates for physical delay times on the bus, such as the propagation times from a transmitter to a receiver and back to a transmitter.

The *Phase_Seg1* and *Phase_Seg2* segments compensate for edge phase errors. These segments can be lengthened or shortened to synchronize the node.

The *Sample Point* (see Figure 8.2) is the point in time where the actual bit value is located and occurs at the end of *Phase_Seg1*.

Each segment is divided into units known as time *quantum*, or T_Q, where a *quantum* is basically equal to one period of the CAN clock (see Figure 8.2). A desired bit timing can be set by adjusting the number of T_Q's that comprise one message bit and the number of T_Q's that comprise each segment in it.

The time quantum of each segment can vary from 1 to 8. The lengths of the various time segments are:

- Sync_Seg is always 1 T_Q
- Prop_Seg is programmable from 1 T_Q to 8 T_Q
- Phase_Seg1 is programmable from 1 T_Q to 8 T_Q
- Phase_Seg2 is programmable and is equal to the larger of the Phase_Seg1 or to the IPT (Information Processing Time). The IPT is normally equal to 2 T_Q , but it can be equal to 3 T_Q if the Baud rate prescaler is set to 1, or if 3 samples per bit are selected.

In addition to the above time segments, another programmable timing parameter called *SJW* (Synchronization Jump Width) is used to define the upper limit of the amount that can be used to shorten or lengthen the phase buffers. By setting the bit timing, a sampling point can be set so that multiple nodes on the bus can sample messages with the same timings.

By considering all the segments, it is obvious that the nominal bit time can be programmed from a minimum of 5 time quanta to a maximum of 25 time quanta. In practise most controllers require a minimum of 8 T_Q and a maximum of 25 T_Q.

The nominal bit time is given by:

$$T_{BIT} = T_Q * (Sync\_Seg + Prop\_Seg + Phase\_Seg1 + Phase\_Seg2)$$

It then follows that the nominal bit rate (NBR) is given by:

$$NBR = 1 / T_{BIT} \tag{8.1}$$

The time quantum is derived from the oscillator frequency and the programmable baud rate prescaler, with integer values usually from 1 to 64. The time quantum can be expressed as:

$$T_Q = 2 * BRP / F_{OSC} \tag{8.2}$$

Where, T_Q is in μs, F_{OSC} is in MHz, and BRP is the Baud rate prescaler (1 to 64).

From equation (8.2) we can write:

$$T_Q = 2 * BRP * T_{OSC} \tag{8.3}$$

where, T_{OSC} is in μs.

As an example, assuming a controller clock frequency of 20MHz, a baud rate prescaler value of 2, and a nominal bit time of $T_{BIT} = 8 * T_Q$, we can calculate the nominal bit rate as:

From equation (8.2):

$$T_Q = 2 * BRP / F_{OSC}$$

or,

$$T_Q = 2 * 2 / 20 = 0.2\mu s$$

Also,

$$T_{BIT} = 8 * T_Q = 8 * 0.2 = 1.6\mu s$$

Thus, from equation (8.1):

$$NBR = 1 / T_{BIT} = 1 / 1.6\mu s = 625{,}000 \text{ bps}$$

or, NBR = 625 Kb/s

8.2 Selection of Bit Timing Segments

The correct selection of the bit timing segments are important for the correct synchronization of the nodes on the bus.

The *Sync_Seg* is always 1 T_Q and is not programmable. The minimum value of SJW is 1 T_Q, and the maximum is 4 T_Q, and it can not exceed Phase_Seg1. In this section we shall look at how the other timing segments (Prop_Seg, Phase_Seg1, Phase_Seg2, and SJW) can be calculated so that a given CAN bus controller can be programmed correctly.

8.2.1 The Prop_Seg

Figure 8.3 shows the propagation delay between two nodes A and B. Here, a bit sent by Node A is received by Node B after time $t_{prop(A,B)}$ and a bit sent by Node B is received by Node A after $t_{prop(B,A)}$. The propagation time from sending Node A to receiving Node B consists of the delay through Node A's controller ($t_{TX(A)}$), plus the propagation delay along the bus from Node A to Node B ($t_{BUS(A,B)}$), plus the delay through the controller of receiving Node B ($t_{RX(B)}$). i.e.:

$$t_{prop(A,B)} = t_{TX(A)} + t_{BUS(A,B)} + t_{RX(B)} \tag{8.4}$$

Assuming the Nodes A and B are at the opposite ends of the bus (i.e. the propagation delay is maximum between Node A and Node B), the minimum time for the propagation segment, Prop_Seg to ensure correct synchronization is given by:

$$t_{PROP\_SEG} = t_{prop(A,B)} + t_{prop(B,A)} \tag{8.5}$$

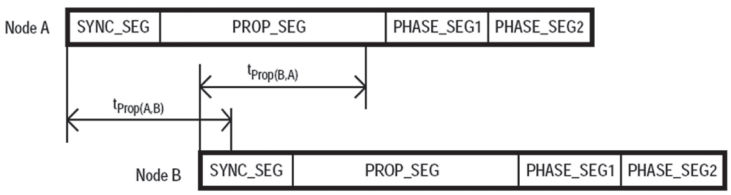


Figure 8.3 Propagation delay between nodes

From Equation (8.4) we have:

$t_{PROP\_SEG} = t_{TX(A)} + t_{BUS(A,B)} + t_{RX(B)} \quad + t_{TX(A)} + t_{BUS(A,B)} + t_{RX(B)}$

or,

$$t_{PROP\_SEG} = 2(t_{TX(A)} + t_{RX(B)} + t_{BUS(A,B)})$$

or, in general,

$$t_{PROP\_SEG} = 2(t_{TX} + t_{RX} + t_{BUS}) \tag{8.6}$$

where, t_{TX} is the propagation delay through the transmitter part of the CAN controller, t_{RX} is the propagation delay through the receiver part of the CAN controller, and t_{BUS} is the propagation delay between two furthest nodes on the bus.

We can now calculate the minimum time quantum that must be allocated to the Prop_Seg alone as:

$$\text{Prop\_Seg} = \text{NEAREST\_HIGHER\_INTEGER}(t_{PROP\_SEG} / T_Q) \quad (8.7)$$

Where the nearest integer value should be taken after the division.

Table 8.1 shows the recommended bit timings in CAN bus systems, showing the recommended length of time quantum, and the location of the sample point.

Table 8.1 Recommended bit timings

Baud rate	Bit time (μs)	Time quanta (per bit)	Length of time quantum (ns)	Location of sampling point (T_Q)
1 Mbps	1	8	125	6
500 kbps	2	16	125	14
250 kbps	4	16	250	14
125 kbps	8	16	500	14
50 kbps	20	16	1250	14
10 kbps	100	16	6250	14

8.2.2 The Oscillator Tolerance

The oscillator tolerance is also an important factor to consider as the acceptable value depends on the bit timing segments and the value of SJW.

The relationship between the maximum oscillator frequency and the synchronization jump width, SJW, is given by the equation (source: *Bosch CAN Specification, Version 1.2, 1990*):

$$(2 \times \Delta f) \times 10 \times T_{BIT} < t_{SJW}$$

or,

$$\Delta f < t_{SJW} / (20 \times T_{BIT})$$

where,

Δf is the maximum oscillator frequency tolerance, T_{BIT} is the nominal bit time, and t_{SJW} is the synchronization jump width time. We can write the above equation as:

$$\Delta f < SJW * T_Q / (20 \times TQPB * T_Q)$$

or,

$$\Delta f < SJW / (20 \times TQPB) \quad (8.8)$$

Where, TQPB is the Time Quanta Per Bit. The TQPB can be calculated as:

TQPB = Data bit period / Controller time quantum

For example, if the required Bit rate is 125 kbps, then the Data bit period is 8 μs. If the CAN controller clock frequency (after the prescaler) is 2 MHz, then the time quantum is 0.5μs. The TQPB is then calculated to be 8 / 0.5 = 16.

In the event of an error, an error Flag is transmitted on the bus. All the nodes that receive the Error Flag transmit their own Error Flags. An Error Flag consists of 6 dominant bits, and there could be up to 6 dominant bits before the Error Flag. A node must sample the 13th bit after last re-synchronization. The relationship between the maximum oscillator frequency and bit segments during an Error Flag can be expressed as (source: *Bosch CAN Specification, Version 1.2, 1990*):

$$(2 \times \Delta f) \times (13 \times T_{BIT} - t_{Phase\_Seg2}) < MIN(t_{Phase\_Seg1}, t_{Phase\_Seg2})$$

or,

$$\Delta f < (MIN(t_{Phase\_Seg1}, t_{Phase\_Seg2}) / [2 \times (13 \times T_{BIT} - t_{Phase\_Seg2})]$$

where, $MIN(t_{Phase\_Seg1}, t_{Phase\_Seg2})$ returns the smaller of the arguments.

We can write the above equation as:

$$\Delta f < (MIN(Phase\_Seg1, Phase\_Seg2) * T_Q / [2 \times (13 \times TQPB * T_Q - Phase\_Seg2 * T_Q)]$$

or,

$$\Delta f < (MIN(Phase\_Seg1, Phase\_Seg2) / [2 \times (13 \times TQPB - Phase\_Seg2)] \quad (8.9)$$

We can now select the timing parameters as described in the following steps (see source: *AN1798, CAN Bit Timing Requirements, Stuart Robb, Freescale Semiconductor, East Kilbride, Scotland*):

Selecting the Timing Parameters

1. Calculate the maximum propagation time on the bus using the cable length, the propagation speed on the bus, and the controller transmit and receive delays (from manufacturers data sheets). Use Equation (8.6) to calculate the propagation delay.

2. Choose CAN clock frequency so that the nominal bit time, T_{BIT} is an integer between 8 and 25.

3. Use Equation (8.7) to calculate the required time quanta for the Prop_Seg. If the calculated value is greater than 8, go back to Step 2 and choose a lower clock frequency.

4. Determine Phase_Seg1 and Phase_Seg2. Subtract 1 (for Sync_Seg) and Prop_Seg (calculated in Step 4). From Time_quanta_per_bit. i.e. Phase_Seg1+Phase_Seg2 = Time_quanta_per_bit – 1 – Prop_Seg

   If the remaining number is less than 3, go back to Step 2 and Select a higher clock frequency.

   If the remaining number is an odd number greater than 3 then add one to the Prop_Seg value and recalculate.

   If the remaining number is equal to 3 then Phase_Seg1 = 1 and Phase_Seg2 = 2 and only one sample per bit may be chosen. Otherwise, divide the remaining number by two and assign the result to Phase_Seg1 and Phase_Seg2.

5. Determine SJW as the smaller of 4 and Phase_Seg1.

6. Calculate the required clock tolerance from Equations (8.8) and (8.9) and take the smaller of the two.

   If a reduced oscillator tolerance is required, then consider the following steps:

   If the Phase_Seg1 > 4 T_Q, it is recommended to repeat Steps 2 to 6 with a larger value for the prescaler (i.e. smaller T_Q period).

   If the Phase_Seg1 < 4 T_Q, it is recommended to repeat steps 2 to 6 with a smaller value of prescaler (i.e. larger T_Q period). If the prescaler is already 1, the only option would be to use a higher oscillator frequency.

Some examples are given below to illustrate the process.

Example 8.1

It is required to design a CAN bus system having the following specifications. Calculate the bit timing parameters.

Bit rate = 1 Mbps
Bus length = 10 m
Bus propagation delay = 5 ns/m
Controller TX delay = 80 ns
Controller RX delay = 20 ns
Oscillator frequency = 8 MHz

Solution 8.1

Following the above steps we have:

1. Bus delay = Bus length x propagation delay on the bus or, Bus Delay = 10 m x 5 ns/m = 50 ns.

   From Equation (8.6):

   t_{PROP_SEG} = 2 * (80 ns + 20 ns+ 50 ns) = 300 ns

2. Choosing the prescaler = 1, the CAN controller clock becomes equal to oscillator frequency, i.e. 8MHz. Thus, the time quantum is $\mathbf{T_Q = 125\ ns.}$ The Bit rate is 1 Mbps, i.e. the period is 1 μs. This gives, 1 μs / 125 ns = 8 time quanta per bit (see also Table 8.1).

   Thus, $\mathbf{T_Q = 125\ ns}$ and **TQPB = 8**. The bit rate is 1 Mbps, thus
   $\mathbf{T_{BIT} = 1\ \mu s}$. It also follows that the nominal bit rate is NBR = 1 / T_{BIT} = 1 / 10^{-6} = 10^6 Hz.

3. From Equation (8.7):

   Prop_Seg = NEAREST_HIGHER_INTEGER(300 ns / 125 ns)
   Giving, **Prop_Seg = 3.**

4. Phase_Seg1 + Phase_Seg2 = 8 – 1 – 3

   giving, Phase1_Seg + Phase_Seg2 = 4
   The remainder is greater than 3 and an even number. Thus,
   Dividing by two and assigning to bit segments, we have:

Phase_Seg1 = 2
Phase_Seg2 = 2

5. SJW is the smaller of 4 and Phase_Seg1. Thus, **SJW = 2.**

6. From Equation (8.8):

$\Delta f < t_{SJW} / (20 \times TQPB) = 2 / (20 \times 8) = 0.01250$

i.e.

$\Delta f < 0.01250$

From equation (8.9):

$\Delta f < (MIN(Phase\_Seg1, Phase\_Seg2) / [2 \times (13 \times TQPB - Phase\_Seg2)]$

$= MIN(2, 2) / [2 \times (13 \times 8 - 2]$
$= 2 / 204$

or,

$\Delta f < 0.0098$

The required oscillator tolerance is the smaller one of the two calculated values, which is 0.0098, i.e. **0.98%.** Note that a lower oscillator tolerance can be obtained by increasing the system (and hence the controller) clock frequency.

In this example, the bit sampling point is located at the 6th sample as illustrated in Figure 8.4. This corresponds to time 6 T_Q (see also Table 8.1), or 75% into the bit time.

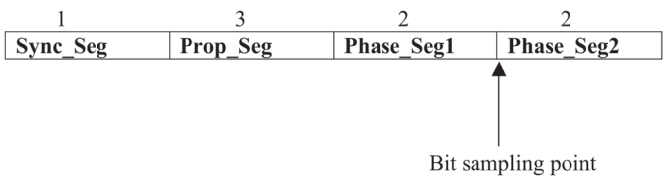


Figure 8.4 The bit sampling point

In summary, the specifications, and the calculated values are:

Specifications:

Bit rate = 1 Mbps
Bus length = 10 m
Bus propagation delay = 5 ns/m
Controller TX delay = 80 ns
Controller RX delay = 20 ns
Oscillator frequency = 8 MHz

Calculation:

Sync_Seg = 1
Prop_Seg = 3
Phase_Seg1 = 2
Phase_Seg2 = 2
SJW = 2
Oscillator tolerance = 0.98%
Oscillator prescaler = 1
Oscillator frequency = 8 MHz
Time quantum (T_Q) = 125 ns
Time quanta per bit (TQPB) = 8

The Sync_Seg, Prop_Seg, Phase_Seg1, Phase_Seg2, SJW, and the prescaler value should be loaded into the CAN controller chip during programming of the chip (we shall see how to do this in Chapter 11).

A slightly different example is given below to illustrate the process again. In this example, the bit rate is chosen to be much smaller.

Example 8.2

It is required to design a CAN bus system having the following specifications. Calculate the bit timing parameters.

Bit rate = 125 kbps
Bus length = 20 m
Bus propagation delay = 5 ns/m
Controller TX delay = 80 ns
Controller RX delay = 120 ns
Oscillator frequency = 8 MHz

Solution 8.2

Following the steps we have:

1. Bus delay = Bus length x propagation delay on the bus or, Bus Delay = 20 m x 5 ns/m = 100 ns.

   From Equation (8.6):

   t_{PROP_SEG} = 2 * (80 ns + 120 ns+ 100 ns) = 600 ns

2. Choosing the prescaler = 4, the CAN controller clock becomes 2 MHz. Thus, the time quantum is **T_Q = 500 ns.** The Bit rate is 125 kbps, i.e. the period is 8 μs. This gives, 8 μs / 500 ns = 16 time quanta per bit (see also Table 8.1).

   Thus, **T_Q = 500 ns** and **TQPB = 16**. The bit rate is 125 kbps, thus **T_{BIT} = 8 μs**. It also follows that the nominal bit rate is NBR = 1 / T_{BIT} = 1 / 8 x 10^{-6} = 125 kHz.

3. From Equation (8.7):

   Prop_Seg = NEAREST_HIGHER_INTEGER(600 ns / 500 ns)
   Giving, **Prop_Seg = 2**.

4. Phase_Seg1 + Phase_Seg2 = 16 – 1 – 2

   giving, Phase1_Seg + Phase_Seg2 = 13
   The remainder is greater than 3 and an odd number. Thus,we add 1 to Prop_Seg and divide the remainder between the phase segment to give:

   Prop_Seg = 3
   Phase_Seg1 = 6
   Phase_Seg2 = 6

5. SJW is the smaller of 4 and Phase_Seg1. Thus, **SJW = 4**.

6. From Equation (8.8):

   $\Delta f < t_{SJW} / (20 \times TQPB) = 4 / (20 \times 16) = 0.0125$

i.e.
$\Delta f < 0.0125$

From equation (8.9):

$\Delta f < (MIN(Phase\_Seg1, Phase\_Seg2) / [2 \times (13 \times TQPB – Phase\_Seg2)]$

$= MIN(6, 6) / [2 \times (13 \times 16 – 6]$
$= 6 / 404$

or,
$\Delta f < 0.01485$

The required oscillator tolerance is the smaller one of the two calculated values, which is 0.0125, i.e. **1.25%.**

A lower oscillator tolerance can be obtained since Phase_Seg1 > 4. This is shown below:

Repeating steps 2 to 6 with a larger prescaler value, we have:

2. Choosing the prescaler = 8, the CAN controller clock becomes 1 MHz. Thus, the time quantum is $\mathbf{T_Q = 1\ \mu s.}$ The Bit rate is 125 kbps, i.e. the period is 8 μs. This gives, 8 μs / 1 μs = 8 time quanta per bit.

   Thus, $\mathbf{T_Q = 1\ \mu s}$ and **TQPB = 8**. The bit rate is 125 kbps,
   thus $\mathbf{T_{BIT} = 8\ \mu s}$. It also follows that the nominal bit rate is
   NBR = 1 / T_{BIT} = 1 / 8 x 10^{-6} = 125 kHz.

3. From Equation (8.7):

   Prop_Seg = NEAREST_HIGHER_INTEGER(600 ns / 1000 ns) = 0.6. Taking the nearest higher integer, we have,
   Prop_Seg = 1.

4. Phase_Seg1 + Phase_Seg2 = 8 – 1 – 1

   giving, Phase1_Seg + Phase_Seg2 = 6
   The remainder is greater than 3 and an even number. Thus,we divide the remainder by two to find the phase segments:

   Phase_Seg1 = 3
   Phase_Seg2 = 3

5. SJW is the smaller of 4 and Phase_Seg1. Thus, **SJW = 3**.

6. From Equation (8.8):

$$\Delta f < t_{SJW} / (20 \times TQPB) = 3 / (20 \times 8) = 0.01875$$

i.e.
Δf < 0.01875

From equation (8.9):

$$\Delta f < (MIN(Phase\_Seg1, Phase\_Seg2) / [2 \times (13 \times TQPB - Phase\_Seg2)]$$

$$= MIN(3, 3) / [2 \times (13 \times 8 - 3]$$
$$= 3 / 202$$

or,
Δf < 0.01485

Taking the smaller of the two, we have 0.01485. i.e. the required oscillator tolerance is 1.485%, which is an improvement over the previous value.

In summary, the specifications, and the calculated values are:

Specifications:

Bit rate = 125 kbps
Bus length = 20 m
Bus propagation delay = 5 ns/m
Controller TX delay = 80 ns
Controller RX delay = 120 ns
Oscillator frequency = 8 MHz

Calculation:

Sync_Seg = 1
Prop_Seg = 1
Phase_Seg1 = 3
Phase_Seg2 = 3
SJW = 3
Oscillator tolerance = 1.485%

Oscillator prescaler = 8
Oscillator frequency = 1 MHz
Time quantum (T_Q) = 1 μs
Time quanta per bit (TQPB) = 8

Example 8.3

It is required to design a CAN bus system having the following specifications. Calculate the bit timing parameters.

Bit rate = 500 kbps
Bus length = 20 m
Bus propagation delay = 5 ns/m
Controller TX delay = 80 ns
Controller RX delay = 120 ns
Oscillator frequency = 16 MHz

Solution 8.3

Following the steps we have:

1. Bus delay = Bus length x propagation delay on the bus or, Bus Delay = 20 m x 5 ns/m = 100 ns.

   From Equation (8.6):

   t_{PROP_SEG} = 2 * (80 ns + 120 ns+ 100 ns) = 600 ns

2. Choosing the prescaler = 2, the CAN controller clock becomes 8 MHz. Thus, the time quantum is **T_Q = 125 ns.** The Bit rate is 500 kbps, i.e. the period is 2 μs. This gives, 2 μs / 125 ns = 16 time quanta per bit (see also Table 8.1).

   Thus, **T_Q = 125 ns** and **TQPB = 16**. The bit rate is 500 kbps,
   thus **T_{BIT} = 2 μs**. It also follows that the nominal bit rate is
   NBR = 1 / T_{BIT} = 1 / 2 x 10^{-6} = 500 kHz.

3. From Equation (8.7):

   Prop_Seg = NEAREST_HIGHER_INTEGER(600 ns / 125 ns) = 4.8. Taking the nearest higher integer, we have,
   Prop_Seg = 5.

4. Phase_Seg1 + Phase_Seg2 = 16 – 1 – 5

   giving, Phase1_Seg + Phase_Seg2 = 10
   The remainder is greater than 3 and an even number. Thus,we divide the remainder by two to find the phase segments:

   Phase_Seg1 = 5
   Phase_Seg2 = 5

5. SJW is the smaller of 4 and Phase_Seg1. Thus, **SJW = 4**.

6. From Equation (8.8):

$$\Delta f < t_{SJW} / (20 \times TQPB) = 4 / (20 \times 16) = 0.0125$$

i.e.
$\Delta f < 0.0125$

   From equation (8.9):

$$\Delta f < (MIN(Phase\_Seg1, Phase\_Seg2) / [2 \times (13 \times TQPB - Phase\_Seg2)]$$

   = MIN(5, 5) / [2 x (13 x 16 – 5]
   = 5 / 406
or,
$\Delta f < 0.0123$

Taking the smaller of the two, we have 0.0123. i.e. the required oscillator tolerance is 1.23%.

In summary, the specifications, and the calculated values are:

Specifications:

   Bit rate = 500 kbps
   Bus length = 20 m
   Bus propagation delay = 5 ns/m
   Controller TX delay = 80 ns
   Controller RX delay = 120 ns
   Oscillator frequency = 16 MHz

Calculation:

Sync_Seg = 1
Prop_Seg = 5
Phase_Seg1 = 5
Phase_Seg2 = 5
SJW = 4
Oscillator tolerance = 1.23%
Oscillator prescaler = 2
Oscillator frequency = 8 MHz
Time quantum (T_Q) = 125 ns
Time quanta per bit (TQPB) = 16

In this example, the bit sampling point is located at the 11th sample as illustrated in Figure 8.5. This corresponds to time 11 T_Q (see also Table 8.1), or 68.75% into the bit time.

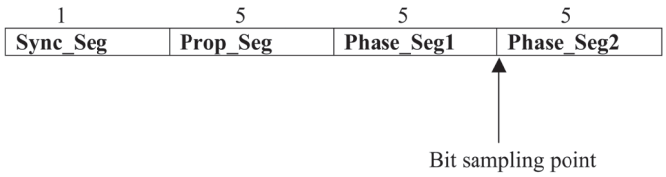


Figure 8.5 The bit sampling point

8.3 Summary

This chapter has explained the important topic of timing and synchronizing data on CAN bus.

It is very important to program the CAN bus controller correctly so that data can be transmitted and received at the expected times. Programming the controller requires selecting a clock frequency, selecting a prescaler value, and calculating the bit segment values to be loaded into the controller.

Three examples are given in this chapter to illustrate the principle of how the controller timing parameters can easily be calculated.

Chapter 9 CAN Bus development tools

There is a wide variety of development tools available that can assist engineers in the development, testing, debugging, and monitoring of the CAN bus, and CAN bus based projects. Some of the development tools are:

- CAN bus development boards
- Microcontroller development boards with built-in CAN bus modules
- High level language libraries for CAN bus functions
- CAN bus analyzers
- CAN bus data loggers
- CAN bus stimulators
- CAN bus simulators

We can divide the CAN bus development tools into two: hardware development tools, and software development tools. In this chapter we are more interested in tools that can be used for the design and development of CAN bus based projects. Such projects usually incorporate a microcontroller.

We shall now look at some examples of each type of tool.

9.1 Hardware Development Tools

Can bus hardware development tools are generally used for the development of CAN bus based projects using microcontroller systems. The development of such projects generally require knowledge of electronics, and computer programming skills, preferably using a high level language (e.g. C). In this section we shall be looking at some of the hardware development tools.

9.1.1 The RCDK8C CAN Development Kit

The **RCDK8C CAN Development Kit** by Renesas (source: http://am.renesas.com) shown in Figure 9.1 can be used in the development of CAN bus based projects.

Figure 9.1 RCDK8C CAN development kit

The kit is distributed with the following parts:

- 2 Starter kit boards pre-programmed with demonstration boards
- DC power supply
- LCD
- *CAN Sniffer* for monitoring CAN bus traffic
- In system programmer and debugger
- Cable assembly for USB and Can bus
- CDROM including a quick start guide, drivers, device manuals, and sample programs.

Figure 9.2 shows how two nodes can be connected via the CAN bus using this kit. The CAN Sniffer shown is connected to monitor the bus activities.

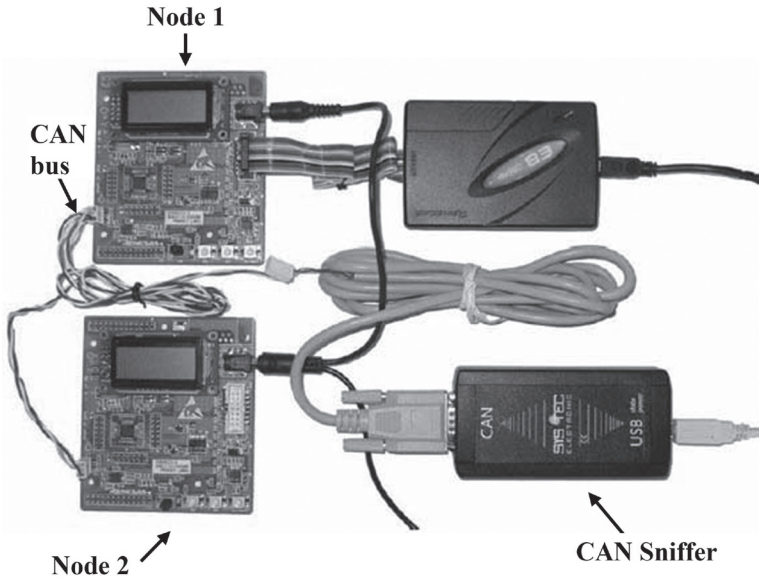


Figure 9.2 Connecting two nodes

9.1.2 CCS CAN Bus Development Kit

The Custom Computer Services (CCS) Can Bus Development Kit (source: *http://www.ccsinfo.com*) is shown in Figure 9.3. The kit enables users to develop CAN bus based projects using the PIC18 family of microcontrollers. The supplied kit has 4 nodes on a CAN bus. The PCWH integrated software development environment developed by the company, including a powerful C compiler is included with the kit.

The CAN bus board supplied with the kit has following specifications:

- PIC18F4580
- PIC16F876A
- 30 I/O Pins
- MCP2515
- Two MCP25050
- Three Potentiometers
- Nine LEDs
- 7-Segment LED
- Two RS-232 Ports
- RS-232 Level Converter
- ICD Jack

The first node is made from the PIC18F4580 microcontroller which includes a built-in CAN module.

The second node is made from PIC16F876A microcontroller connected to an external MCP2515 type controller.

The third and the fourth nodes are made from the MCP25050 stand-alone CAN expanders which have been pre-programmed by CCS to respond to specific message IDs. One of these nodes in connected to a potentiometer, three LEDs, and three puch-button switches. The other node is connected to a 7-segment LED display.

Figure 9.3 CCS CAN bus development kit

9.1.3 CAN MicroMOD Development Kit

The MicroMOD CAN development kit (source: http://www.peak-system.com) includes a PC CAN interface, a MicroMode Evaluation board (see Figure 9.4), a MicroMod CPU board, power adapter, and 3 ft CAN cable. In addition, a CDROM is supplied with MicroMod configuration software. User can create their CAN bus network and experiment with the kit. No embedded programming skills are required. Configuration data is sent to the module via CAN bus and each individual node over the bus can be enabled to read or send data.

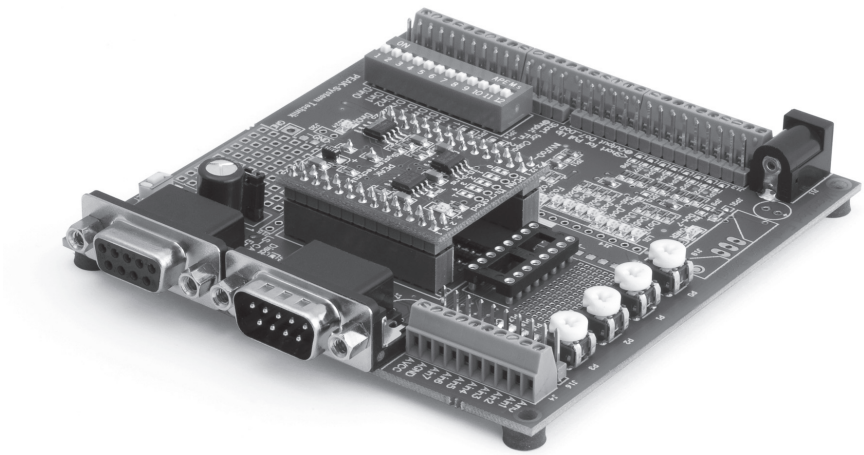


Figure 9.4 CAN MicroMod Evaluation board

9.1.4 Starterkit MB91360

The Starterkit MB91360 (see Figure 9.5) Evaluation board (source: http://emea.fujitsu.com/semiconductor) is supplied with Windows based development software, and it can be used in CAN based project development applications. The board is equipped with a CAN transceiver, 2x16 character LCD, and 8 LEDs that can be used during the development.

9.1.5 BASIC-Tiger CAN-Bus Prototyping Board

This is a Eurocard size CAN bus prototyping board (source: http://www.wilke-technology.com) with the following specifications (see Figure 9.6):

- Socket for the computer
- 50-pin extension connector
- 2 serial ports
- Connector for LCD
- Buttons, keys and switches
- 3 analog inputs (configurable)
- CAN bus connection
- Patch area
- Power supply

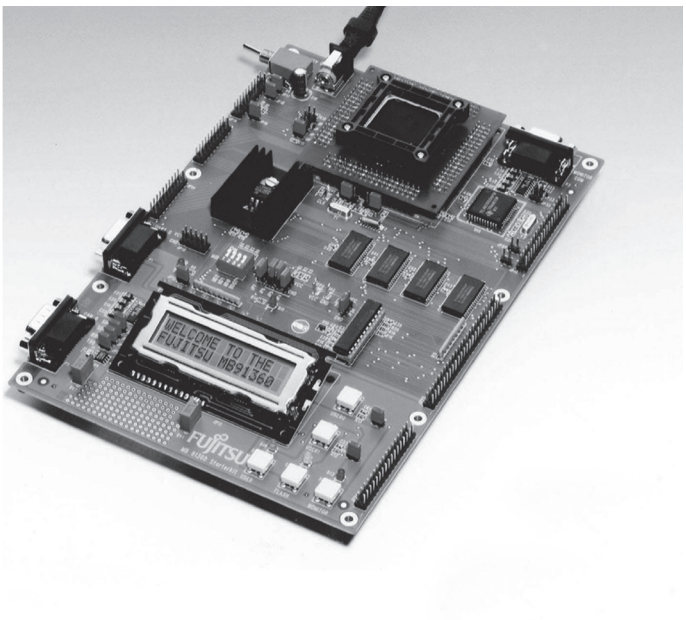


Figure 9.5 MCBXC167 Evaluation board

9.1.6 MikroElektronika CAN Communication Kit

The CAN Communication Kit developed by mikroElektronika (source: http://www.mikroe.com) is shown in Figure 9.7. This kit is based on company's highly successful EasyPIC6 microcontroller development board. The popular mikroC Pro high level programming language compiler is included in the package.

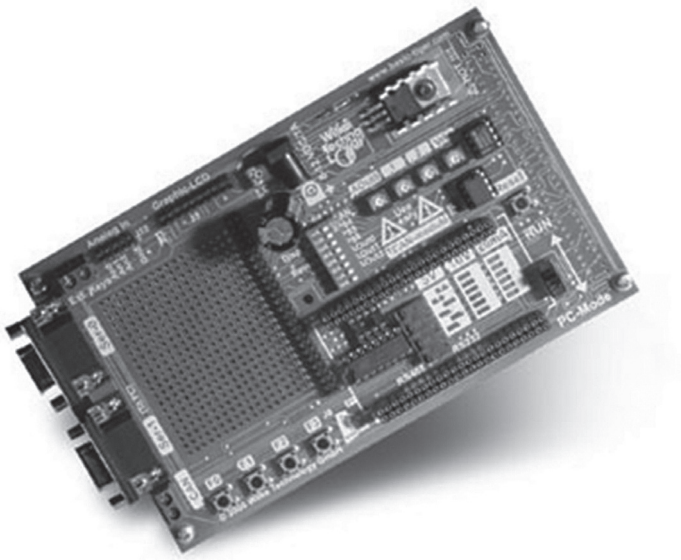


Figure 9.6 BASIC-Tiger CAN-Bus Prototyping Board

The kit has the following specifications:

- 2 x EasyPIC6 development boards
- mikroC Pro compiler
- 2 x CANSPI boards
- 2 x SmartPROTO boards
- 2 x EasyConnect boards
- 2 x Character LCDs
- 2 x DS1820 temperature sensor chips
- 2 x Graphic LCD
- Twisted CAN bus cable (2 m)
- USB cable
- CDROM and printed manual including drivers and example programs

Figure 9.7 mikroElektronika CAN communication Kit

9.1.7 mikroElektronika CAN-1 Board

This board (see Figure 9.8) from mikroElektronika (source: http://www.mikroe.com) is a CAN bus development tool for microcontrollers with integrated CAN modules. CAN-1 board basically consists of a MCP2551 CAN transceiver chip with some jumpers and switches.

9.1.8 mikroElektronika CANSPI Board

This board (see Figure 9.9) from mikroElektronika (source: http://www.mikroe.com) is a CAN development tool for microcontroller with SPI interface. CANSPI board basically consists of a MCP2515 CAN controller chip, a MCP2551 CAN transceiver chip, and some jumpers and switches.

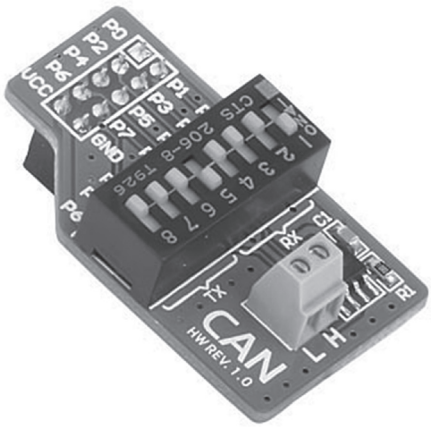


Figure 9.8 CAN-1 Board

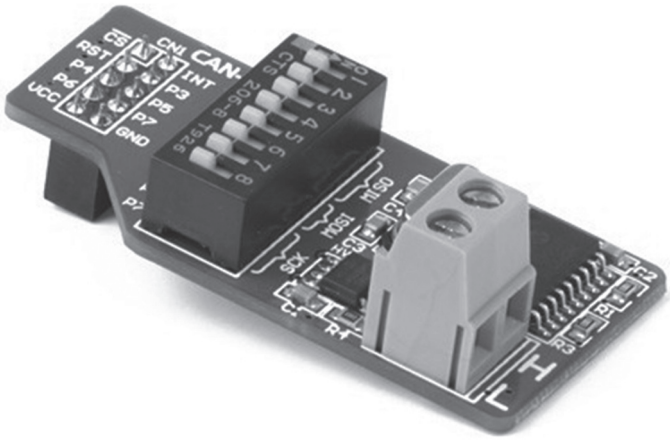


Figure 9.9 CANSPI Board

9.2 Software Development Tools

CAN bus software development tools are generally used for the development of CAN bus based programs for microcontrollers. CAN bus controllers are almost always used with microcontrollers, and the software tools are used to develop, test, and debug the software for these microcontrollers.

CAN bus software development tools are basically language compilers and software debug aids. Almost in all CAN bus projects a high level programming language is used to program the microcontroller.

There are several high level language compilers developed by different companies that can be used to develop CAN bus based projects. Some companies provide CAN bus software library functions that make CAN bus programming a much easier task. In this book we shall be using the mikroC language compiler, developed by mikroElektronika (source: http://www.mikroe.com), and described in detail in Chapter 11. CAN bus project examples are also given in Chapter 11 to demonstrate how to use the CAN bus software library functions.

9.3 CAN Bus Analyzers

CAN bus analyzers are available with varied functionalities and prices. Basically, a bus analyzer consists of a small hardware device (called the analyzer hardware), and a dedicated software (called the analyzer software, usually runs on a PC). One end of the analyzer hardware is attached to the CAN bus as a node, while the other end is usually connected to a PC via the USB port. Once the analyzer software is activated the analyzer hardware starts collecting all the frames sent over the bus, with time stamping. The collected data can then be analyzed offline, and any data transmission errors or timing errors can easily be detected. Bus analyzers can be invaluable tools during the development of a new CAN bus based project. In addition, these devices can be very useful as teaching aids where students can analyze and learn the collected frame structures and timing details.
In this section we shall be looking at some popular CAN bus analyzers available commercially.

9.3.1 Microchip Inc CAN Bus Analyzer

The Microchip CAN bus analyzer (source: http://www.microchip.com) is a low cost tool that can be used during the development and debugging of high speed CAN network. The tool supports CAN 2.0B standard, and comes with all the necessary hardware and software (see Figure 9.10). The basic specifications of this device are:

- Supports CAN 2.0B
- PC User Interface for functions such as configuration, trace, transmit, filter, log etc
- Direct access to CAN H and CAN L, CAN TX and CAN RX signals for debugging.
- Flexible CAN bus interface options i.e. standard DB9 connector or screw terminals.
- Software control of termination resistance and LED display for status, traffic, and BUS error.

9.3.2 CAN Bus X-Analyser

This is a compact device that can be used for the analysis and stimulation of CAN and similar networks. The device records messages and error frames from the CAN bus. The recorded data is time stamped in microsecond resolution. The collected data can be displayed and analyzed in a time chart (see Figure 9.11). The device works on a Windows compatible PC. A suitable CAN bus interface card is needed.

Figure 9.10 Microchip Inc CAN Bus Analyzer

Figure 9.11 CAN Bus X-Analyser

9.3.3 PCAN Lite

This is a software package (source: http://www.computer-solutions.co.uk) allowing the user to view messages using the included PCAN View (see Figure 9.12) on the CAN bus and to create messages. The software runs under Windows PC. All data is displayed in Hex and errors such as overrun and baud rate problems are reported. A CAN interface is required to operate the software. PCAN Lite is supplied free of charge with company's PEAK CAN interfaces.

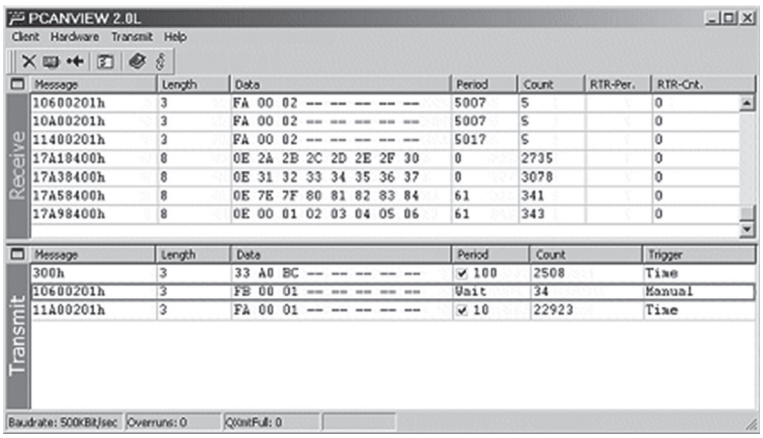


Figure 9.12 PCAN View

9.3.4 PCAN Explorer

The PCAN Explorer program shows what is happening on the CAN bus. It can also be used for controlling and interacting with CAN bus systems, for debugging the system, or for a PC program driving it.

Some of the specifications of PCAN Explorer are:

- Shows all received messages in a receiving list containing message ID, length and data bytes.
- Indication is given of received remote-frames, number and receiving interval.
- Any number of messages can be put into a transmit list to be sent at fixed intervals, manually , from function keys or as the answer to a remote-frame request.
- Errors on the CAN-bus and of the controller are indicated.
- Each CAN message ID can be given a unique name (eg. speed) which is then used in place of the ID when messages are logged making complex systems easy to debug.
- An extensive Visual Basic Script Macro language can be used to test or simulate CAN drivers.
- Add-ins (written in C++, VB or Borland) allow the users to automate tasks in PCAN Explorer by adding commands to perform these tasks, by adding toolbar buttons to carry out these commands, and by responding to PCAN Explorer events.

Some of the specifications of the data logger are:

- Variable buffer size.
- Errors can be logged.
- The logged data can be saved in Excel format so that the data can be analyzed offline using Excel.
- Message types to be logged can be selected by the user

A data logging example is shown in Figure 9.14.

9.3.5 CAN Physical Layer Analyzer (CANwatch)

The CAN Physical Layer Analyzer (source: http://www.ems-wuensche.com), or CANwatch, shown in Figure 9.15 is an analyzer supporting easy error detection during installation and operation of CAN networks. Errors on the physical layer of CAN bus can not be detected by standard analyzers and CANwatch could therefore could be a very useful tool during the setup of a new network or during the addition of a new node to an existing bus.

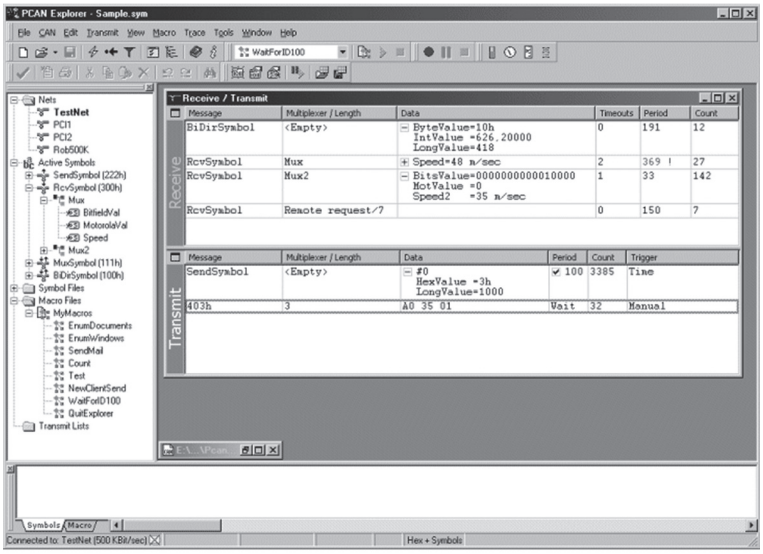


Figure 9.13 PCAN Explorer

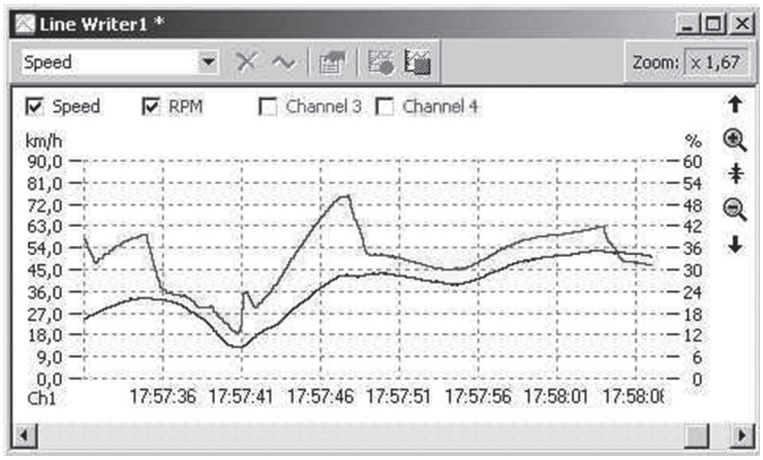


Figure 9.14 Data logging example

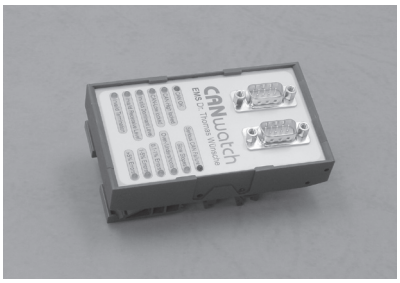


Figure 9.15 CAN Physical layer analyzer

9.3.6 CAN-BUS-Tester

CAN-Bus-Tester (source: http://www.ixxat.com) is a device used for the commissioning analysis, monitoring, troubleshooting, and maintenance of CAN bus systems. The device (see Figure 9.16) can be used for both CAN2.0A and CAN2.0B systems. The CAN-Bus-Tester has automatic baud rate detection and this enables the device to be quickly and easily connected to the system to be analyzed. With the CAN-Bus-Tester, the bus wiring can be tested while the system is being set-up and its transmission properties can be saved. The device is connected to a Windows PC via USB, and operated with the supplied software.

9.3.7 LeCroy Bus Analyzer

The LeCroy bus analyzer (source: http://www.lecroy.com) is a conventional oscilloscope with additional hardware and software that can decode CAN bus data. The device (see Figure 9.17) can be used to investigate the data sent over a CAN bus, and has the ability to view additional in-circuit electrical signals, such as sensors, actuators, transients etc that influence the CAN bus. In addition, the device has the capability to decode up to four different CAN busses at the same time. In addition to CAN bus, the device can be used to analyze a variety of different protocols.

Figure 9.16 CAN-BUS-Tester

Figure 9.17 LeCroy Bus Analyzer

9.4 An Example Using a CAN Bus Analyzer

In this section we shall be looking at the details of a CAN bus analyzer device. The device will be connected to a CAN bus and the frames sent over the bus will be collected and analyzed. Readers should find this section useful as it will demonstrate the actual data being sent over the CAN bus.

The analyzer device selected in this section is the LAP-C 16032 Logic Analyzer from Zeroplus Logic Cube (source: http://www.nkcelectronics.com). Figure 9.18 shows a picture of this analyzer.

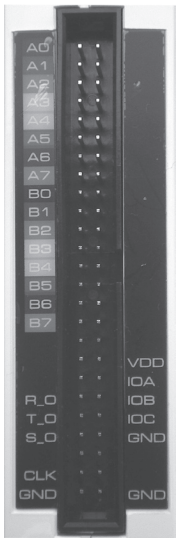


Figure 9.18 LAP-C 16032 logic analyzer

The LAP-C 16032 is a standard 16-channel PC-based logic analyzer with added protocol analyzer capabilities. The standard package includes analyzer plug-ins for protocols such as I2C, UASRT, SPI, and 7-segment LED. Additional protocol analyzers, such as CAN, USB, LIN, and so on can be purchased. The logic analyzer connects to a PC via its USB port, supporting the USB 2.0 speed. Power is received from this USB port. The LAP-C series features models with 16 and 32-channels and various sizes of data capture memory.
In the example given in this section, the CAN 2.0B protocol plug-in has been installed to the basic analyzer. Figure 9.19 shows the side view of the device where cables are plugged in. Note that only PORT A and PORT B are used in the basic model LAP-C 16032.

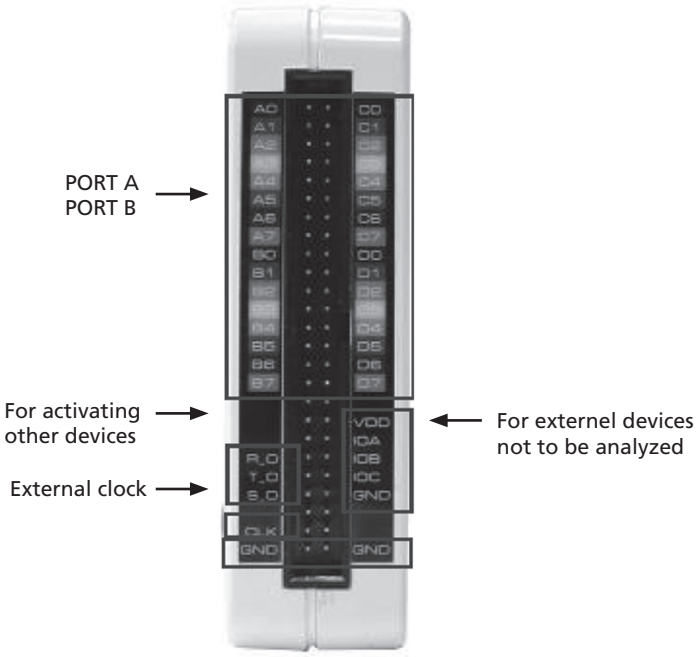


Figure 9.19 Side view of the device

Figure 9.20 shows how the analyzer can be connected to a PC and to a board to be analyzed. The logic analyzer takes its power from the PC and communicates via the USB port. Ports of the logic analyzer should be connected to the circuit under test as required. It is important to make sure that the ground pin of the logic analyzer is connected to the ground pin of the circuit under test.

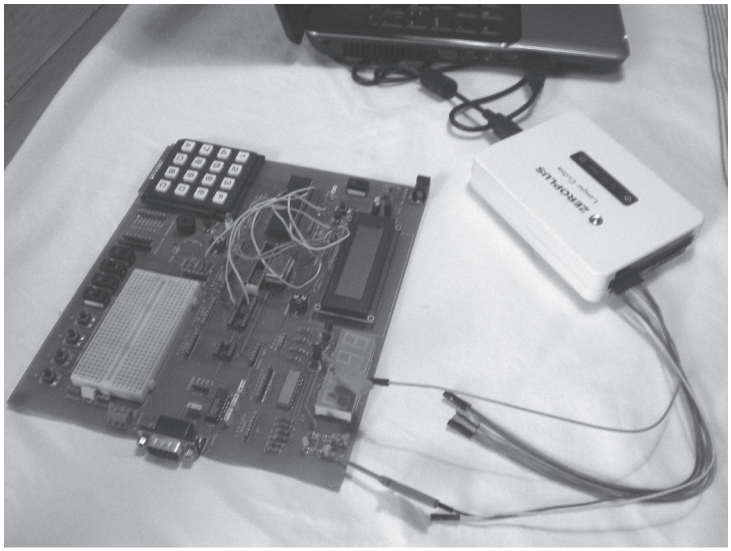


Figure 9.20 Connecting the analyzer to PC and to a board to be analyzed

Figure 9.21 shows the screen layout when the analyzer software is activated. The screen consists of 9 sections:

Section 1 is the Main Menu bar.

Section 2 is the Tool Bar including the commonly used options.

Section 3 is the Information Bar which displays information about the waveforms being displayed.

Section 4 is the Waveform display/listing ruler which shows the time information of the waveforms being displayed.

Section 5 shows the channels names. The colours in the waveforms match the channel colours in this section.

Section 6 is the Trigger Column, enabling the user to set triggering conditions.

Section 7 is the Filter Column, allowing the user to set filters.

Section 8 is the actual display area.

Section 9 is the status area.

CAN bus protocol analyzer is an option and should be purchased and installed before it can be used. One of the port pins of the logic analyzer (e.g. A0) should be connected to the CAN bus under test together with the ground pin. Then the captured data should automatically be decoded in CAN frame format and all the fields should be shown on the screen. Figure 9.22 is a typical data capture from the CAN bus using the analyzer. Various fields of the frame are in different colours for easy identification, and these colours can be changed if desired. It is also possible to zoom-in or zoom-out the fields for easy viewing. For example, Figure 9.23 shows part of the screen with the fields zoomed in, and in Figure 9.24 only the data part is zoomed-in.

In the example shown in Figures 9.22, 9.23, and 9.24 the fields have the following values:

Identifier: 0x05
RTR: 0
IDE: 0
DLC: 3
DATA: 0x30 0x04 0xC3

The data is then followed by the CRC field.

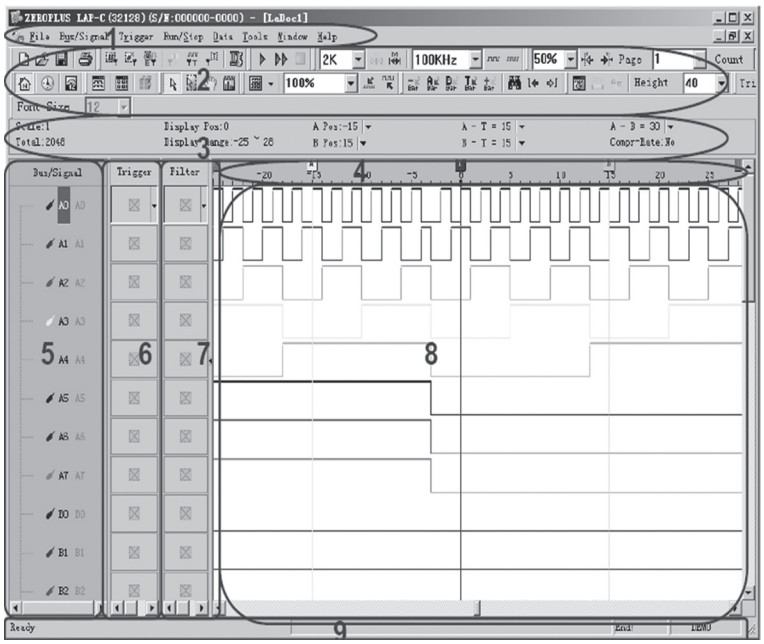


Figure 9.21 The logic analyzer screen layout

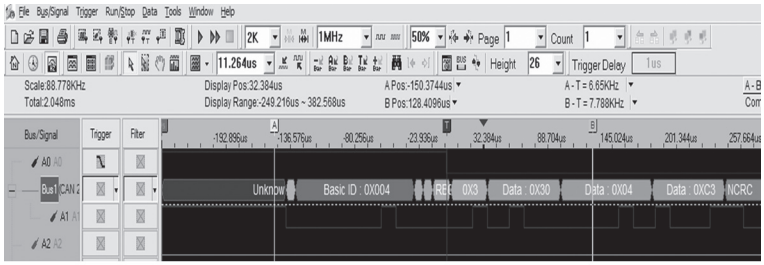


Figure 9.22 Example CAN bus data capture

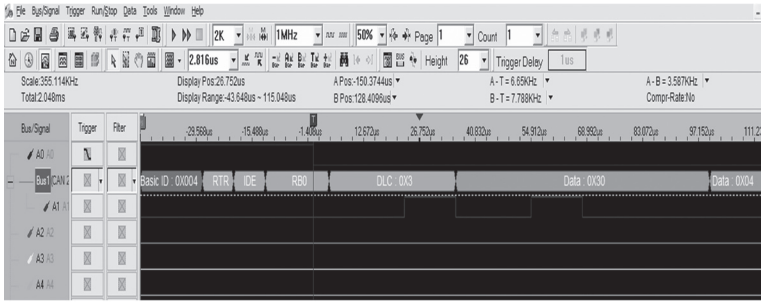


Figure 9.23 Screen zoomed-in in Figure 9.22

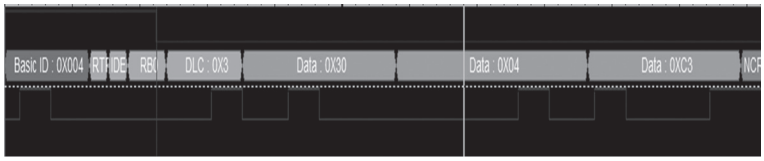


Figure 9.24 Only the data section zoomed-in

9.5 Summary

This Chapter has described the important topic of development tools. Without a development tool it is very hard to develop a project, especially a complex project using the CAN bus.

The hardware tools consist of hardware development boards. It is shown that some companies provide integrated development tools where a complete CAN bus system is offered. Users can simply create a CAN bus by connecting the supplied hardware together. In addition, high level language compilers are also provided so that users can develop, test, and debug their programs easily.

CAN bus analyzers are an important part of developing CAN bus based projects. With the aid of these analyzers, users can monitor and log the data moving over the CAN bus. For example, during the development of a project, users can monitor and investigate the various CAN bus frame structures and timings in order to make sure that the correct data is being transferred over the bus at the correct times. In addition, analyzers can be used to learn the structure of CAN bus frames. For example, the data over the bus can be logged and then analyzed offline to learn the frame structures and the timing details.

In addition to describing the various development tools, in this section, a typical CAN data capture is shown using a logic analyzer with CAN protocol option. Such tools are extremely important during the development process as they show exactly what the data is on the CAN bus at any moment in time

Chapter 10 Can Bus Controllers

In general, a CAN bus controller provides the interface between a microcontroller and the CAN bus. A CAN bus controller consists of two parts:

- The basic CAN bus controller
- CAN Transceiver

The CAN transceiver provides the physical interface to the CAN bus. It consists of bus driver and receiver logic. The controller is programmable and it contains the error detection logic, acceptance filters, masks, and buffers. The controller receives CAN data from the transceiver, accepts or rejects this data, and if accepted transfers the data to a microcontroller system. Similarly, the controller sends data to the CAN bus through the CAN transceiver.

Some high-end microcontrollers have built-in transceiver circuits and hence they can directly be connected to CAN bus. Figure 10.1 shows such a microcontroller system. If a microcontroller has no built-in CAN bus transceiver then an external transceiver chip can be connected to its I/O pins. Figure 10.2 shows a microcontroller system where a transceiver chip is used to interface to the CAN bus.

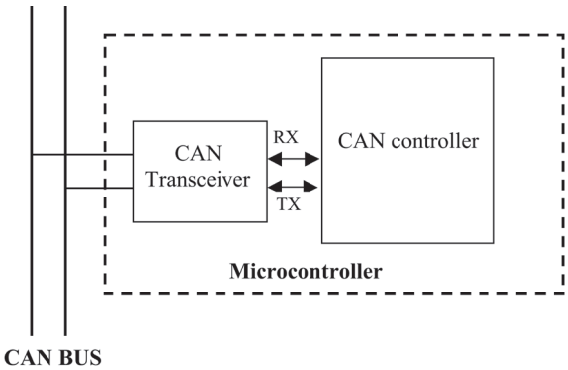


Figure 10.1 Microcontroller with built-in transceiver

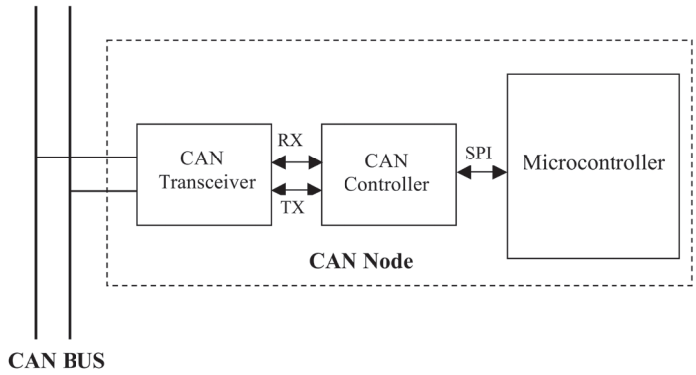


Figure 10.2 Microcontroller with external transceiver

In this Chapter we shall be looking at the basic structure of various CAN transceivers and CAN controllers available in the market.

10.1 The Basic Structure of a CAN Transceiver

As mentioned earlier in the Chapter, a CAN transceiver provides the actual physical interface to the CAN bus. A typical CAN transceiver consists of the following:

- TX and RX pins to connect to the controller
- CAN_H and CAN_L pins to connect to CAN bus
- Power supply pins

Figure 10.3 shows the pin configuration of MCP2551,which is a typical CAN transceiver from Microchip Inc. (source: http://www.microchip.com).

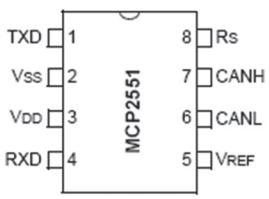


Figure 10.3 Pin configuration of MCP2551 CAN transceiver

The internal structure of the MCP2551 chip is shown in Figure 10.4. Notice the open collector output of pin CAN_H. The chip supports operation up to 1 Mbps and is suitable for both 12V (e.g. cars) and 24V systems. MCP2551 includes short-circuit protection circuitry, and automatic thermal shutdown protection. Protection against high voltage transients is also provided. Low power standby operation provides energy saving in power critical applications. Up to 112 nodes can be connected to the CAN bus.

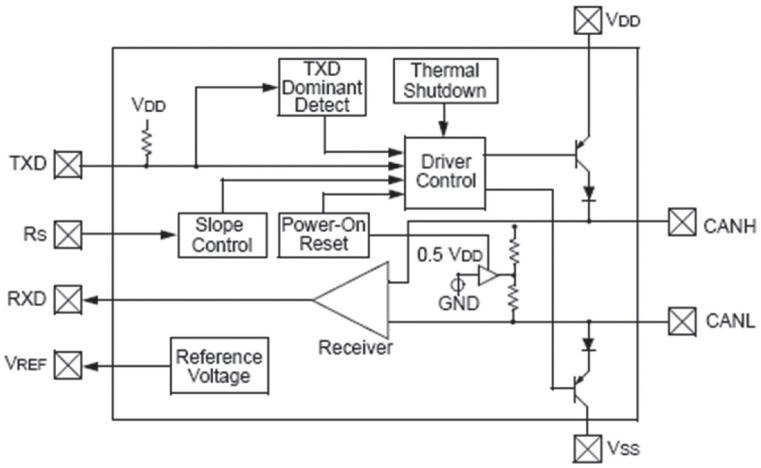


Figure 10.4 Internal structure of MCP2551 CAN transceiver

Another popular CAN transceiver chip is the PCA82C250 (for 5V systems), and PCA82C251 (for 24V systems) manufactured by Philips Semiconductors (source: http://www.semiconductors.philips.com). The structure of this transceiver is very similar to MCP2551.

10.2 The Basic Structure of a CAN Controller

CAN controllers provide the intelligent interface between a CAN transceiver and a microcontroller system (see Figure 10.2). A CAN controller consists of the following parts:

- CAN protocol identifier
- Acceptance filters
- Control registers
- Transmit buffer
- Receive buffer
- Host interface
- CAN bus Transceiver interface

The first CAN controller was called "Basic" CAN interface and it was implemented by the Philips 82C200 chip. This chip provided very basic and limited functionality. "Basic" CAN interface (still available today) only offers a limited number of receive buffers and acceptance filters (typically 1 to 3). If a node using such a controller needs to listen to a large number of different messages with different CAN message identifiers, the microcontroller has to receive every message and check its message identifier to decide whether to accept or reject the message.

Figure 10.5 shows the block diagram of the "Basic" CAN interface chip (or the CAN controller).

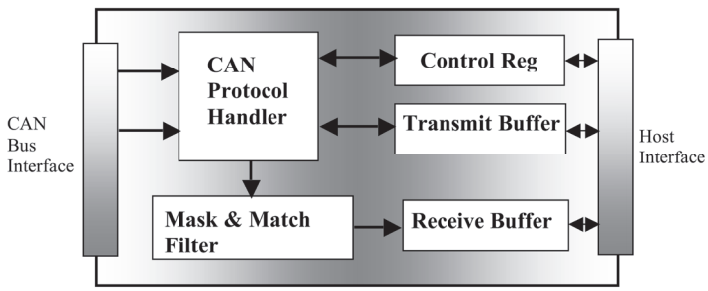


Figure 10.5 Block diagram of a "Basic" CAN controller

As the complexity of the devices attached to the CAN bus has increased, the functionality provided by the "Basic" chip was not enough, and today over 20 manufacturers offer sophisticated CAN controller chips. The new state of the art CAN controller chips are also called "Full" CAN controller chips. The first "Full" CAN controller chip was the Intel 82526. This chip is more complex than the "Basic" controller. As shown in Figure 10.6, a number of message objects are provided with Match IDs, where each message object is bi-directional and can receive or transmit. Each message object

has one buffer and a Match ID. The message IDs can be set to listen for a unique message by programming its Match ID. This setup is efficient as long as the message objects are enough to listen to all required types of messages. One problem with the early "Full" CAN controller is that there is only one buffer for each message object and the buffer contents can be destroyed if a new message comes in before the current message is read. Newer controllers include FIFO buffers to overcome the message override problems.

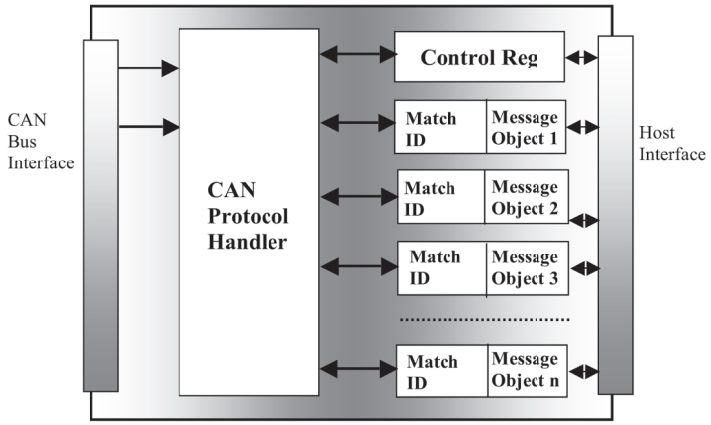


Figure 10.6 Block diagram of the early "Full" CAN controller

One of the recent popular stand-alone CAN controller chips which also contains a FIFO buffer is the SJA1000 from Philips Semiconductors (source: http://www.semiconductors.philips.com). This chip supports the CAN 2.0B protocol specification meaning that extended data frames (29-bit message identifier) can be transmitted over the CAN bus. As shown in Figure 10.7, this chip contains the following modules:

- Interface Management Logic
- Bit timing Logic
- Error Management Logic
- Bit Stream Processor
- Transmit Buffer
- Receive Buffer
- Acceptance Filter

Interface Management Logic receives commands from the microcontroller system and controls the internal CAN registers.

Bit Timing Logic monitors the CAN bus and handles the bus related bit timing functions.

Error Management Logic is responsible for error confinement.

Bit Stream Processor controls the data stream between the transmit buffer and the CAN bus. This module also provides programmable time segments to compensate for the propagation delay times.

Transmit Buffer is an interface between the host microcontroller and the Bit Stream Processor and it can store a complete message for transfer over the bus.

Receive Buffer is an interface between the acceptance filer and the host microcontroller. A FIFO is included within the receive buffer to avoid received data overrun.

Acceptance filter compares the message identifier with the acceptance filter register contents and decides whether or not to accept a message.

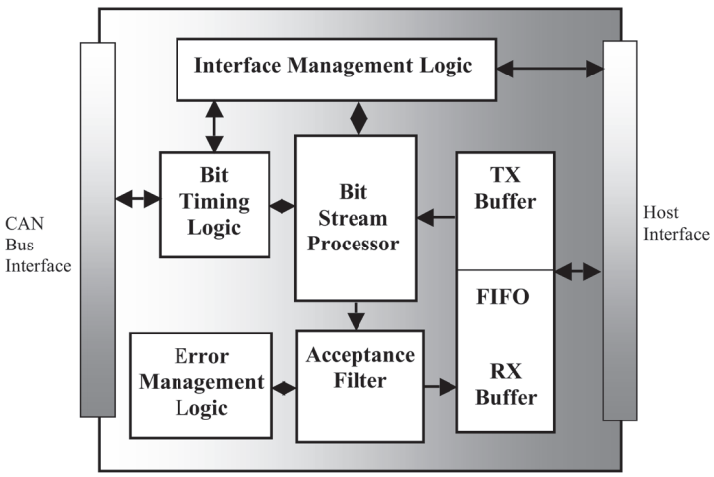


Figure 10.7 The block diagram of the SJA1000 controller

10.3 The MCP2515 CAN Controller

Another popular CAN controller chip is from Microchip Inc, called MCP2515. This is a 1 Mbps controller compatible with the CAN 2.0B protocol specification. This chip has the following specifications:

- Two receive buffers
- Three transmit buffers
- Two acceptance masks
- Six acceptance filters
- SPI interface to the microcontroller
- Output clock pin
- Interrupt pin
- SOF signal is available for monitoring
- Low power operation

Block diagram of the MCP2515 is shown in Figure 10.8. The chip contains the following modules:

- CAN controller
- Control logic
- SPI interface logic
- Control and interrupt registers

The CAN controller module consists of the CAN protocol engine, TX and RX buffers, acceptance masks and filters. Messages detected on the bus are checked for errors. If there are no errors, the message identifier is compared with the user defined filters to see if there is a match, and if a match is detected the message is accepted by the module and is ready to be read by the microcontroller. The CRC and bit timing are also controlled by the CAN controller module.

Communication between the microcontroller and the MCP2515 is using the SPI protocol, and this is handled by the SPI interface logic.

The control logic is responsible for controlling all the internal operations of the chip.

Control and interrupt registers are under the control of the control logic and they provide the interrupt capability to the chip.

The MCP2515 chip is controlled from the Serial Peripheral Interface (SPI) bus available on many microcontrollers. Commands and data are sent to the chip via the SI pin, with the rising edge of the clock CLK input. Similarly, data is out from pin SO on the falling edge of the clock. The chip select input, CS, must be low for normal operation of the device. The registers in the device must be programmed before the device can be used in a microcontroller circuit.

10.4 Microcontrollers with Built-in CAN Bus Modules

Some microcontrollers have built-in CAN bus controllers. The only chip needed to connect them to the bus is a CAN bus transceiver chip (see Figure 10.1).

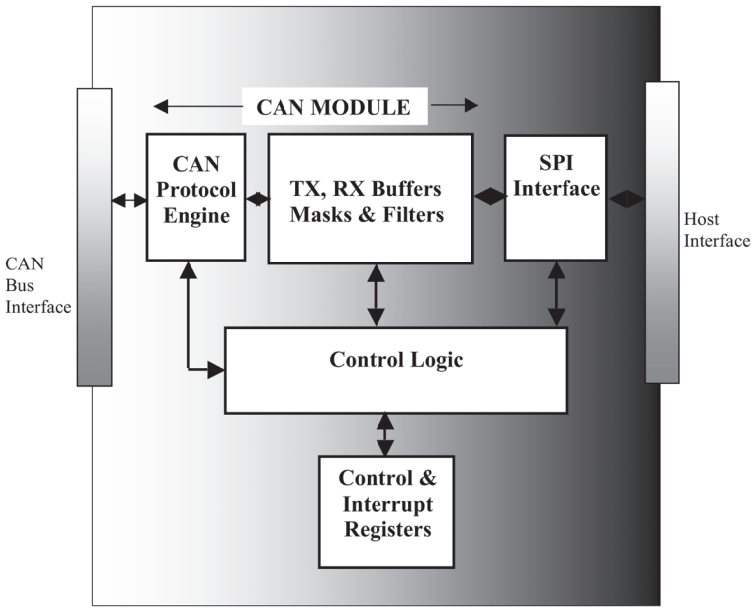


Figure 10.8 Block diagram of the MCP2515

A list of some of some of the microcontrollers with built-in CAN controllers is given below:

- P8xC592
- P8xCE598
- PIC18F258
- PIC18F2580
- PIC18F2680
- PIC18F4480
- PIC18F8585
- PIC18F8680
- STR710FZ2
- DS80C410
- TMS320F241
- and so on.

In general, it is much easier to use a microcontroller with a built-in CAN module when it is required to design a CAN bus based project. The advantages of this approach are that the circuit is less complex, it is easier to implement the circuit on a PCB, the power consumption is less, there are no compatibility issues, and in general it is easier to program the CAN bus modules when the module is integrated inside a microcontroller.

10.5 Summary

This Chapter has described the CAN bus transceivers and controllers. A CAN bus transceiver makes the actual physical connection to the CAN bus. It receives messages and passes them to the CAN controller. Similarly, messages received from the controller are received by the transceiver and sent on the bus.

The CAN bus controller provides the intelligence between a microcontroller and the CAN bus. The controller receives messages, checks for errors, and then accepts or rejects these messages based on its acceptance filters. Messages received from the microcontroller are sent to CAN transceiver with error checking codes for transmission over the bus.

Selecting a CAN controller for a CAN bus based project is not an easy task. As mentioned earlier in the Chapter, it is in general much better to choose a microcontroller with a built-in CAN module. Most of present day CAN controllers are "Full" controllers offering highly complex and sophisticated operations, with several receive and transmit buffers, several acceptance filters, and several acceptance masks. These controllers are very efficient as long as the types of messages a node needs to listen to is smaller than the number of acceptance filters implemented on the controller.

Chapter 11 Microcontroller Based CAN Bus Projects

In this Chapter we shall be designing microcontroller based projects using the CAN bus as the communications medium. The aim of the Chapter is to show how CAN bus based real projects can be developed using state of the art microcontrollers and programming techniques.

The hardware design, flow diagram, complete program listing, and the description of the projects will be given in detail. The projects in this book will be based on the highly popular PIC18F microcontroller series. But, it shouldn't be a difficult task to implement the projects on different types of microcontrollers from different manufacturers, since the complete design of the projects will be given.

At the end of this Chapter you should be able to develop your own CAN bus based projects, or to interface to an existing CAN bus and increase the functionality on the bus, or monitor the data on the bus by developing your own programs.

Before going into the design details of the projects, it is worthwhile to review the basic principles and the architecture of PIC® microcontrollers, and their programming techniques.

The projects in this book will be using the popular mikroC "C" compiler (source: http://www.mikroe.com) developed by mikroElektronika. Knowledge of the C programming language will be useful, and familiarity with at least one member of the PIC® microcontroller series will be an advantage. Knowledge of assembly language programming is not required since all the projects in the book are based on the C language. Later in the Chapter, an introduction will be made to the mikroC language and a simple example program will be given to show how a microcontroller program can be developed, compiled, and then tested.

11.1 What is a Microcontroller ?

A microcontroller is a single-chip computer. *Micro* suggest that the device is small, and *Controller* suggests that it is used in control applications. Another term used for microcontrollers is *embedded controller*, since most of the microcontrollers are built into (or embedded in) the devices they control. For example, a microwave oven has an embedded controller that controls all operations of the oven. A modern car has a large number of embedded controllers that control various functions in the car, such as the temperature, pressure, engine speed, and so on.

A microcontroller differs from a microprocessor in many ways. The main distinction is that a microprocessor requires several other external components for its operation, such as program memory, data memory, input-output devices, clock, timer circuits, interrupt handling circuits, and so on. A microcontroller on the other hand, has all the support chips incorporated inside its single chip. As a result, compared to microprocessor systems, a microcontroller requires less support components, occupies smaller foot print, consumes much less power, and is much cheaper.

Microcontrollers have traditionally been programmed using the assembly language of the target processor. Although the assembly language is fast, it has many disadvantages, such as the difficulty to develop and then maintain large and complex projects. In addition, microcontrollers manufactured by different companies have different instruction sets, and as a result, it is very hard to transport a developed program to another microcontroller.

Nowadays almost all microcontroller projects use a high level programming language, such as BASIC, PASCAL, or C. High-level languages are much easier to learn than the assembly level languages, and complex projects can be developed in much shorter time. The testing and maintenance of high level languages are also easier.

In this book we shall be using the mikroC programming language. mikroC is a popular C language, developed specifically for microcontrollers. As we shall see in later sections, mikroC language supports a large number of built-in libraries for the development of projects based on protocols such as RS232, RS485, I^2C, USB, CAN bus, SD card, Ethernet, Compact Flash card and so on.

In theory, a single chip is sufficient to have a running microcontroller system. In practical applications, however, additional components may be required so that the microcontroller can interface with its environment. With the advent of the microcontrollers, the development time of complex electronic projects has reduced to several hours instead of several weeks.

11.2 The PIC18F Microcontroller Series

Microchip Inc. (source: http://www.microchip.com) has developed PIC18F microcontroller series for use in high density, and complex applications. The PIC18F microcontrollers offer cost-efficient solutions for general purpose applications written in C that use a real-time operating system and require a complex communication protocol stack such as TCP/IP, CAN, USB, or ZigBee. PIC18F microcontrollers provide flash program memories in sizes from 8 to 128 Kbytes, and data memories from 256 bytes to 4 Kbytes, operating at a range of 2.0V to 5.0V, at speeds from DC to 40 MHz.

The basic features of the PIC18F microcontroller series are:

- 77 instructions
- PIC16 source code compatible
- Program memory addressing up to 2Mbyte
- Data memory addressing up to 4Kbytes
- DC to 40 MHz operation
- 8 x 8 hardware multiplier
- Interrupt priority levels
- 16-bit wide instructions, 8-bit wide data path
- Up to two 8-bit timer/counters
- Up to three 16-bit timer/counters
- Up to four external interrupts
- High current (25 mA) sink/source capability
- Up to five capture/compare/PWM modules
- Master synchronous serial port module (SPI and I2C modes)
- Up to two USART modules
- Parallel slave port (PSP)
- Fast 10-bit analog-to-digital converter
- Programmable low voltage detection (LVD) module
- Power-on reset (POR), power-up timer (PWRT), and oscillator start-up timer (OST)
- Watchdog timer (WDT) with on-chip RC oscillator
- In-circuit programming

In addition some microcontrollers in the family offer the following special features:

- Direct CAN 2.0B bus interface
- Direct USB 2.0 bus interface
- Direct LCD control interface
- TCP/IP interface
- ZigBee interface

11.3 PIC18F Microcontroller Architecture

The manufacturers' data sheets give detailed information about the architecture of their devices. In this section we shall be looking at the basic architecture of the PIC18F microcontroller series. PIC18F microcontroller series consists of a large number of microcontroller models. Perhaps the easiest way to learn their architecture is to look at a basic microcontroller in the family, namely the PIC18F452.

PIC18F452 microcontroller is a 40-pin device, housed in a DIL package, with a pin configuration as shown in Figure 11.1.

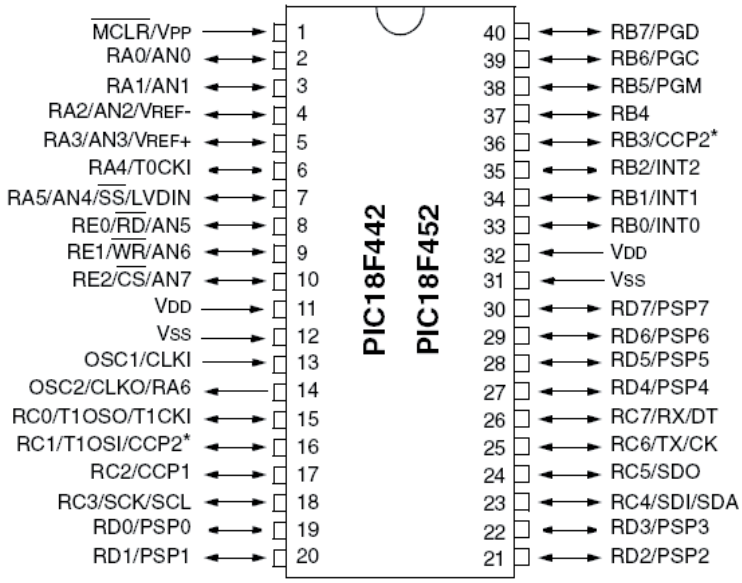


Figure 11.1 PIC18F452 pin configuration

Figure 11.2 shows the internal block diagram of the PIC18F452 microcontroller. The CPU is at the center of the figure and consists of an 8-bit ALU, an 8-bit working accumulator register (WREG), and an 8 x 8 hardware multiplier. The higher byte and lower byte of a multiplication are stored in two 8-bit registers called PRODH and PRODL respectively.

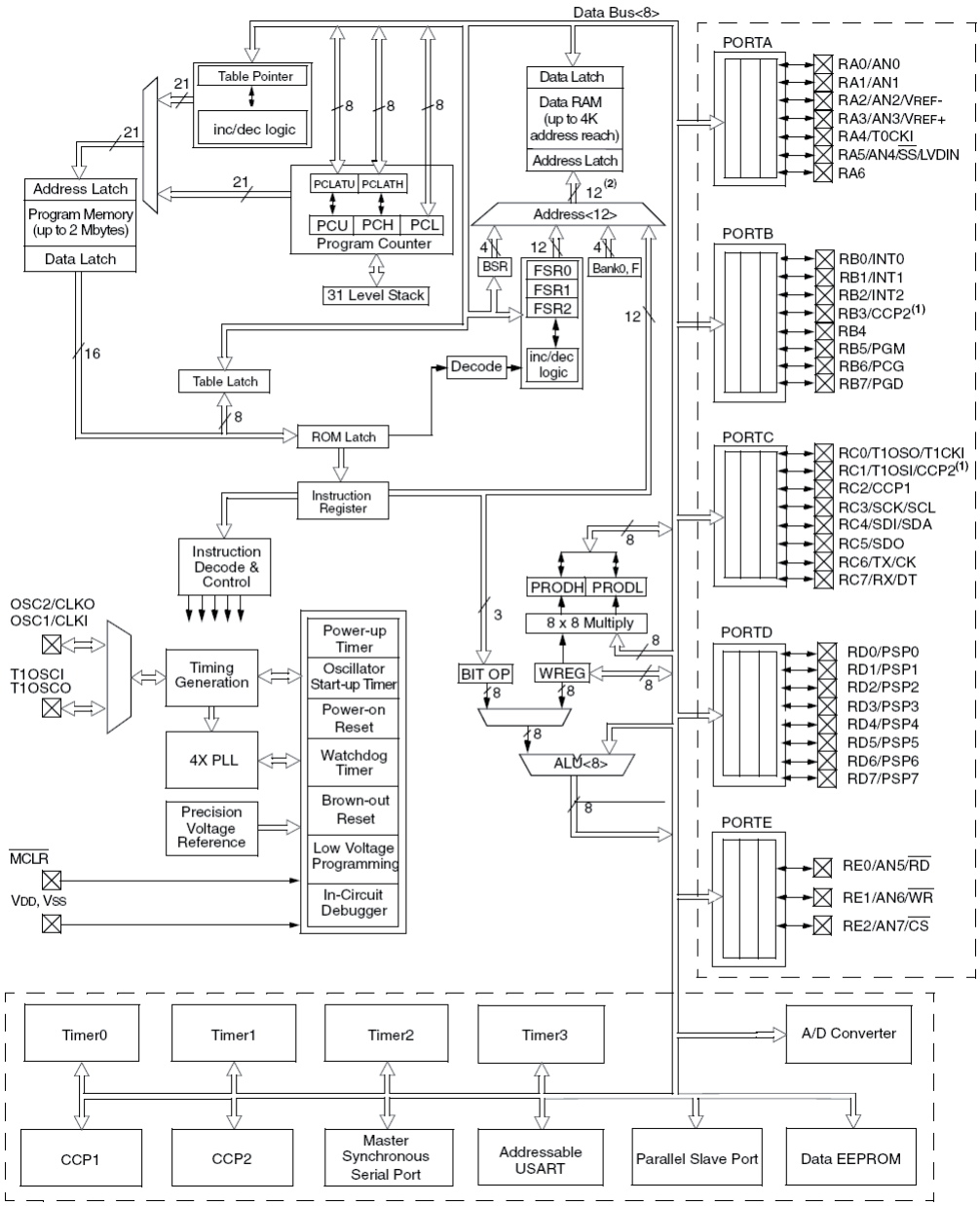


Figure 11.2 Block diagram of the PIC18F452 microcontroller

The program counter and the program memory are shown at the top left corner of the diagram. Program memory addresses consists of 21 bits, capable to access 2MByte of program memory locations. PIC18F452 has only 32Kbytes of program memory which requires only 15 bits, thus the remaining 6 address bits are redundant and not used. A table pointer provides access to tables and to the data stored in program memory. The program memory contains a 31 level stack which is normally used to store the interrupt and subroutine return addresses.

The data memory can be seen at the top central part of the diagram. The data memory bus is 12-bits wide, capable to access 4Kbytes of data memory locations. The data memory consists of the special function registers (SFR) and the general purpose registers, all organized in banks.

Bottom part of the diagram shows the timers/counters, capture/compare/PWM registers, USART, A/D converter and the EEPROM data memory. PIC18F452 consists of:

- 4 counters/timers
- 2 capture/compare/PWM modules
- 2 serial communication modules
- 8 10-bit A/D converter channels
- 256 bytes EEPROM

The oscillator circuit is located at the left hand side of the diagram. This circuit consists of :

- Power-up timer
- Oscillator start-up timer
- Power-on reset
- Watchdog timer
- Brown-out reset
- Low-voltage programming
- In-circuit debugger
- PLL circuit
- Timing generation circuit

The PLL provides the option of multiplying up the oscillator frequency to speed up the overall operation. The watchdog timer can be used to force a restart of the microcontroller in the event of a program crash. The in-circuit debugger is useful during program development and it can be used to return diagnostic data, including the register values as the microcontroller is executing a program.

The input-output ports are located at the right hand side of the diagram. PIC18F452 consists of 5 parallel ports named PORTA, PORTB, PORTC, PORTD and PORTE. Most port pins have multiple functions. For example, PORTA pins can be used as either parallel input-output, or as analog inputs. PORTB pins can be used as either parallel input-output, or they can be used as interrupt inputs.

11.4 Resetting the Microcontroller

Resetting a PIC® microcontroller starts execution from address 0 of the program memory. As described in this section, there are several methods to reset a PIC® microcontroller.

Power-on Reset

The power-on reset is generated automatically when power supply voltage is applied to the chip. The MCLR pin should be tied to the supply voltage directly or preferably through a 10K resistor. Figure 11.3 shows a typical reset circuit.

For applications where the rise time of the voltage is slow, it is recommended to use a diode, a capacitor, and a series resistor as shown in Figure 11.4.

External Reset

In some applications it may be required to reset the microcontroller externally by pressing a button. Figure 11.5 shows the circuit that can be used to reset the microcontroller externally. Normally the MCLR input is at logic 1. When the RESET button is pressed this pin goes to logic 0 and resets the microcontroller.

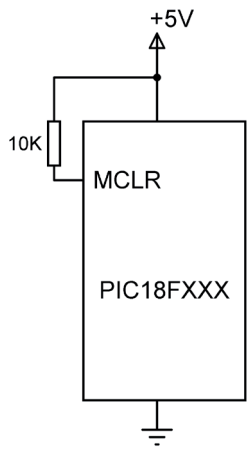


Figure 11.3 Typical reset circuit

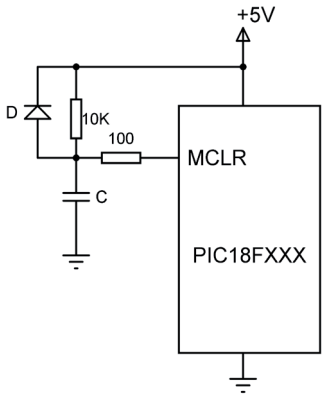


Figure 11.4 Reset circuit for slow rising voltages

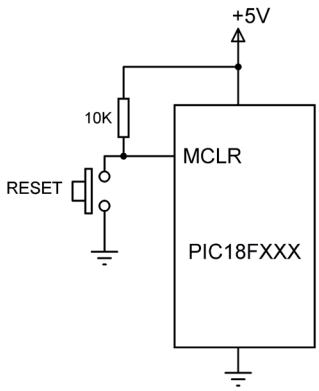


Figure 11.5 External reset circuit

11.5 Clock Sources

PIC18F452 microcontroller can be operated from external crystal or ceramic resonator connected to the OSC1 and OSC2 pins of the microcontroller. In addition, external resistor and capacitor, external clock source, and in some models internal oscillators can be used to provide clock pulses to the microcontroller. There are eight clock sources on the PIC18F452 microcontroller, selected by the configuration register CONFIG1H. These are:

- Low power crystal (LP)
- Crystal or ceramic resonator (XT)
- High speed crystal or ceramic resonator (HS)

- High speed crystal or ceramic resonator with PLL (HSPLL)
- External resistor/capacitor with FOSC/4 output on OSC2 (RC)
- External resistor/capacitor with I/O on OSC2 (port RA6) (RCIO)
- External clock with FOSC/4 on OSC2 (EC)
- External clock with I/O on OSC2 (port RA6) (ECIO)

Crystal or Ceramic Resonator Operation

The first modes use an external crystal or ceramic resonator, connected to the OSC1 and OSC2 pins. For applications where the timing accuracy is important crystal should be used. If a crystal is used, a parallel resonant crystal must be chosen since series resonant crystals do not oscillate when the system is first powered.

Figure 11.6 shows how a crystal is connected to the microcontroller. The capacitor values depend on the mode of the crystal and the selected frequency, and is usually in the range of 15-33 pF.

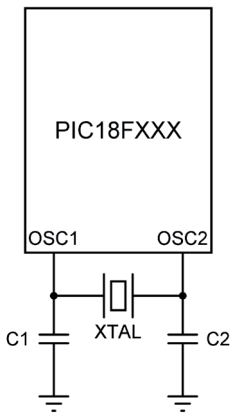


Figure 11.6 Using a crystal as the clock source

Resonators should be used in low cost applications where also high accuracy timing is not required. Figure 11.7 shows how a resonator is connected to the microcontroller.

LP (Low Power) oscillator mode should be selected in applications to up to 200kHz clock. XT mode should be selected to up to 4MHz, and the high speed HS mode should be selected in applications where the clock frequency is between 4MHz to 25MHz.

External Clock Operation

An external clock source may also be connected to the OSC1 pin in the LP, XT, and HS modes as shown in Figure 11.8.

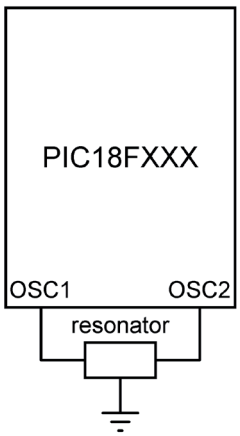


Figure 11.7 Using a resonator as the clock source

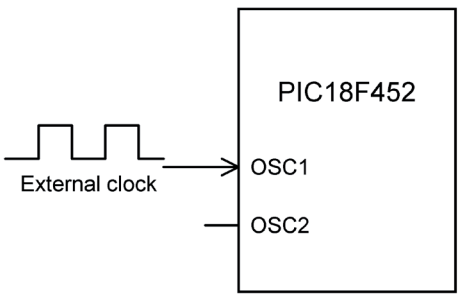


Figure 11.8 Connecting an external clock in LP, XT or HS modes

Resistor/Capacitor Operation

There are many applications where accurate timing is not required. In such applications we can use an external resistor and a capacitor to provide clock pulses (Figure 11.9 shows the RC values for a 2 MHz clock). The clock frequency is a function of the resistor, capacitor, power supply voltage, and the temperature. The clock frequency is not accurate and can vary from unit to unit due to manufacturing and component tolerances.

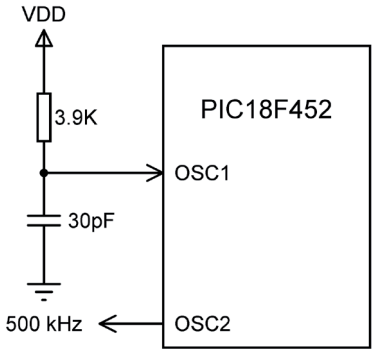


Figure 11.9 Generating clock in RC mode

Crystal or Resonator With PLL

One of the problems when high frequency crystals or resonators is electromagnetic interference. A Phase Locked Loop circuit is provided that can be enabled to multiply the clock frequency by 4. Thus, for a crystal clock frequency of 10MHz, the internal operation frequency will be multiplied to 40MHz. The PLL mode is enabled when the oscillator configuration bits are programmed for HS mode.

11.6 Parallel I/O Ports

Parallel I/O ports are used to interface the microcontroller to external world. The number of I/O ports and port pins vary depending on the PIC18F family member used, but all versions have at least PORT A and PORT B. The pins of a port are labelled as RPn, where P is the port letter and n is the port bit number. For example, PORT A pins are labelled RA0 to RA7, PORT B pins are labelled RB0 to RB7 and so on.

When working with a port we may want to:

- Set port direction
- Set an output value
- Read an input value
- Set an output value and then read back the output value

Parallel port directions are configured by programming registers called TRISn, where n is the port name, e.g. TRISA, TRISB, TRISC and so on. Each I/O port has a corresponding TRIS register. A bit programmed as "0" in the TRIS register configures the corresponding bit of the associated port pin as an output pin. Similarly, a bit programmed as "1" in the TRIS register configures the corresponding bit of the associated port pin as an input pin. For example, setting the TRISB register to bit pattern "11000001" configures bits 0, 7, and 8 of PORTB as input pins, and configures pins 1, 2, 3, 4, and 5 of PORT B as output pins (see Figure 11.10).

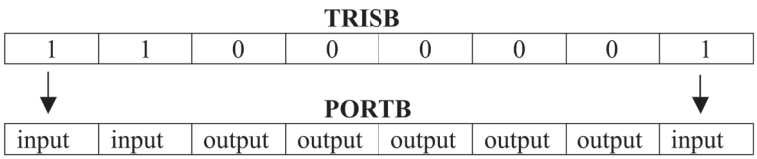


Figure 11.10 Using TRISB to configure PORTB pin directions

11.7 mikroC Programming Language

In this section we shall review the basic programming concepts of the C language. Further programming details and techniques can be obtained from the vast amount of books available in the market on C programming.
There are several C compilers in the market to develop programs for the PIC® microcontrollers. Most of the features of these compilers are similar and they can all be used to develop C based high-level programs for PIC® microcontrollers.

Some of the popular C compilers used in the development of commercial, industrial, and educational PIC18 microcontroller applications are:

- mikroC C compiler
- PICC18 C compiler
- C18 C compiler
- CCS C compiler

In this book we shall be using the highly popular mikroC compiler. mikroC compiler has been developed by *MikroElektronika* (web site: www.microe.com) and is one of the easy to learn compilers with rich resources, such as a large number of library functions and an integrated development environment with built-in simulator, and an in-circuit-debugger (e.g. mikroICD). A demo version of the compiler with a 2K program limit is available from mikroElektronika web site.

11.7.1 Structure of a mikroC Program

Figure 11.11 shows the simplest structure of a mikroC program. This program flashes an LED connected to port RB0 (bit 0 of PORT B) of a PIC® microcontroller with one second intervals. Do not worry if you do not understand the operation of the program at this stage as all will be clear as we progress through this chapter. Some of the programming elements used in Figure 11.11 are described below in detail.

Comments

Comments are used by programmers to clarify the operation of the program or a programming statement. Comment lines are ignored and not compiled by the compiler. Two types of comments can be used in mikroC programs: long comments extending several lines, and short comments occupying only a single line. Comment lines are usually used at the beginning of a program to describe briefly the operation of the program, the name of the author, the program filename, the date program was written, and a list of version numbers together with the modifications in each version. As shown in Figure 11.11, comments can also be used after statements to describe the operations performed by the statements. A well commented program is important for the maintenance and thus for the future lifetime of a program. In general, any programmer will find it easier to modify and/or update a well commented program.

```
/*****************************************************************

                        LED FLASHING PROGRAM
                        ********************

This program flashes an LED connected to port pin RB0 of PORT B
with one second intervals.

Programmer:             D. Ibrahim
File:                   LED.C
Date:                   January, 2011
Micro:                  PIC18F452

*****************************************************************/

void main()
{
  for(;;)               // Endless loop
  {
    TRISB = 0;          // Configure PORT B as output
    PORTB.0 = 0;        // RB0 = 0
    Delay_Ms(1000);     // Wait 1 second
    PORTB.0 = 1;        // RB0 = 1
    Delay_Ms(1000);     // Wait 1 second
  }                     // End of loop
}
```

Figure 11.11 Structure of a mikroC Program

As shown in Figure 11.11, long comments start with characters "/*" and terminate with characters "*/". Similarly, short comments start with characters "//" and there is no need to terminate short comments.

Beginning and Ending of a Program

In C language a program begins with the keywords:

```
void main()
```

After this, a curly opening bracket is used to indicate the beginning of the program body. The program is terminated with a closing curly bracket. Thus, as shown in Figure 11.11, the program has the following structure:

```
void main()
{
    program body
}
```

Terminating Program Statements

In C language all program statements must be terminated with the semicolon (";") character, otherwise a compiler error will be generated:

```
cnt = 25;
    // correct
cnt = 25
    // error
```

11.7.2 Variable Names

In C language variable names can begin with an alphabetical character or with the underscore character. In essence, variable names can be any of the characters a-z and A-Z, the digits 0-9 and the underscore character "_". Each variable name should be unique within the first 31 characters of its name. Variable names can contain upper case and lower case characters (see 3.1.5 above) and numeric characters can be used inside a variable name. Examples of valid variable names are:

```
Sum   count   sum100   counter   i1   UserName   _myName
```

Some names are reserved for the compiler itself and they can not be used as variable names in our programs.

11.7.3 Variable Types

mikroC language supports the variable types shown in Table 11.1. Examples of variables are given in this section.

Table 11.1 mikroC variable types

Type	Size (bits)	Range
unsigned char	8	0 to 255
unsigned short int	8	0 to 255
unsigned int	16	0 to 65535
unsigned long int	32	0 to 4294967295
signed char	8	-128 to 127
signed short int	8	-128 to 127
signed int	16	-32768 to 32767
signed long int	32	-2147483648 to 2147483647
float	32	±1.17549435082E-38 to ±6.80564774407E38
double	32	±1.17549435082E-38 to ±6.80564774407E38
long double	32	±1.17549435082E-38 to ±6.80564774407E38

(unsigned) char or unsigned short (int)

These are 8-bit unsigned variables with a range 0 to 255. In the following example two 8-bit variables named total and sum are created and sum is assigned decimal value 150:

```
    unsigned char total, sum;
    sum = 150;
or,
    char total, sum;
    sum = 150;
```

Variables can be assigned values during their declaration. Thus, the above statements can also be written as:

```
    char total, sum = 150;
```

signed char or (signed) short (int)

These are 8-bit signed character variables with a range -128 to +127. In the following example a signed 8-bit variable named counter is created with a value of -50:

```
    signed char counter = -50;
or,
    short counter = -50;
or,
    short int counter = -50;
```

(signed) int

These are 16-bit variables with a range -32768 to +32767. In the following example a signed integer named Big is created:

```
    int Big;
```

unsigned (int)

These variables are unsigned 16-bit with a range 0 to 65535. In the following example an unsigned 16-bit variable named count is created and is assigned value 12000:

```
    unsigned int count = 12000;
```

(signed) long (int)

These variables are 32-bits long with a range -2147483648 to + 2147483647. An example is given below:

```
    signed long LargeNumber;
```

unsigned long (int)

These are 32-bit unsigned variables having the range 0 to 4294967295. An example is given below:

```
    unsigned long VeryLargeNumber;
```

float or double or long double

These are floating point variables, implemented in mikroC using Microchip AN575 32-bit format which is IEEE 754 compliant. Floating point numbers range from ±1.17549435082E-38 to ±6.80564774407E38. In the following example a floating point variable named area is created and is assigned value 12.235:

```
float area;
area = 12.235;
```

In order to avoid confusion during program development it is recommended that you specify the sign of a variable type (signed or unsigned), followed by the type of the variable. For example, use **unsigned char** instead of **char** only. Similarly, use **unsigned int** instead of **unsigned** only.
In this book we shall be using the following mikroC data types which are easier to remember, and are also compatible with most other C compilers:

unsigned char	0 to 255
signed char	-128 to 127
unsigned int	0 to 65535
signed int	-32768 to 32767
unsigned long	0 to 4294967295
signed long	-2147483648 to 2147483647
float	±1.17549435082E-38 to ±6.80564774407E38

11.7.4 Constants

Constants represent fixed values (numeric or character) in programs that can not be changed. Constants are stored in the flash program memory of the PIC microcontroller, thus the valuable and limited RAM memory is not wasted. In mikroC constants can be: integer, floating point, character, string, or enumerated types.

Integer Constants

Integer constants can be decimal, hexadecimal, octal, or binary. The data type of a constant is derived by the compiler from its value. But, suffixes can be used to change the type of a constant.

From Table 11.1 we can see that decimal constants can have values from -2147483648 to +4294967295. For example, constant number 210 is stored as **unsigned char** (or **unsigned short int**). Similarly, constant number -200 is stored as **signed int**.

Using the suffix u or U forces the constant to be **unsigned**. Using the suffix L or l forces the constant to be **long**. Using both U (or u) and L (or l) forces the constant to be **unsigned long**.

Constants are declared using the keyword **const** and they are stored in the flash program memory of the PIC® microcontroller, thus not wasting any valuable RAM space. In the following example, constant **MAX** is declared as 100 and is stored in the flash program memory of the PIC® microcontroller:

```
const MAX = 100;
```

Hexadecimal constants start with characters 0x or 0X and may contain numeric data 0 to 9 and hexadecimal characters A to F. In the following example, constant **TOTAL** is given the hexadecimal value FF:

```
const TOTAL = 0xFF;
```

Octal constants have a zero at the beginning of the number and may contain numeric data 0 to 7. In the following example constant **CNT** is given octal value 17:

```
const CNT = 017;
```

Binary constant numbers start with 0b or 0B and may contain only 0 or 1. In the following example, a constant named **Min** is declared having the binary value "11110000":

```
const Min = 0b11110000
```

Floating Point Constants

Floating point constant numbers have integer parts, a dot, fractional part, and an optional e or E followed by a signed integer exponent. In the following example, a constant named **TEMP** is declared having the fractional value 37.50:

```
const TEMP = 37.50
```

or,

```
const TEMP = 3.750E1
```

Character Constants

A character constant is a character enclosed in a single quote. In the following example, a constant named **First_Alpha** is declared having the character value "A":

```
const First_Alpha = 'A';
```

String Constants

String constants are fixed sequences of characters stored in the flash memory of the microcontroller. The string must begin with a double quote character (") and also terminate with a double quote character. The compiler automatically inserts a null character as a terminator. An example string constant is:

```
"This is an example string constant"
```

A string constant can be extended across a line boundary by using a backslash character ("\"):

```
"This is first part of the string \
  and this is the continuation of the string"
```

The above string constant declaration is same as:

"This is first part of the string and this is the continuation of the string"

Enumarated Constants

Enumaration constants are integer type and they are used to make a program easier to follow. In the following example constant **colours** stores the names of colours. The first element is given the value 0:

```
enum colours {black, brown, red, orange, yellow, green, blue, grey, white};
```

Escape Sequences

Escapes sequences are used to represent non printable ASCII characters. For example, the character combination "\n" represents the newline character. An ASCII character can also be represented by specifying its hexadecimal code after a backslash. For example, the newline character can also be represented as '\x0A'.

11.7.5 Arrays

Arrays are used to store related items together in the same block of memory and under a specified name. An array is declared by specifying its type, name, and the number of elements it will store. For example,

```
unsigned int Total[5];
```

Creates an array of type unsigned int, with name Total, and having 5 elements. The first element of an array is indexed with 0. Thus, in the above example, Total[0] refers to the first element of this array and Total[4] refers to the last element. The array total is stored in memory in five consecutive locations as follows:

Total[0]
Total[1]
Total[2]
Total[3]
Total[4]

Data can be stored in the array by specifying the array name and index. For example, to store 25 in the second element of the array we have to write:

```
Total[1] = 25;
```

Similarly, the contents of an array can be read by specifying the array name and its index. For example, to copy the third array element to a variable called temp we have to write:

```
Temp = Total[2];
```

The contents of an array can be initialized during the declaration of the array by assigning a sequence of comma delimited values to the array. An example is given below where array months has 12 elements and months[0] = 31, months[1] = 28, and so on.:

unsigned char months[12] = {31,28,31,30,31,30,31,31,30,31,30,31};

The above array can also be declared without specifying the size of the array:

unsigned char months[ ] = {31,28,31,30,31,30,31,31,30,31,30,31};

Character arrays can be declared similarly. In the following example a character array named Hex_ Letters is declared with 6 elements:

unsigned char Hex_Letters[ ] = {'A', 'B', 'C', 'D', 'E', 'F'};

Strings are character arrays with a null terminator. Strings can either be declared by enclosing the string in double quotes, or each character of the array can be specified within single quotes, and then terminated with a null character. In the following example the two string declarations are identical and both occupy 5 locations in memory:

unsigned char Mystring[ ] = "COMP";
and
unsigned char Mystring[ ] = {'C', 'O', 'M', 'P', '\0'};

In C programming language we can also declare arrays with multiple dimensions. One dimensional arrays are usually called vectors, and two dimensional arrays are called matrices. A two dimensional array is declared by specifying the data type of the array, array name, and the size of each dimension. In the following example a two dimensional array named P is created having 3 rows and 4 columns. Altogether the array has 12 elements. The first element of the array is P[0][0], and the last element is P[2][3]. The structure of this array is shown below:

P[0][0]	P[0][1]	P[0][2]	P[0][3]
P[1][0]	P[1][1]	P[1][2]	P[1][3]
P[2][0]	P[2][1]	P[2][2]	P[2][3]

Elements of a multi-dimensional array can be specified during the declaration of the array. In the following example, two dimensional array Q has 2 rows and 2 columns and its diagonal elements are set to one, non-diagonal elements are cleared to zero:

unsigned char Q[2][2] = { {1,0}, {0,1} };

11.7.6 Pointers

Pointers are an important part of the C language and they hold the memory addresses of variables. Pointers are declared same as any other variables, but with the character ("*") in front of the variable name. In general, pointers can be created to point to (or hold the addresses of) character variables, integer variables, long variables, floating point variables, or they can point to functions (mikroC currently does not support pointers to functions).

In the following example, an unsigned character pointer named pnt is declared:

```
unsigned char *pnt;
```

When a new pointer is created its content is initially unspecified and it does not hold the address of any variable. We can assign the address of a variable to a pointer using the ("&") character:

```
pnt = &Count;
```

now pnt holds the address of variable Count. Variable Count can be set to a value by using the character ("*") in front of its pointer. For example, Count can be set to 10 using its pointer:

```
*pnt = 10;                  // Count = 10
```

which is same as

```
Count = 10;                 // Count = 10
```

Or, the value of Count can be copied to variable Cnt using its pointer:

```
Cnt = *pnt;                 // Cnt = Count
```

Array Pointers

In C language the name of an array is also a pointer to the array. Thus, for the array:

```
unsigned int Total[10];
```

Name Total is also a pointer to this array and it holds the address of the first element of the array. Thus, the following two statements are equal:

```
Total[2] = 0;   and   *(Total + 2) = 0;
```

Also, the following statement is true:

```
&Total[j] = Total + j
```

In C language we can perform pointer arithmetic which may involve:

- Comparing two pointers
- Adding or subtracting pointer and an integer value
- Subtracting two pointers
- Assigning one pointer to another one
- Comparing a pointer to null

For example, let us assume that pointer P is set to hold the address of array element Z[2]

```
P = &Z[2];
```

We can now clear elements 2 and 3 of array Z as in the following two examples. The two examples are identical except that in the first example pointer P holds the address of Z[3] at the end of the statements, and it holds the address of Z[2] at the end of the second set of statements :

```
   *P = 0;                 // Z[2] = 0
   P = P + 1;              // P now points to element 3 of Z
   *P = 0;                 // Z[3] = 0
or
   *P = 0;                 // Z[2] = 0
   *(P + 1) = 0;           // Z[3] = 0
```

A pointer can be assigned to another pointer. An example is given below where variables Cnt and Tot are both set to 10 using two different pointers:

```
   unsigned int *i, *j;    // declare 2 pointers
   unsigned int Cnt, Tot;  // declare two variables
   i = &Cnt;               // i points to Cnt
   *i = 10;                // Cnt = 10
   j = i;                  // copy pointer i to pointer j
   Tot = *j;               // Tot = 10
```

11.7.7 Structures

Structures can be used to collect related items as single objects. Unlike arrays, the members of structures can be a mixture of any data type. For example, a structure can be created to store the personal details (name, surname, age, date of birth etc.) of a student.

A structure is created by using the keyword **struct**, followed by a structure name, and a list of member declarations. Optionally, variables of the same type as the structure can be declared at the end of the structure.

The following example declares a structure named Person:

```
struct Person
{
        unsigned char name[20];
        unsigned char surname[20];
        unsigned char nationality[20];
        unsigned char age;
}
```

Declaring a structure does not occupy any space in memory, but the compiler creates a template describing the names and types of the data objects or member elements that will eventually be stored within such a structure variable. It is only when variables of the same type as the structure are created then these variables occupy space in memory. We can declare variables of the same type as the structure by giving the name of the structure and the name of the variable. For example, two variables **Me** and **You** of type Person can be created by the statement:

```
struct Person Me, You;
```

Variables of type Person can also be created during the declaration of the structure as shown below:

```
struct Person
{
        unsigned char name[20];
        unsigned char surname[20];
        unsigned char nationality[20];
        unsigned char age;
} Me, You;
```

We can assign values to members of a structure by specifying the name of the structure, followed by a dot (“.”), and the name of the member. In the following example, the **age** of structure variable **Me** is set to 25, and variable **M** is assigned to the value of **age** in structure variable **You**:

```
Me.age = 25;
M = You.age;
```

Structure members can be initialized during the declaration of the structure. In the following example, the radius and height of structure Cylinder are initialized to 1.2 and 2.5 respectively:

```
struct Cylinder
{
        float radius;
        float height;
} MyCylinder = {1.2, 2.5};
```

Values can also be set to members of a structure using pointers by defining the variable types as pointers. For example, if **TheCylinder** is defined as a pointer to structure Cylinder then we can write:

```
struct Cylinder
{
        float radius;
        float height;
} *TheCylinder;

TheCylinder -> radius = 1.2;
TheCylinder -> height = 2.5;
```

The size of a structure is the number of bytes contained within the structure. We can use the **sizeof** operator to get the size of a structure. Considering the above example,

```
sizeof(MyCylinder)
```

returns 8 since each float variable occupies 4 bytes in memory.

Bit fields can be defined using structures. With bit fields we can assign identifiers to bits of a variable. For example, to identify bits 0, 1, 2 and 3 of a variable as **LowNibble** and to identify the remaining 4 bits as **HighNibble** we can write:

```
struct
{
        LowNibble       : 4;
        HighNibble      : 4;
} MyVariable;
```

We can then access the nibbles of variable MyVariable as:

```
MyVariable.LowNibble = 12;
MyVariable.HighNibble = 8;
```

In C language we can use the **typedef** statements to create new types of variables. For example, a new structure data type named **Reg** can be created as follows:

```
typedef struct
{
        unsigned char name[20];
        unsigned char surname[20];
        unsigned age;
} Reg;
```

Variables of type **Reg** can then be created in exactly the same way as creating any other types of variables. In the following example, variables **MyReg**, **Reg1** and **Reg2** are created from data type **Reg**:

```
Reg MyReg, Reg1, Reg2;
```

The contents of one structure can be copied to another structure, provided that both structures have been derived from the same template. In the following example two structure variables P1 and P2 of same type have been created and P2 is copied to P1:

```
struct Person
{
        unsigned char name[20];
        unsigned char surname[20];
        unsigned int age;
        unsigned int height;
        unsigned weight;
}

struct Person P1, P2;
……
……
P2 = P1;
```

11.7.8 Operators in C

Operators are applied to variables and other objects in expressions and they cause some conditions or some computations to occur.

mikroC language supports the following operators:

- Arithmetic operators
- Logical operators
- Bitwise operators
- Conditional operators
- Assignment operators
- Relational operators
- Preprocessor operators

Arithmetic Operators

Arithmetic operators are used in arithmetic computations. Arithmetic operators associate from left to right and they return numerical results. A list of the mikroC arithmetic operators is given in Table 11.2.

Table 11.2 mikroC arithmetic operators

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder (integer division)
++	Auto increment
--	Auto decrement

Example use of arithmetic operators is given below:

```
/* Adding two integers */
5 + 12                          // equals 17

/* Subtracting two integers */
120 – 5                         // equals 115
10 – 15                         // equals -5

/* Adding two floating point numbers */
3.1 + 2.4                       // equals 5.5

/* Remainder (not for float) */
7 % 3                           // equals 1

/* Post-increment operator */
j = 4;
k = j++;                        // k = 4, j = 5

/* Pre-increment operator */
j = 4;
k = ++j;                        // k = 5, j = 5

/* Post-decrement operator */
j = 12;
k = j--;                        // k = 12, j = 11

/* Pre-decrement operator */
j = 12;
k = --j;                        // k = 11, j = 11
```

Relational Operators

Relational operators are used in comparisons. If the expression evaluates to TRUE, a 1 is returned, otherwise a 0 is returned.

All relational operators associate from left to right and a list of mikroC relational operators is given in Table 11.3.

Table 11.3 mikroC relational operators

Operator	Operation
= =	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Example use of relational operators is given below:

Assuming x = 10

```
x > 8          // returns 1
x = = 10       // returns 1
x < 100        // returns 1
x > 20         // returns 0
x != 10        // returns 0
x >= 10        // returns 1
x <= 10        // returns 1
```

Logical Operators

Logical operators are used in logical and arithmetic comparisons and they return TRUE (i.e. logical 1) if the expression evaluates to nonzero, or FALSE (i.e. logical 0) if the expression evaluates to zero. If more than one logical operator is used in a statement and if the first condition evaluates to false, the second expression is not evaluated.

A list of the mikroC logical operators is given in Table 11.4.

Table 11.4 mikroC logical operators

Operator	Operation
&&	AND
ll	OR
!	NOT

Example use of logical operators is given below:

Assuming x = 7

```
x > 0 && x < 10          // returns 1
x > 0 || x < 10          // returns 1
x >=0 && x <=10          // return 1
x >=0 && x < 5           // returns 0
```

Bitwise Operators

Bitwise operators are used to modify the bits of a variable. A list of the mikroC bitwise operators is given in Table 11.5.

Table 11.5 mikroC bitwise operators

Operator	Operation
&	Bitwise AND
l	Bitwise OR
^	Bitwise EXOR
~	Bitwise complement
<<	Shift left
>>	Shift right

Example use of bitwise operators is given below:

i. 0xFA & 0xEE returns 0xEA

0xFA: 1111 1010
0xEE: 1110 1110

0xEA: 1110 1010

ii. 0x01 | 0xFE returns 0xFF

0x08: 0000 0001
0xFE: 1111 1110

0xFE: 1111 1111

iii. 0x14 >> 1 returns 0x08 (shift 0x14 right by 1 digit)

0x14: 0001 0100
>>1 : 0000 1010

0x0A: 0000 1010

iv. 0x1A << 3 returns 0xD0 (shift left 0x1A by 3 digits)

0x1A: 0001 1010
<<3 : 1101 0000

0xD0: 1101 0000

Assignment Operators

In C language there are two types of assignments: simple assignments, and compound assignments. In simple assignments an expression is simply assigned to another expression, or an operation is performed using an expression and the result is assigned to another expression:

```
    Expression1 = Expression 2
or
    Result = Expression1 operation Expression2
```

Examples of simple assignments are:

```
    Temp = 10;
    Cnt = Cnt + Temp;
```

Compound assignments have the general format:

```
    Result  operation= Expression1
```

Here, the specified operation is performed on **Expression1** and the result is stored in **Result**.
For example:

```
    j += k;          is same as:          j = j + k;
also,
    p *=m;           is same as           p = p * m;
```

The following compound operators can be used in mikroC programs:

```
    +=      -=      *=      /=      %=
    &=      |=      ^=      >>=     <<=
```

Conditional Operator

The syntax of the conditional operator is:

```
    Result = Expression1 ? Expression2 : Expression3
```

Expression1 is evaluated first and if its value is true, **Expression2** is assigned to **Result**, otherwise **Expression3** is assigned to **Result**. In the following example maximum of **x** and **y** is found where **x** is compared with **y** and if **x > y** then **max = x**, otherwise **max = y**

```
max = (x > y) ? x : y;
```

In the following example lower case characters are converted to upper case. If the character is lower case (between 'a' and 'z') then by subtracting 32 from the character we obtain the equivalent upper case character:

```
c = (c >= 'a' && c <= 'z') ? (c – 32) : c;
```

Pre-Processor Operators

The pre-processor allows a programmer to:

- Compile a program conditionally such that parts of the code is not compiled
- Replace symbols with other symbols or values
- Insert text files into a program

The pre-processor operator is the ("#") character and any line of code with a leading ("#") is assumed to be a pre-processor command. Semicolon character (";") is not needed to terminate a pre-processor command.

mikroC compiler supports the following pre-processor commands:

```
#define
#undef
#if
#elif
#endif
#ifdef
#ifndef
#error
#line
```

#define, #undef, #ifdef, #ifndef

#define pre-processor command provides Macro expansion where every occurrence of an identifier in the program is replaced with the value of the identifier. For example, to replace every occurrence of MAX with value 100 we can write:

```
#define MAX 100
```

An identifier which has already been defined can not be defined again unless both definitions have the same values. One way to get round this problem is to remove the Macro definition:

```
#undef MAX
```

or, the existence of a Macro definition can be checked. In the following example, if **MAX** has not already been defined then it is given value 100, otherwise the #define line is skipped:

```
#ifndef   MAX
      #define MAX 100
#endif
```

Note that the **#define** pre-processor command does not occupy any space in memory.
We can pass parameters to a Macro definition by specifying the parameters in a parenthesis after the Macro name. For example, consider the Macro definition

```
#define ADD(a, b)      (a + b)
```

when this Macro is used in a program, (a,b) will be replaced with (a + b) as shown below:

```
p = ADD(x, y)
will be transformed into p = (x + y)
```

Similarly, we can define a Macro to calculate the square of two numbers:

```
#define SQUARE(a)   (a * a)
```

when we now use this Macro in a program:

```
p = SQUARE(x)        will be transformed into p = (x * x)
```

#include

The pre-processor directive #include is used to include a source file in our program. Usually header files with extension ".h" are used with #include. There are two formats of using the **#include**:

#include <file>
and
#include "file"

In first option the file is searched in the mikroC installation directory first and then in user search paths. In second option the specified file is searched in the mikroC project folder, then in the mikroC installation folder, and then in user search paths. It is also possible to specify a complete directory path as:

#include "C:\temp\last.h"

The file is then searched only in the specified directory path.

#if, #elif, #else, #endif

The above pre-processor commands are used for conditional compilation where parts of the source code can be compiled only if certain conditions are met. In the following example the code section where variables A and B are cleared to zero is compiled if M has a nonzero value, otherwise the code section where A and B are both set to 1 is compiled. Notice that the #if must be terminated with #endif:

11.7.9 Modifying the Flow of Control

Statements are normally executed sequentially from the beginning to the end of a program. We can use control statements to modify the normal sequential flow of control in a C program. The following control statements are available in mikroC programs:

- Selection statements
- Unconditional modification of flow
- Iteration statements

Selection Statements

There are two selection statements: ***If*** and ***switch***.

If Statement

The general format of the **if** statement is:

```
if(expression)
        Statement1;
else
        Statement2;
```

or,

```
if(expression)Statement1; else Statement2;
```

If the **expression** evaluates to TRUE, **Statement1** is executed, otherwise **Statement2** is executed. The **else** keyword is optional and may be omitted if not required. In the following example, if the value of **x** is greater than **MAX** then variable **P** is incremented by 1, otherwise it is decremented by 1:

```
if(x > MAX)
        P++;
else
        P--;
```

We can have more than one statement by enclosing the statements within curly brackets. For example,

```
if(x > MAX)
{
        P++;
        Cnt = P;
        Sum = Sum + Cnt;
}
else
        P--;
```

In the above example if **x** is greater than **MAX** then the three statements within the curly brackets are executed, otherwise the statement **P--** is executed.

Another example using the **if** statement is given below:

```
if(x > 0 && x < 10)
{
        Total += Sum;
        Sum++;
}
else
{
        Total = 0;
        Sum = 0;
}
```

switch Statement

The **switch** statement is used when there are a number of conditions and different operations are performed when a condition is true. The syntax of the **switch** statement is:

```
switch (condition)
{
        case condition1:
                Statements;
                break;
        case condition2:
                Statements;
                break;
        ……
        ……
        case conditionn:
                Statements;
                break;
        default:
                Statements;
}
```

The **switch** statement functions as follows: First the **condition** is evaluated. The **condition** is then compared to **condition1** and if a match is found statements in that case block are evaluated and control jumps outside the **switch** statement when the **break** keyword is encountered. If a match is not found, **condition** is compared to **condition2** and if a match is found statements in that case block are evaluated and control jumps outside the switch statements and so on. The **default** is optional and statements following **default** are evaluated if the **condition** does not match to any of the conditions specified after the **case** keywords.

In the following example, the value of variable Cnt is evaluated. If Cnt = 1, A is set to 1. If cnt = 10, B is set to 1, and if Cnt = 100, C is set to 1. If Cnt is not equal to 1, 10, or 100 then D is set to 1:

```
switch (Cnt)
{
        case 1:
                A = 1;
                break;

        case 10:
                B = 1;
                break;

        case 100:
                C = 1;
                break;

        default:
                D = 1;
}
```

Because white spaces are ignored in C language we could also write the above code as:

```
switch (Cnt)
{
        case 1:      A = 1; break;

        case 10:     B = 1; break;

        case 100:    C = 1; break;

        default:     D = 1;
}
```

Write a switch statement that will return the Y value given the X value.

11.7.10 Iteration Statements

Iteration statements enable us to perform loops in our programs where part of a code is repeated required number of times. In mikroC there are 4 ways that iteration can be performed and we will look at each one with examples:

- Using **for** statement
- Using **while** statement
- Using **do** statement
- Using **goto** statement

for Statement

The syntax of the **for** statement is:

```
for(initial expression; condition expression; increment expression)
{
        Statements;
}
```

The **initial expression** sets the starting variable of the loop and this variable is compared against the **condition expression** before an entry to the loop. Statements inside the loop are executed repeatedly, and after each iteration the value of **increment expression** is incremented. The iteration continues until the **condition expression** becomes false. An endless loop is formed if the **condition expression** is always true.

The following example shows how a loop can be set up to execute 10 times. In this example variable **i** starts from 0 and increments by 1 at the end of each iteration. The loop terminates when i =10 in which case the condition i < 10 becomes false. On exit from the loop the value of **i** is 10:

```
for(i = 0; i < 10; i ++)
{
        statements;
}
```

The above loop could also be formed by starting the **initial expression** with a nonzero value. Here, **i** starts with 1 and the loop terminates when I = 11. Thus, on exit from the loop the value of **i** is 11:

```
for(i = 1; i <= 10; i++)
{
        Statements;
}
```

The parameters of a **for** loop are all optional and can be omitted. If the **condition expression** is left out, it is assumed to be true. In the following example an endless loop is formed where the **condition expression** is always true and the value of **i** is starts with 0 and is incremented after each iteration:

```
/* Endless loop with incrementing i */
for(i=0; ; i++)
{
        Statements;
}
```

Another example of an endless loop is given below where all the parameters are omitted:

```
/* Example of endless loop */
for(; ;)
{
        Statements;
}
```

In the following endless loop **i** starts with 1 and is not incremented inside the loop:

```
/* Endless loop with i = 1 */
for(i=1; ;)
{
        Statements;
}
```

If there is only one statement inside the **for** loop we can omit the curly brackets as shown in the following example:

```
for(k = 0; k < 10; k++)Total = Total + Sum;
```

Nested for loops can be used in programs. In a nested for loop the inner loop is executed for each iteration of the outer loop. An example is given below where the inner loop is executed 5 times and the outer loop is executed 10 times. The total iteration count is 50:

```
/* Example of nested for loops */
for(i = 0; i < 10; i++)
{
        for(j = 0; j < 5; j++)
        {
                        Statements;
        }
}
```

In the following example the sum of all the elements of a 3x4 matrix M is calculated and stored in variable called Sum:

```
/* Add all elements of a 3x4 matrix */
Sum = 0;
for(i = 0; i < 3; i++)
{
        for(j = 0; j < 4; j++)
        {
                        Sum = Sum + M[i][j];
        }
}
```

Since there is only one statement to be executed, the above example could also be written as:

```
/* Add all elements of a 3x4 matrix */
Sum = 0;
for(i = 0; i < 3; i++)
{
        for(j = 0; j < 4; j++) Sum = Sum + M[i][j];
}
```

while Statement

This is another statement which can be used to create iteration in programs. The syntax of the **while** statement is:

```
while (condition)
{
        Statements;
}
```

Here, the statements are executed repeatedly until the **condition** becomes false, or, the statements are executed repeatedly as long as the **condition** is true. If the **condition** is false on entry to the loop then the loop will not be executed and the program will continue from the end of the **while** loop. It is important that the **condition** is changed inside the loop, otherwise an endless loop will be formed.

The following code shows how to set up a loop to execute 10 times using the **while** statement:

```
/* A loop that executes 10 times */
k = 0;
while (k < 10)
{
        Statements;
        k++;
}
```

At the beginning of the code variable **k** is 0. Since **k** is less than 10 the **while** loop starts. Inside the loop the value of **k** is incremented by 1 after each iteration. The loop repeats as long as k < 10 and is terminated when k = 10. At the end of the loop the value of **k** is 10.

Notice that an endless loop will be formed if **k** is not incremented inside the loop:

```
/* An endless loop */
   k = 0;
   while (k < 10)
   {
         Statements;
   }
```

An endless loop can also be formed by setting the **condition** to be always true:

```
   /* An endless loop */
   while (k = k)
   {
         Statements;
   }
```

Here is an example of calculating the sum of numbers from 1 to 10 and storing the result in variable called **sum**:

```
   /* Calculate the sum of numbers from 1 to 10 */
   unsigned int k, sum;
   k = 1;
   sum = 0;
   while(k <= 10)
   {
         sum = sum + k;
         k++;
   }
```

It is possible to have a **while** statement with no body. Such a statement is useful for example if we are waiting for an input port to change its value. An example is given below where the program will wait as long as bit 0 of PORT B (PORTB.0) is at logic 0. The program will continue when the port pin changes to logic 1:

```
while(PORTB.0 == 0);
// Wait until PORTB.0 to becomes 1
```

or,

```
while(PORTB.0);
```

It is possible to have nested **while** statements.

do Statement

The **do** statement is similar to the **while** statement but here the loop executes until the **condition** becomes false, or, the loop executes as long as the **condition** is true. The **condition** is tested at the end of the loop. The syntax of the **do** statement is:

```
do
{
        Statements;
} while (condition);
```

The first iteration is always performed whether the **condition** is true or false, and this is the main difference between the **while** statement and the **do** statement.

The following code shows how to setup a loop to execute 10 times using the **do** statement:

```
/* Execute 10 times */
k = 0;
do
{
        Statements;
        k++;
} while (k < 10);
```

The loop starts with k = 0 and the value of **k** is incremented inside the loop after each iteration. **k** is tested at the end of the loop and if **k** is not less than 10 the loop terminates. In this example because k = 0 at the beginning of the loop, the value of **k** is 10 at the end of the loop.

An endless loop will be formed if the condition is not modified inside the loop as shown in the following example. Here **k** is always less than 10:

```
/* An endless loop */
   k = 0;
   do
   {
         Statements;
   } while (k < 10);
```

An endless loop can also be created if the condition is set to be true all the time:

```
   /* An endless loop */
   do
   {
         Statements;
   } while (k = k);
```

It is possible to have nested **do** statements.

goto Statement

The **goto** statement can be used to alter the normal flow of control in a program. This statement causes the program to jump to a specified label. A label can be any alphanumeric character set starting with a letter and terminating with the colon (":") character.

Although not recommended, the **goto** statement can be used together with the **if** statement to create iterations in a program. The following example shows how to setup a loop to execute 10 times using the **goto** and **if** statements:

```
   /* Execute 10 times */
   k = 0;
Loop:
   Statements;
   k++;
   if(k < 10)goto Loop;
```

The loop starts with label **Loop** and variable k = 0 at the beginning of the loop. Inside the loop the statements are executed and **k** is incremented by 1. The value of **k** is then compared with 10 and the program jumps back to label **Loop** if k < 10. Thus, the loop is executed 10 times until the condition at the end becomes false. At the end of the loop the value of **k** is 10.

Continue and *break* Statements

The **continue** and **break** statements can be used inside iterations to modify the flow of control. The **continue** statement is usually used with **if** statement and causes the loop to skip an iteration. An example is given below which calculates the sum of numbers from 1 to 10 except number 5:

```
/* Calculate sum of numbers 1,2,3,4,6,7,8,9,10 */
Sum = 0;
i = 1;
for(i = 1; i <= 10; i++)
{
        if(i == 5) continue;          // Skip number 5
        Sum = Sum + i;
}
```

Similarly, the **break** statement can be used to terminate a loop from inside the loop. In the following example the sum of numbers from 1 to 5 are calculated even though the loop parameters are set to iterate 10 times:

```
/* Calculate sum of numbers 1,2,3,4,5 */
Sum = 0;
i = 1;
for(i = 1; i <= 10; i++)
{
        if(i > 5) break;
// Stop loop if i > 5
        Sum = Sum + i;
}
```

11.7.11 Functions and Libraries

A function is a self contained section of code written to perform a well defined action. Functions are usually created when it is required to perform an operation at several different parts of a main program. In addition, it is a good programming practice to divide a large program into a number of smaller independent functions. The statements within a function may be executed by calling (or invoking) the function.

An example function definition is shown below. This function, named **Mult**, receives two integer arguments **a** and **b** and returns their product. Notice that using brackets in a return statement are optional:

```
int Mult(int a, int b)
{
      return (a*b);
}
```

When a function is called, it generally expects to be given the number of arguments expressed in the function's argument list. For example, the above function can be called as:

```
z = Mult(x,y);
```

where variable **z** has the data type **int**. Notice that the arguments declared in the function header and the arguments passed when the function is called are independent of each other, even if they may have the same name. In the above example when the function is called variable **x** is copied to **a**, and variable **y** is copied to **b** on entry to function **Mult**.

Some functions do not return any data and the data type of such functions must be declared as **void**. An example is given below:

```
void LED(unsigned char D)
{
      PORTB = D;
}
```

void functions can be called without any assignment statements, but the brackets must be used to tell the compiler that a function call is made:

```
LED( );
```

Also, some functions do not have any arguments. In the following example the function, named **Compl**, complements PORT C of the microcontroller and it returns no data and has no arguments:

```
void Compl( )
{
      PORTC = ~PORTC;
}
```

The above function can be called as:

```
Compl( );
```

mikroC compiler provides a set of built-in functions which can be called from our programs. In addition, mikroC provides a large set of library functions. These library functions can be called from anywhere of a program and they do not require any header files to be included in the program. mikroC user manual gives detailed descriptions of each library function with examples.
The CAN bus projects in this book use LCD displays. The operation of an LCD display and the mikroC built-in LCD functions are described below to make you familiar with the LD interface and programming.

11.7.12 LCD Interface

In microcontroller systems the output of a measured variable is usually displayed using LEDs, 7-segment displays, or LCD type displays. LCDs have the advantages that they can be used to display alphanumeric or graphical data. Some LCDs have 40 or more character lengths with the capability to display several lines. Some other LCD displays can be used to display graphics images. Some modules offer colour displays while some others incorporate back lighting so that they can be viewed in dimly lit conditions.

There are basically two types of LCDs as far as the interface technique is concerned: parallel LCDs and serial LCDs. Parallel LCDs (e.g. Hitachi HD44780) are connected to a microcontroller using more than one data line and the data is transferred in parallel form. It is common to use either 4 or 8 data lines. Using a 4 wire connection saves I/O pins but it is slower since the data is transferred in two stages. Serial LCDs are connected to the microcontroller using only one data line and data is usually sent to the LCD using the standard RS-232 asynchronous data communication protocol. Serial LCDs are much easier to use but they cost more than the parallel ones.

The programming of a parallel LCD is usually a complex task and requires a good understanding of the internal operation of the LCD controllers, including the timing diagrams. Fortunately, mikroC language provides special library commands for displaying data on alphanumeric as well as on graphical LCDs. All the user has to do is connect the LCD to the microcontroller, define the LCD connection in the software, and then send special commands to display data on the LCD.

HD44780 LCD Module

HD44780 is one of the most popular alphanumeric LCD modules used in industry and also by hobbyists. This module is monochrome and comes in different sizes. Modules with 8, 16, 20, 24, 32, and 40 columns are available. Depending on the model chosen, the number of rows varies between 1,2 or 4. The display provides a 14-pin (or 16-pin) connector to a microcontroller. Table 11.6 gives the pin configuration and pin functions of a 14-pin LCD module. Below is a summary of the pin functions:

Table 11.6 Pin configuration of HD44780 LCD module

Pin no	Name	Function
1	V_{SS}	Ground
2	V_{DD}	+ ve supply
3	V_{EE}	Contrast
4	RS	Register select
5	R/W	Read/write
6	E	Enable
7	D0	Daat bit 0
8	D1	Data bit 1
9	D2	Data bit 2
10	D3	Data bit 3
11	D4	Data bit 4
12	D5	Data bit 5
13	D6	Data bit 6
14	D7	Data bit 7

V_{SS} is the 0V supply or ground. The V_{DD} pin should be connected to the positive supply. Although the manufacturers specify a 5V d.c. supply, the modules will usually work with as low as 3V or as high as 6V.

Pin 3 is named V_{EE} and this is the contrast control pin. This pin is used to adjust the contrast of the display and it should be connected to a variable voltage supply. A potentiometer is normally connected between the power supply lines with its wiper arm connected to this pin so that the contrast can be adjusted.

Pin 4 is the Register Select (RS) and when this pin is LOW, data transferred to the display is treated as commands. When RS is HIGH, character data can be transferred to and from the module.
Pin 5 is the Read/Write (R/W) line. This pin is pulled LOW in order to write commands or character data to the LCD module. When this pin is HIGH, character data or status information can be read from the module.

Pin 6 is the Enable (E) pin which is used to initiate the transfer of commands or data between the module and the microcontroller. When writing to the display, data is transferred only on the HIGH to LOW transition of this line. When reading from the display, data becomes available after the LOW to HIGH transition of the enable pin and this data remains valid as long as the enable pin is at logic HIGH.

Pins 7 to 14 are the eight data bus lines (D0 to D7). Data can be transferred between the microcontroller and the LCD module using either a single 8-bit byte, or as two 4-bit nibbles. In the latter case only the upper four data lines (D4 to D7) are used. 4-bit mode has the advantage that four less I/O lines are required to communicate with the LCD. In this book we shall be using alphanumeric based LCD only and look at the 4-bit interface only.

Connecting the LCD

mikroC compiler includes a built-in LCD library to help connect and control LCD displays. LCD connection to the microcontroller pins must be defined at the beginning of the program using the variables given in Table 11.7.

Table 11.7 LCD variables to be defined at the beginning of a program

The following variables must be defined in all projects using LCD Library	Description	Example
`extern sfr sbit LCD_RS`	Register Select line.	`sbit LCD_RS at RB4_bit`
`extern sfr sbit LCD_EN`	Enable line.	`sbit LCD_EN at RB5_bit`
`extern sfr sbit LCD_D7`	Data 7 line.	`sbit LCD_D7 at RB3_bit`
`extern sfr sbit LCD_D6`	Data 6 line.	`sbit LCD_D6 at RB2_bit`
`extern sfr sbit LCD_D5`	Data 5 line.	`sbit LCD_D5 at RB1_bit`
`extern sfr sbit LCD_D4`	Data 4 line.	`sbit LCD_D4 at RB0_bit`
`extern sfr sbit LCD_RS_Direction`	Register Select direction pin.	`sbit LCD_RS_Direction at TRISB4_bit`
`extern sfr sbit LCD_EN_Direction`	Enable direction pin.	`sbit LCD_EN_Direction at TRISB5_bit`
`extern sfr sbit LCD_D7_Direction`	Data 7 direction pin.	`sbit LCD_D7_Direction at TRISB3_bit`
`extern sfr sbit LCD_D6_Direction`	Data 6 direction pin.	`sbit LCD_D6_Direction at TRISB2_bit`
`extern sfr sbit LCD_D5_Direction`	Data 5 direction pin.	`sbit LCD_D5_Direction at TRISB1_bit`
`extern sfr sbit LCD_D4_Direction`	Data 4 direction pin.	`sbit LCD_D4_Direction at TRISB0_bit`

An example is given below.:

```
// Lcd pinout settings
sbit LCD_RS at RB4_bit;
sbit LCD_EN at RB5_bit;
sbit LCD_D7 at RB3_bit;
sbit LCD_D6 at RB2_bit;
sbit LCD_D5 at RB1_bit;
sbit LCD_D4 at RB0_bit;

// Pin direction
sbit LCD_RS_Direction at TRISB4_bit;
sbit LCD_EN_Direction at TRISB5_bit;
sbit LCD_D7_Direction at TRISB3_bit;
sbit LCD_D6_Direction at TRISB2_bit;
sbit LCD_D5_Direction at TRISB1_bit;
sbit LCD_D4_Direction at TRISB0_bit;
```

In the above example, the connection between the LCD and the microcontroller is as follows:

LCD pin	Microcontroller pin
RS	RB4
EN	RB5
D7	RB3
D6	RB2
D5	RB1
D4	RB0

Notice that only the 4 high data pins (D4, D5, D6, D7) of the LCD are connected to the microcontroller. This is known as 4-bit LCD interface. Also, the R/W pin of the LCD is not used and is normally connected to GND.

mikroC built-in LCD library supports the following functions:

Lcd_Init: Initializes the LCD module to configure the connection between the LCD and the microcontroller. The LCD to microcontroller interface must be defined before this function is called. This must be the first LCD function to be called.

Lcd_Out: Displays text on the LCD, starting at the specified row and column position.

For example the function Lcd_Out(1, 4, “TEXT”); displays the message TEXT at row 1, column 4 of the LCD.

Lcd_Out_Cp: Displays text at the current cursor position.

For example, the function Lcd_Out_Cp(“HELLO”);
displays text HELLO at the current cursor position.

Lcd_Chr: Displays a character on the LCD at the specified row and column position.

For example, the function Lcd_Chr(1, 3, “X”);
displays character X at row 1, column 3 of the LCD

Lcd_Chr_Cp: Displays a character at the current cursor position.

For example, the function Lcd_Chr_Cp(“X”);
displays character X at the current cursor position.

Lcd_Cmd: Sends a command to the LCD. Valid commands are shown in Table 11.8.

For example, the function Lcd_Cmd(_LCD_CLEAR);
clears the LCD display.

Table 11.8 LCD Commands

LCD Command	Purpose
_LCD_FIRST_ROW	Move cursor to the 1st row
_LCD_SECOND_ROW	Move cursor to the 2nd row
_LCD_THIRD_ROW	Move cursor to the 3rd row
_LCD_FOURTH_ROW	Move cursor to the 4th row
_LCD_CLEAR	Clear display
_LCD_RETURN_HOME	Return cursor to home position, returns a shifted display to its original position. Display data RAM is unaffected.
_LCD_CURSOR_OFF	Turn off cursor
_LCD_UNDERLINE_ON	Underline cursor on
_LCD_BLINK_CURSOR_ON	Blink cursor on
_LCD_MOVE_CURSOR_LEFT	Move cursor left without changing display data RAM
_LCD_MOVE_CURSOR_RIGHT	Move cursor right without changing display data RAM
_LCD_TURN_ON	Turn Lcd display on
_LCD_TURN_OFF	Turn Lcd display off
_LCD_SHIFT_LEFT	Shift display left without changing display data RAM
_LCD_SHIFT_RIGHT	Shift display right without changing display data RAM

11.7.13 Example Program

We should be able now to develop simple mikroC based programs. In this section we shall see how to compile and test a simple LED flashing program.

Example 11.1

An LED is connected to bit 0 of PORTB (pin RB0) of a PIC18FXXX (any model) microcontroller through a current limiting resistor. Choose a suitable value for the resistor and write a program that will flash the LED ON and OFF continuously at one-second intervals.

Solution 11.1

LEDs can be connected to a microcontroller in two modes: current sinking mode, and current sourcing mode. In current sinking mode (see Figure 11.12) one leg of the LED is connected to the +5V and the other leg is connected to the microcontroller output port pin through a current limiting resistor R.

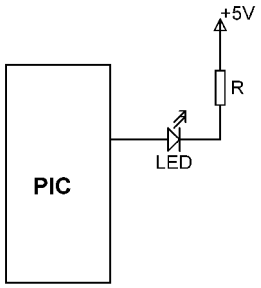


Figure 11.12 Connecting the LED in current sinking mode

Under normal working conditions the voltage across an LED is about 2V, and the current through the LED is about 10mA (some low power LEDs can operate at as low as 1mA current). The maximum current that can be sourced or sinked at the output port of a PIC microcontroller is 25mA.

The value of the current limiting resistor R can be calculated as follows. In current sinking mode the LED will be turned ON when the output port of the microcontroller is at logic 0. i.e. at approximately 0V. The required resistor is then:

$$R = \frac{5V - 2V}{10mA} = 0.3K$$

The nearest resistor to choose is 290 Ohm (a slightly higher resistor can be chosen for a lower current and slightly less brightness).

In current sourcing mode (see Figure 11.13) one leg of the LED is connected to the output port of the microcontroller and the other leg is connected to ground through a current limiting resistor. The LED will be turned ON when the output port of the microcontroller is at logic 1. i.e. at approximately 5V. Both in current sinking and current sourcing modes we can use the same value resistor.

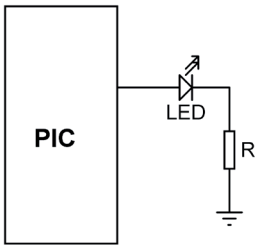


Figure 11.13 Connecting the LED in current sourcing mode

The required program listing is given in Figure 11.14 (program FLASH.C). The program should directly be written using the mikroC editor.

At the beginning of the program PORT B is configured as output using the TRISB = 0 statement. An endless loop is then formed with the **for** statement and inside this loop the LED is turned ON and OFF with 1 second delay between each output. The built-in function Delay_Ms(n) generates program delay of *n* milliseconds. Thus, Delay_Ms(1000) generates a one second delay in the program.

```
/*****************************************************************

                          FLASHING AN LED
                          ===============

This program flashes an LED connected to port RB0 of a microcontrol-
ler with one second intervals. mikroC built-in function Delay_ms is
used to create a 1 second delay between the flashes.

Programmer:             Dogan Ibrahim
File:                   FLASH.C
Date:                   January, 2011

*****************************************************************/
```

```
void main()
{
  TRISB = 0;              // Configure PORT B as output
  for(; ;)                // Endless loop
  {
      PORTB = 1;          // Turn ON LED
      Delay_ms(1000);     // 1 second delay
      PORTB = 0;          // Turn OFF LED
      Delay_Ms(1000);     // 1 second delay
  }
}
```

Figure 11.14 Program to flash an LED

After writing the program we should compile it by ticking the mikroC compile button. This is shown in Figure 11.15.

If the program is compiled with no errors, a message is displayed at the bottom of the screen to say that the compilation is successful (see Figure 11.16).

After a successful compilation, a listing file, a HEX file, and some other files are generated. The listing file shows the assembly listing of the program with line numbers. The important file is the HEX file as this is the file that is loaded into program memory of the target microcontroller. The HEX file of this example is shown in Figure 11.17.

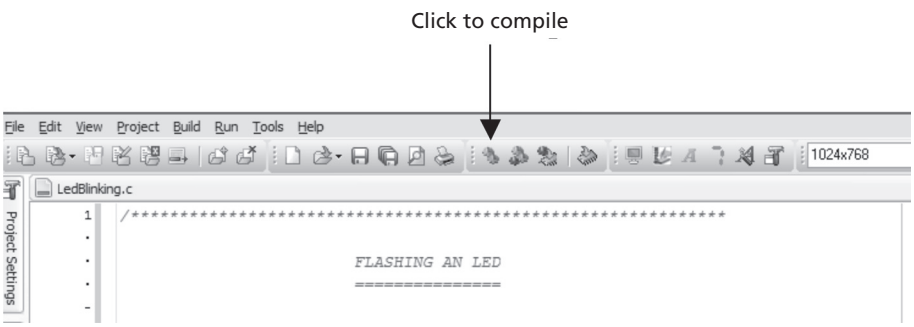


Figure 11.15 Compiling the program

Line	Message No.	Message Text
13	123	Compiled Successfully
0	127	All files Compiled in 328 ms
0	1144	Used RAM (bytes): 1 (1%) Free RAM (bytes): 1514 (99%)
0	1144	Used ROM (bytes): 84 (1%) Free ROM (bytes): 32684 (99%)
0	125	Project Linked Successfully
0	128	Linked in 875 ms
0	129	Project 'test.mcppi' completed: 2312 ms
0	103	Finished successfully: 26 Dec 2010, 20:59:06

Figure 11.16 Successful compilation

```
:1000000016EF00F00000000000EF00F0000000001C
:0A0010000000000000000000F3D71C
:10001C000900F5CFE6FF0006FBE10106F9E112004D
:10002C00156A946A15C082FF152A020E0B6E040E17
:10003C000C6EBA0E0D6E0D2EFED70C2EFCD70B2EA1
:08004C00FAD70000EFD7FFD73F
:020000040030CA
:0D000100220F0EFF0181FF0FC00FE00F4026
:00000001FF
```

Figure 11.17 HEX file of the example

11.7.14 Testing

The hardware must be constructed before the program can be tested. Here, the user has the following options:

- Use a microcontroller development board
- Construct the hardware on a PCB
- Construct the hardware on a breadboard

The simplest and the most effective way of testing a microcontroller system is to use a development board. There are many development boards available in the market place. The one used by the author is the popular *EasyPIC6*, developed by mikroElektronika (source: http://www.mikroe.com).

Figure 11.18 shows a picture of the EasyPIC6 development board. One of the nice things about this board is that it is fully compatible with the mikroC compiler and a compiled program can easily be loaded into program memory of the target microcontroller (which is placed in a socket on the development board) by selecting **Tools -> meProgrammer** from the mikroC drop-down menu (see Figure 11.19).

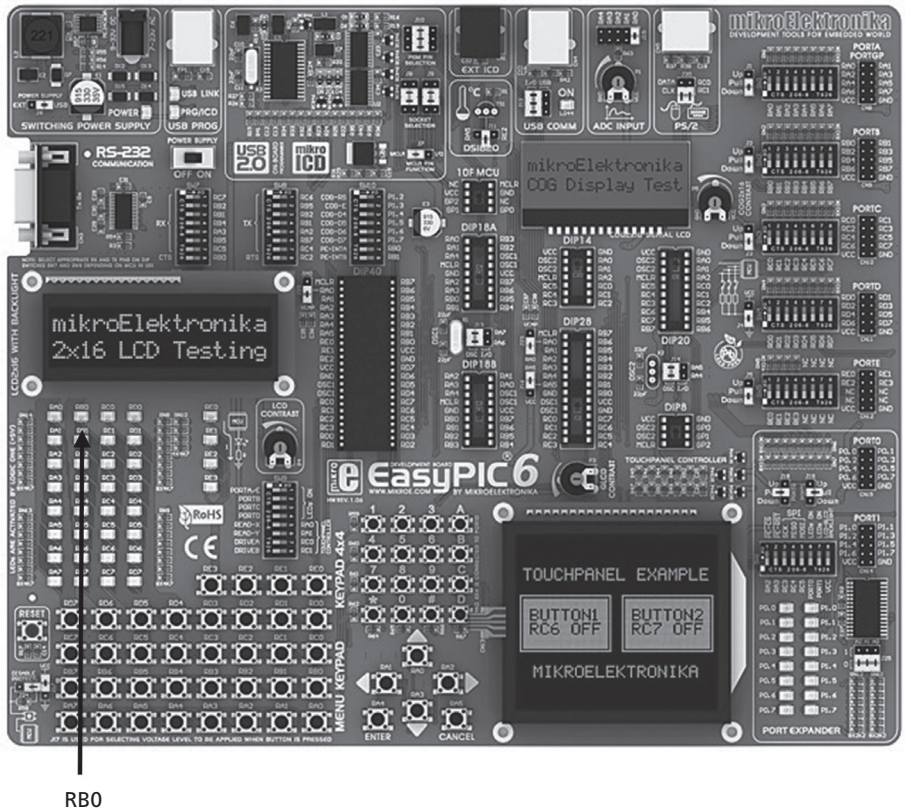


Figure 11.18 EasyPIC6 development board

The EasyPIC6 development board has the following specifications:

- Supports over 160 8, 14, 18, 20, 28 and 40-pin PIC microcontrollers.
- On-board USB programmer
- In-circuit debugger (mikroICD)
- RS232 port.
- 2x16 character LCD and 128x64 graphics LCD.
- Touch screen controller.
- 36 LEDs.
- 36 push-button switches.
- PS2 and USB connector.
- 4x4 keypad.
- Port expander logic.
- USB or external power supply for the board.

After loading the program into program memory of the target microcontroller, press the Reset button on the development board to start the program running. You should see the LED connected to port pin RB0 (see Figure 11.18) flashing at a rate of 1 second.

11.7.15 PIC® Microcontroller CAN Interface

In general, any type of PIC microcontroller can be used in CAN bus-based projects. A microcontroller with a built-in CAN bus module will almost simplify the design and shorten the development and testing times. Figure 11.20 shows the block diagram of a PIC microcontroller based CAN bus application, using a PIC 12 or PIC 16 type microcontroller with no built-in CAN module. As you can see, an external CAN module (MCP2515) and an external CAN transceiver (MCP2551) are used in this design.

For new CAN bus projects it is easier to use a PIC® microcontroller with built-in CAN module. As shown in Figure 11.21, such devices include built-in CAN controller hardware on the chip. All that is required to make a CAN node is to add a CAN transceiver chip.

There are many types of PIC microcontrollers with built-in CAN modules. Some examples are: PIC18F258, PIC 18F2580, PIC18F2680, PIC18F8585, PIC18F8680, and so on. In this book, the popular PIC18F258 is used in CAN based projects.

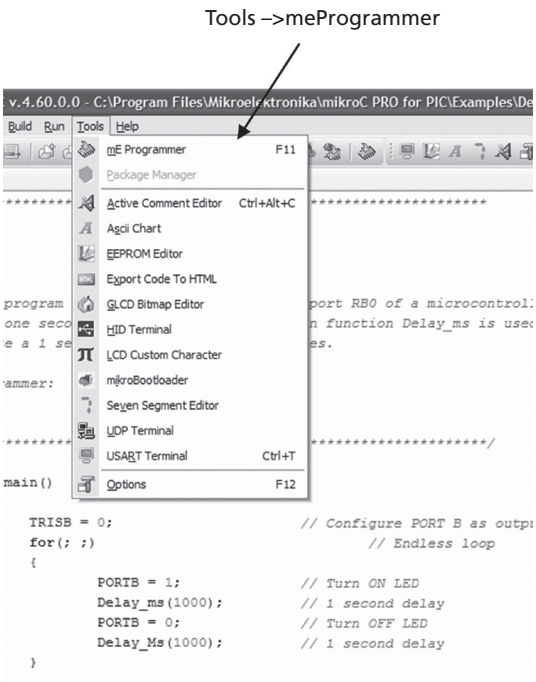


Figure 11.19 Programming the target microcontroller

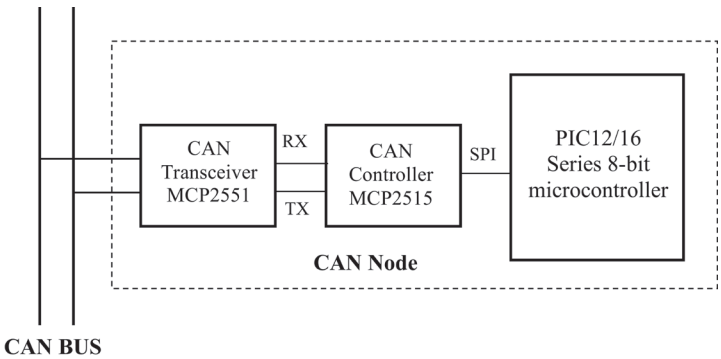


Figure 11.20 CAN node with no built-in CAN bus module

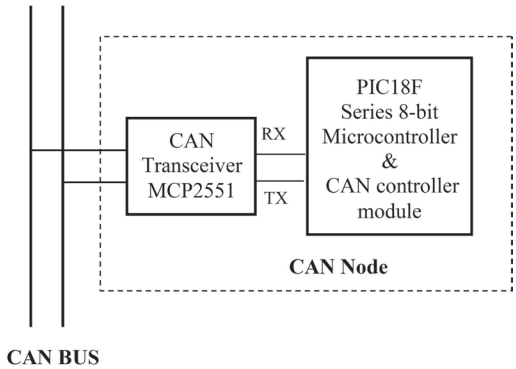


Figure 11.21 CAN node with built-in CAN bus module

11.7.16 PIC18F258 Microcontroller

All the CAN based projects in this book are based on the PIC18F258 microcontroller. PIC18F258 is a high performance microcontroller with integrated CAN module. The device has the following basic features (see the PIC18F258 data sheet, Microchip Inc., source: *http://www.microchip.com*):

- 32K flash program memory
- 1536 bytes RAM data memory
- 256 bytes EEPROM data memory
- 22 I/O ports#10-bit A/D converter (5 channels)
- SPI/I²C bus module
- CAN bus module
- 8 x 8 hardware multiplier

The PIC18F258 microcontroller's CAN module has the following basic features:

- Compatible with CAN 2.0A and CAN 2.0B
- Bit rate up to 1 Mbps
- Three transmit buffers
- Two receive buffers
- Six acceptance filters
- Two acceptance filter masks

The CAN module uses port B, pins RB3/CANRX and RB2/CANTX for CAN receive and transmit functions respectively. It is important to know how the CAN module operates so that it can be used efficiently. A brief operation of this module is given below.

Operation of the CAN Module

A node uses acceptance filters to decide whether or not to accept a received message. Message filtering is applied to the whole identifier field, and mask registers are used to specify which bits in the identifier the filters should examine. For example, setting the mask bits to all 1s causes all the bits in the message identifiers to be examined.

The CAN module in PIC18F258 has six modes of operation:

Configuration Mode: The CAN module is initialized in this mode. Transmission or reception of frames are not allowed in this mode. The error counters are cleared in this mode.

Disable Mode: In this mode the internal clock is stopped and the module can not transmit or receive frames.

Normal Operation Mode: This is the normal operation mode where both transmission and reception of data frames are allowed.

Listen-Only Mode: This mode is usually used to monitor the bus status. The module can receive messages, including errors, but can not transmit frames.

Loop-Back Mode: This mode is used for testing where messages can be directed from internal transmit buffers to receive buffers without actually being transmitted.

Error Recognition Mode: This mode is used to ignore all errors and receive all messages. In this mode, all messages, valid or invalid are received and copied to the receive buffer.

11.7.17 PIC18F258 Message Transmission

The PIC18F258 includes three transmit buffers, named TXB0, TXB1, and TXB2. Messages to be transmitted are in a priority queue. The transmit buffer with the highest priority is sent out first. If two buffers have the same priority, the one with the higher buffer number is sent first.

11.7.18 PIC18F258 Message Reception

The PIC18F258 microcontroller includes two receive buffers, RXB0 and RXB1, with multiple acceptance filters for each (see Figure 11.22). All received messages are initially assembled in the message assembly buffer (MAB). It is important to realize that once a message is received, regardless of its identifier, the entire message is copied into the MAB.

Received messages have priorities, where RXB0 is a higher priority buffer. RXB0 has two acceptance filters: RXF0 and RXF1. Similarly, RXB1 has four acceptance filters: RXF2, RXF3, RXF4, and RXF5. Two programmable acceptance masks are also available, RXM0 and RXM1 (see Figure 11.22) one for each receive buffer.

The CAN module uses message acceptance filters and masks to determine if a received message in the MAB should be accepted and loaded into a receive buffer. Basically, the message identifier of the received message is compared to the acceptance filter values. If there is a match, that message is loaded into the appropriate receive buffer. The acceptance masks determine which bits in the identifier are to be examined with the acceptance filters. Table 11.9 shows how each bit in the identifier is compared to the masks and filters. If a mask bit is set to 0, that bit in the identifier is automatically accepted regardless of the filter bit. If on the other hand, a mask bit is set to 1, that bit in the identifier is used in the comparison.

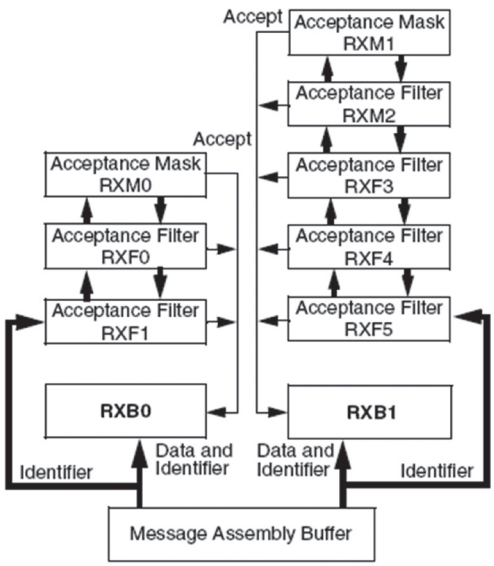


Figure 11.22 PIC18F258 receive buffers, filters, and masks

In summary, in reference to Table 11.9, if a mask bit is 0 (row 1), the corresponding identifier bit will always be accepted. If on the other hand a mask bit is 1 (rows 2, 3, 4, 5), the identifier bit will only be accepted it is equal to the corresponding filter bit.

Table 11.9 Acceptance filter/mask truth table

Mask bit n	Filter bit n	Identifier bit n001	Status of bit n
0	x	x	Accept
1	0	0	Accept
1	0	1	Reject
1	1	0	Reject
1	1	1	Accept

11.7.19 mikroC built-in CAN functions

In this section we shall be looking at the various built-in CAN functions that can be used in our mikroC programs during the development of CAN based programs. Two libraries are provided: one for microcontrollers with no built-in CAN modules, and the other one for microcontrollers with built-in CAN modules. Our projects in this Chapter are based on the PIC18F258 microcontroller which has a built-in CAN module.

The library is based on using the SPI bus for PIC® microcontrollers, and it is important to learn how to use these functions if we want to develop CAN based microcontroller projects. The following functions are provided:

- CANSetOperationMode
- CANGetOperationMode
- CANInitialize
- CANSetBaudRate
- CANSetMask
- CANSetFilter
- CANRead
- CANWrite

We shall now look briefly at each function to see how to use these functions in our programs.

i. CANSetOperationMode

This function sets CAN o requested mode. The function prototype is:

*void CANSetOperationMode(**unsigned short** mode, **unsigned short** wait_flag)*

Parameter *wait_flag* is either 0 or 0xFF. If set to 0xFF, the function will not return until the requested mode is set. If set to 0, the function will return immediately as a non-blocking call.

Parameter *mode* can take one of the following values:

- _CAN_MODE_NORMAL normal mode of operation
- _CAN_MODE_SLEEP Sleep mode of operation
- _CAN_MODE_LOOP Loop-Back mode of operation
- _CAN_MODE_LISTEN Listen-Only mode of operation
- _CAN_MODE_CONFIG Configuration mode

ii. CAN_GetOperationMode

This function returns the current operation mode. The function prototype is:

unsigned short *CANGetOperationMode()*

iii. CAN_Initialize

This function initializes the CAN module. All pending transmissions are aborted, all msk registers are cleared to allow all messages. Upon execution of this function, the normal mode is set. The function prototype is:
*void CANInitialize(**char** SJW, **char** BRP, **char** PHSEG1, **char** PHSEG2, **char** PROPEG, **char** CAN_CONFIG_FLAGS)*

where all the parameters except the last one are the timing parameters (see Chapter 8 on how to calculate these parameters),

SJW	is the synchronization jump width
BRP	is the baud rate prescaler
PHSEG1	is the Phase_Seg1 timing value
PHSEG2	is the Phase_Seg2 timing value
PROPSEG	s the Prop_Seg value

CAN_CONFIG_FLAGS can take one of the following values (these values can be AND'ed to form complex configuration values):

- *_CAN_CONFIG_DEFAULT* Default flags
- *_CAN_CONFIG_PHSEG2_PRG_ON* Use supplied PHSEG2 value
- *_CAN_CONFIG_PHSEG2_PRG_OFF* Use max of PHSEG1 or information processing time (IPT) whichever is greater.
- *_CAN_CONFIG_LINE_FILTER_ON* Use CAN bus line filter for wake-up
- *_CAN_CONFIG_FILTER_OFF* Do not use CAN bus line filter
- *_CAN_CONFIG_SAMPLE_ONCE* Sample bus once at sample point
- *_CAN_CONFIG_SAMPLE_THRICE* Sample bus three times prior to sample point
- *_CAN_CONFIG_STD_MSG* Accept only standard identifier messages
- *_CAN_CONFIG_XTD_MSG* Accept only extended identifier messages
- *_CAN_CONFIG_DBL_BUFFER_ON* Use double buffering to receive data
- *_CAN_CONFIG_DBL_BUFFER_OFF* Do not use double buffering
- *_CAN_CONFIG_ALL_MSG* Accept all messages including invalid ones
- *_CAN_CONFIG_VALID_XTD_MSG* Accept only valid extended identifier messages
- *_CAN_CONFIG_VALID_STD_MSG* Accept only valid standard identifier messages
- *_CAN_CONFIG_ALL_VALID_MSG* Accept all valid messages

iv. CANSetBaudRate

This function is used to set the CAN node baud rate. The function prototype is:

*void CANSetBaudRate(**char** SJW, **char** BRP, **char** PHSEG1, **char** PHSEG2, **char** PROPSEG**, char** CAN_CONFIG_FLAGS)*

The arguments are as in function *CANInitialize*.

v. CANSetMask

This function sets the mask for filtering messages. The function prototype is:

*void CANSetMask(**char** CAN_MASK, **long** value, **char** CAN_CONFIG_FLAGS)*

CAN_MASK can have one of the following values:

- *_CAN_MASK_B1* Receive buffer 1 mask value
- *_CAN_MASK_B2* receive buffer 2 mask value

value is the mask register value

CAN_CONFIG_FLAGS can be one of the following:

- *_CAN_CONFIG_XTD* Extended message
- *_CAN_CONFIG_STD* Standard message

vi. CANSetFilter

This function sets filter values. The function prototype is:

*void CANSetFilter(**char** CAN_FILTER, **long** value, **char** CAN_CONFIG_FLAGS)*

CAN_FILTER can be one of the following:

- *_CAN_FILTER_B1_F1* Filter 1 for buffer 1
- *_CAN_FILTER_B1_F2* Filter 2 for buffer 1
- *_CAN_FILTER_B2_F1* Filter 1 for buffer 2
- *_CAN_FILTER_B2_F2* Filter 2 for buffer 2
- *_CAN_FILTER_B2_F3* Filter 3 for buffer 2
- *_CAN_FILTER_B2_F4* Filter 4 for buffer 2

value is the filter register value.

CAN_CONFIG_FLAGS can be one of the following:

- *_CAN_CONFIG_XTD* Extended message
- *_CAN_CONFIG_STD* Standard message

vii. CANRead

This function reads message from a receive buffer (or returns zero if no message is found). The function prototype is:

*char CANRead(**long** *id, **char** *data, **char** *datalen, **char** *CAN_RX_MSG_FLAGS)*

id is the message identifier
data is an array of bytes up to 8 where the received data bytes are stored.

datalen is the length of the received data in bytes (1 to 8).

CAN_RX_MSG_FLAGS can take one of the following values (the flags can be AND'ed if desired):

- *_CAN_RX_FILTER_1* Receive buffer filter 1 accepted this message
- *_CAN_RX_FILTER_2* Receive buffer filter 2 accepted this message
- *_CAN_RX_FILTER_3* Receive buffer filter 3 accepted this message
- *_CAN_RX_FILTER_4* Receive buffer filter 4 accepted this message
- *_CAN_RX_FILTER_5* Receive buffer filter 5 accepted this message
- *_CAN_RX_FILTER_6* Receive buffer filter 6 accepted this message
- *_CAN_RX_OVERFLOW* Receive overflow occurred
- *_CAN_RX_INVALID_MSG* Invalid message received
- *_CAN_RX_XTD_FRAME* Extended identifier message received
- *_CAN_RX_RTR_FRAME* RTR frame message received
- *_CAN_RX_DBL_BUFFERED* This message was double buffered

vii. CANWrite

This function is used to send a message to the CAN bus. The function prototype is:
*Unsigned short CANWrite(**long** *id, **char** *data, **char** *datalen, **char** *CAN_TX_MSG_FLAGS)*

id is the message identifier.
data is an array of bytes up to 8 where the data to be sent are stored.

datalen is the length of the data in bytes (1 to 8).

CAN_TX_MSG_FLAGS can take one of the following values (the flags can be AND'ed if desired):

- *_CAN_TX_PRIORITY_0* Transmit priority 0
- *_CAN_TX_PRIORITY_1* Transmit priority 1
- *_CAN_TX_PRIORITY_2* Transmit priority 2
- *_CAN_TX_PRIORITY_3* Transmit priority 3
- *_CAN_TX_STD_FRAME* Standard identifier message
- *_CAN_TX_XTD_FRAME* Extended identifier message
- *_CAN_TX_NO_RTR_FRAME* Non RTR message
- *_CAN_TX_RTR_FRAME* RTR message

11.7.20 CAN Bus Project Development

Developing a CAN bus based project using a microcontroller requires two things: a hardware, and a software. The hardware usually consists of a microcontroller development board with a suitable microcontroller, and the software is a high-level language compiler. In this Chapter the EasyPIC6 microcontroller development board (source: *http://www.mikroe.com*) with a PIC18F258 microcontroller are used as the hardware. The software of the projects is based on the mikroC compiler as described earlier in this Chapter.

The software development consists of the following steps:

- Configure the CAN bus I/O port directions (RB2 and RB3)
- Initialize the CAN module (*CANInitialize*)
- Set the CAN module to CONFIG mode (*CANSetOperationMode*)
- Set the mask registers (*CANSetMask*)
- Set the filter registers (*CANSetFilter*)
- Set the CAN module to normal mode (*CANSetOperationMode*)
- Read/write data as required *(CANRead/CANWrite*)

11.7.21 CAN BUS PROJECT 1

A very simple CAN bus project is given in this section. The block diagram of the project is shown in Figure 11.23. The project is made up of two CAN nodes. One node (node: **ACTIVATE**) scans a push-button switch every second and sends the switch status to the other node. An LED is connected to the other node and this node (node: **LED**) reads the switch status and turns ON the LED when the switch is pressed.

The circuit diagram of the project is shown in Figure 11.24. Two PIC18F258 type microcontrollers are used, one for each node. The microcontrollers are connected to the CAN bus using MCP2551 type bus transceivers. The TXD and RXD pins of the transceiver are connected to pins CANTX and CANRX of the microcontrollers respectively.

Port pin RC0 of node ACTIVATE is connected to a push-button switch, where the normal status of the switch is logic 1. When the switch is pressed, a logic 0 is sent to pin RC0 of the microcontroller. Port pin RC0 of node LED is connected to an LED through a current limiting resistor. When the system is powered-up the LED is OFF. The LED turns ON when the push-button switch on node ACTIVATE is pressed and remains ON until the switch is released. This process is repeated forever.

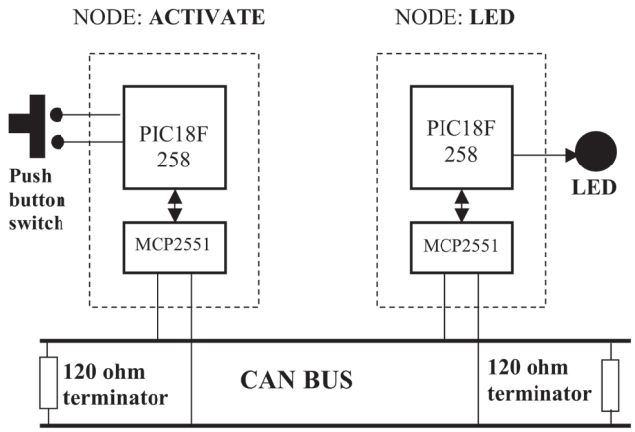


Figure 11.23 Block diagram of the project

Node: ACTIVATE

The ACTIVATE node consists of a PIC18F258 microcontroller with built-in CAN module, and an MCP2551 CAN bus transceiver chip. Pins CANH and CANL of the transceiver chip are connected to the CAN bus, which is terminated with 120 ohm resistors at both ends. The microcontroller is operated from an 8 MHz crystal. The MCLR pin is connected to an external reset button. The push-button switch is normally at logic 1 and is connected to port pin RC0 of this node.

Node LED

Like the ACTIVATE node, this node consists of a PIC18F258 microcontroller with built-in CAN module, and an MCP2551 transceiver chip. The microcontroller is operated from an 8 MHz crystal. An LED is connected to port pin RC0 of the microcontroller.

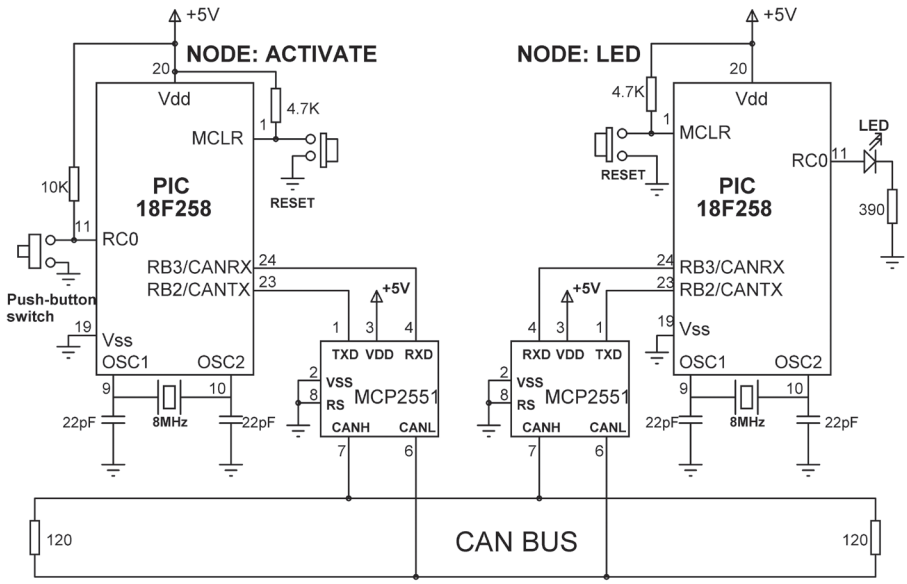


Figure 11.24 Circuit diagram of the project

The operation of the system is described by the Program Description Language (PDL) given in Figure 11.25. Node ACTIVATE is given the *message identifier number 3*, and node LED is given the *message identifier number 5*.

At the beginning, both nodes are initialized and the timing parameters are loaded into the microcontroller CAN modules. Then, the nodes are placed into Configuration Mode, and the acceptance masks and acceptance filters are set as required. Then the nodes are put into normal operation mode. Both nodes have endless loops. Node ACTIVATE scans the push-button switch every second and sends the switch status to node LED over the CAN bus. Node LED on the other hand receives the switch status and turns ON the LED when the switch is pressed.

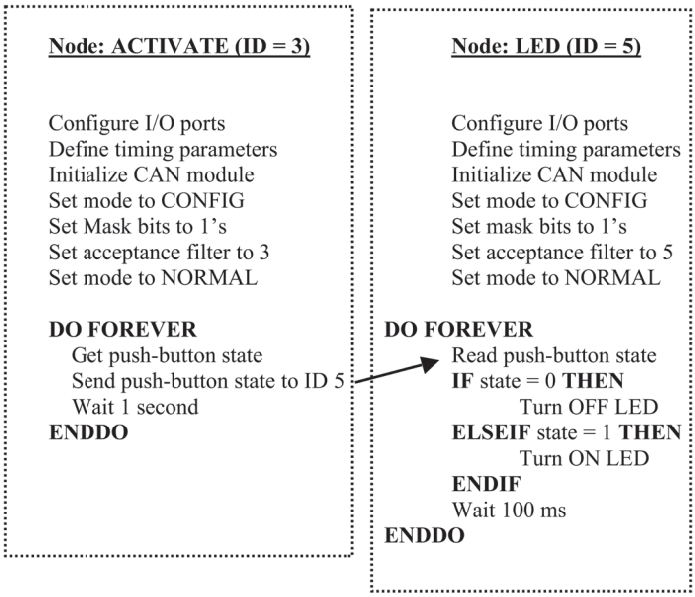


Figure 11.25 Operation of the system

In this project, the bus length is assumed to be about 20 m and the CAN bus bit rate is selected as 125 kbps. As calculated in Example 8.2 in Chapter 8, the timing parameters to be used are as follows:

Specification:

Bit rate =	125 kbps
Bus length =	20 m
Bus propagation delay =	5 ns/m
Controller TX delay =	80 ns
Controller RX delay =	120 ns
Oscillator frequency =	8 MHz

Timing parameters:

Sync_Seg =	1
Prop_Seg =	1
Phase_Seg1 =	3
Phase_Seg2 =	3
SJW =	3
Oscillator tolerance =	1.485%
Oscillator prescaler (BRP) =	8
Oscillator frequency =	1 MHz
Time quantum (T_Q) =	1 μs
Time quanta per bit (TQPB) =	8

We can now develop the programs for both nodes.

Node ACTIVATE Program

Figure 11.26 shows the program listing of node ACTIVATE, called ACTIVATE.C. At the beginning of the program PORT C is bit RC0 is configured as input, RB3 (CANRX) is configured as input, and RB2 (CANTX) is configured as output. Then, *CANInitialize* function is called to initialize the CAN bus module with the following timing parameters:

Sync_Seg =	1
Prop_Seg =	1
Phase_Seg1 =	3
Phase_Seg2 =	3
SJW =	3
BRP =	8

The configuration flag is made up from the bitwise AND of:

```
config_flag = _CAN_CONFIG_SAMPLE_THRICE      &
              _CAN_CONFIG_PHSEG2_PRG_ON      &
              _CAN_CONFIG_STD_MSG            &
              _CAN_CONFIG_DBL_BUFFER_ON      &
              _CAN_CONFIG_VALID_STD_MSG      &
              _CAN_CONFIG_LINE_FILTER_OFF;
```

where, the bus is to be sampled three times, user supplied PHSEG2 is to be used, standard message identifier is specified, double buffering is turned ON, and the line filter is turned OFF. The CAN module is set to operate in standard protocol mode (STD).

The CAN module is then put into configuration mode and the acceptance mask and acceptance filter are specified. In this example, it was decided to use receive buffer 2 (RXB1). Thus, acceptance mask 1 and mask 2 are set to all 1's (-1 is a shorthand way of writing hexadecimal FFFFFFFF, i.e. setting all mask bits to 1's) so that all bits will be examined when comparing the message identifier of a new message with the acceptance filter.

Filter 3 for buffer 2 (CAN_FILTER_B2_F3) is set to value 3 so that message identifiers having values 3 will be accepted by this node, and such messages will be loaded into buffer 2. Note that any one receive buffer, and any filters can be used when there is only one filter to be programmed.

The CAN module is then set to normal operation mode. The rest of the program is an endless loop where the push-button switch is sampled every second and its state is sent to node LED. CAN function *CANWrite* is used to send data over the CAN bus.

```
/*****************************************************************

                    CAN BUS EXAMPLE - NODE: ACTIVATE
                    ================================

This project consists of two CAN nodes. One of the nodes is called
ACIVATE and the other node is called LED. A push-button switch is
connected to node ACTIVATE. Node LED has an LED. Normally the LED
is OFF. When the push-button switch is pressed in node ACTIVATE,
the LED in node LED turns ON.

This is the program of node ACTIVATE of the CAN bus example. In
this project a PIC18F258 type microcontroller is used. An MCP2551
type CAN bus transceiver is used to connect the microcontroller to
the CAN bus. The microcontroller is operated from an 8MHz crystal
with an external reset button.

Pin CANRX and CANTX of the microcontroller are connected to pins
RXD and TXD of the transceiver chip respectively. Pins CANH and
CANL of the transceiver chip are connected to the CAN bus.
```

```
A push-button switch is connected to port pin RC0.Normally pin RC0
is at logic 1. When the switch is pressed the pin goes to logic 0.
The switch is sampld every second, so it must be left pressed for
at least one second so that İt to be recognised every time it is
pressed.

CAN timing parameters are:

  Microcontroller clock: 8MHz
  CAN Bus bit rate:      125 Kb/s
  Sync_Seg:              1
  Prop_Seg:              1
  Phase_Seg1:            3
  Phase_Seg2:            3
  SJW:                   3
  BRP:                   8

Author:                  Dogan Ibrahim
Date:                    January 2011
File:                    ACTIVATE.C

*******************************************************************/

void main()
{
     unsigned char push_button, sdata[8];
     unsigned short config_flag, send_flag;
     char SJW, BRP, Phase_Seg1, Phase_Seg2, Prop_Seg;
     long LED_ID, ACTIVATE_ID, mask;
//
// Message identifiers of nodes
//
      LED_ID = 5;         // Message id of node LED
      ACTIVATE_ID = 3;    // Message id of node ACTIVATE
//
// Configure port directions
//
     TRISC = 1;          // RC0 is input
     TRISB = 0x08;       // RB2 is output, RB3 is input
//
// CAN BUS timing parameters
```

```
//
     SJW = 3;
     BRP = 8;
     Phase_Seg1 = 3;
     Phase_Seg2 = 3;
     Prop_Seg = 1;
//
// Configuration
//
     config_flag = _CAN_CONFIG_SAMPLE_THRICE     &
                   _CAN_CONFIG_PHSEG2_PRG_ON     &
                   _CAN_CONFIG_STD_MSG           &
                   _CAN_CONFIG_DBL_BUFFER_ON     &
                   _CAN_CONFIG_VALID_STD_MSG     &
                   _CAN_CONFIG_LINE_FILTER_OFF;

     send_flag = _CAN_TX_PRIORITY_0             &
                 _CAN_TX_STD_FRAME              &
                 _CAN_TX_NO_RTR_FRAME;

//
// Initialize CAN module
//
      CANInitialize(SJW, BRP, Phase_Seg1, Phase_Seg2, Prop_Seg,
  config_flag);
//
// Set CAN CONFIG mode
//
      CANSetOperationMode(_CAN_MODE_CONFIG, 0xFF);

      mask = -1;
//
// Set all MASK1 bits to 1's
//
      CANSetMask(_CAN_MASK_B1, mask, _CAN_CONFIG_STD_MSG);

//
// Set all MASK2 bits to 1's
//
      CANSetMask(_CAN_MASK_B2, mask, _CAN_CONFIG_STD_MSG);
//
```

```
// Set id of filter B2_F3 to 3 (ACTIVATE_ID)
//
      CANSetFilter(_CAN_FILTER_B2_F3, ACTIVATE_ID,
                              _CAN_CONFIG_STD_MSG);
//
// Set CAN module to NORMAL mode
//
      CANSetOperationMode(_CAN_MODE_NORMAL, 0xFF);

//
// Endless program loop. SCAN the push-button switch every second
// and send its value to node LED. If the push-button switch is
pressed
// a "1" will be sent to node LED, otherwise a "0" will be sent.
Node LED
// will turn ON its LED if it receives a "1".
//
    for(;;)                                    // Endless loop
    {
      push_button = PORTC.F0;                  // Get button state

      if(push_button != 0)
          sdata[0] = '0';                      // If not pressed
      else
          sdata[0] = '1';                      // If pressed

      //
      // Send button state ("0" or "1") to node LED
      //
      CANWrite(LED_ID, sdata, 1, send_flag);   // Send button state

      Delay_Ms(1000);                          // Wait 1 second
  }                                            // end of for loop
}                                              // end of program
```

Figure 11.26 Program listing of node ACTIVATE

Node LED Program

Figure 11.27 shows the program listing of node LED, called LED.C. At the beginning of the program PORT C, bit RC0 is configured as output, RB3 (CANRX) is configured as input, and RB2 (CANTX) is configured as output. Then, *CANInitialize* function is called to initialize the CAN bus module with the following timing parameters:

```
Sync_Seg =          1
Prop_Seg =          1
Phase_Seg1 =        3
Phase_Seg2 =        3
SJW =               3
BRP =               8
```

The configuration flag is made up from the bitwise AND of:

```
config_flag = _CAN_CONFIG_SAMPLE_THRICE                    &
              _CAN_CONFIG_PHSEG2_PRG_ON                    &
              _CAN_CONFIG_STD_MSG                          &
              _CAN_CONFIG_DBL_BUFFER_ON                    &
              _CAN_CONFIG_VALID_STD_MSG                    &
              _CAN_CONFIG_LINE_FILTER_OFF;
```

where, the bus is to be sampled three times, user supplied PHSEG2 is to be used, standard message identifier is specified, double buffering is turned ON, and the line filter is turned OFF. The CAN module is set to operate in standard protocol mode (STD)

The CAN module is then put into configuration mode and the acceptance mask and acceptance filter are specified. In this example, it was decided to use receive buffer 2 (RXB1). Thus, acceptance mask 1 and mask 2 are set to all 1's (-1 is a shorthand way of writing hexadecimal FFFFFFFF, i.e. setting all mask bits to 1's) so that all bits will be examined when comparing the message identifier of a new message with the acceptance filter.

Filter 3 for buffer 2 (CAN_FILTER_B2_F3) is set to value 5 so that message identifiers having values 5 will be accepted by this node, and such messages will be loaded into buffer 2. Note that any one receive buffer, and any filters can be used when there is only one filter to be programmed.

The CAN module is then set to normal operation mode. The LED is intially turned OFF to start with. The rest of the program is an endless loop where messages are received on the CAN bus. Messages

consist of one character. If the character is "0" then the LED is turned OFF, otherwise the LED is turned ON. This loop is repeated every 100 milliseconds.

TESTING

All the projects in this Chapter were built and tested using the CAN Bus Development Kit (see Chapter 9), consisting of two EasyPIC6 microcontroller development boards, two CAN transceivers, and a twisted cable. The PIC18F258 microcontroller in each board is programmed from the mikroC compiler. Figure 11.28 shows the development and testing environment where everything is connected, and is ready for testing.

Testing the project was easy. After the two programs were compiled successfully they were loaded into the program memories of the two microcontrollers using the in-circuit programming feature of the EasyPIC6 board. Power was then applied to the boards (e.g. by connecting the boards to the USB port of a computer). Normally the LED connected to port pin RC0 (see Figure 11.28) of node LED should be OFF. Pressing the push-button switch connected to port pin RC0 of node ACTIVATE should turn the LED ON.

```
/*****************************************************************

                      CAN BUS EXAMPLE - NODE: LED
                      ===========================

This project consists of two CAN nodes. One of the nodes is called
ACIVATE and the other node is called LED. A push-button switch is
connected to node ACTIVATE. Node LED has an LED. Normally the
LED is OFF. When the push-button switch is pressed in node
ACTIVATE, the LED in node LED turns ON.

This is the program of node LED of the CAN bus example. In this
project a PIC18F258 type microcontroller is used. An MCP2551 type
CAN bus transceiver is used to connect the microcontroller to the
CAN bus. The microcontroller is operated from an 8MHz crystal with
an external reset button.

Pin CANRX and CANTX of the microcontroller are connected to pins RXD
and TXD of the transceiver chip respectively. Pins CANH and CANL of
the transceiver chip are connected to the CAN bus.
```

```
An LED is connected to port pin RC0 of the microcontroller. The LED
is normally OFF and is turned ON when the push-button switch is
pressed in node ACTIVATE.

CAN timing parameters are:

  Microcontroller clock: 8MHz
  CAN Bus bit rate:      125 Kb/s
  Sync_Seg:              1
  Prop_Seg:              1
  Phase_Seg1:            3
  Phase_Seg2:            3
  SJW:                   3
  BRP:                   8

Author:                  Dogan Ibrahim
Date:                    January 2011
File:                    LED.C

*****************************************************************/

void main()
{
     unsigned char push_button, read_flag, rdata[8];
     unsigned short config_flag, len;
     char SJW, BRP, Phase_Seg1, Phase_Seg2, Prop_Seg;
     long LED_ID, ACTIVATE_ID, id, mask;
//
// Message identifiers of nodes
//
      LED_ID = 5;         // Message id of node LED
      ACTIVATE_ID = 3;    // Message id of node ACTIVATE
//
// Configure port directions
//
     TRISC = 0;           // RC0 is output (LED port)
     TRISB = 0x08;        // RB2 is output, RB3 is input
//
// CAN BUS timing parameters
//
```

```
     SJW = 3;
     BRP = 8;
     Phase_Seg1 = 3;
     Phase_Seg2 = 3;
     Prop_Seg = 1;
//
// Configuration
//
     config_flag =_CAN_CONFIG_SAMPLE_THRICE     &
                  _CAN_CONFIG_PHSEG2_PRG_ON     &
                  _CAN_CONFIG_STD_MSG           &
                  _CAN_CONFIG_DBL_BUFFER_ON     &
                  _CAN_CONFIG_VALID_STD_MSG     &
                  _CAN_CONFIG_LINE_FILTER_OFF;

      read_flag = 0;
//
// Initialize CAN module
//
      CANInitialize(SJW, BRP, Phase_Seg1, Phase_Seg2, Prop_Seg,
                    config_flag);
//
// Set CAN CONFIG mode
//
      CANSetOperationMode(_CAN_MODE_CONFIG, 0xFF);

      mask = -1;
//
// Set all MASK1 bits to 1's
//
      CANSetMask(_CAN_MASK_B1, mask, _CAN_CONFIG_STD_MSG);

//
// Set all MASK2 bits to 1's
//
      CANSetMask(_CAN_MASK_B2, mask, _CAN_CONFIG_STD_MSG);
//
// Set id of filter B2_F3 to 5 (LED_ID)
//
CANSetFilter(_CAN_FILTER_B2_F3, LED_ID,
             _CAN_CONFIG_STD_MSG);
```

```
//
// Set CAN module to NORMAL mode
//
      CANSetOperationMode(_CAN_MODE_NORMAL, 0xFF);

//
// Endless program loop. Read the single character sent by node
// ACTIVETE. If this character is 0, turn OFF the LED, otherwise
turn ON
// the LED. Turn OFF the LED to start with
//
     PORTC.F0 = 0;                                    // Turn OFF LED

     for(;;)                                          // Endless loop
     {
         //
         // Read data on CAN bus
         //
         read_flag = CANRead(&id, rdata, &len, &read_flag);
         //
         // Check the message ID just in case.
         // read_flag != 0 if message is available
         //

         if(read_flag != 0 && id == LED_ID)
         {
                  if(rdata[0] == '0')
                           PORTC.F0 = 0;          // Turn OFF LED
                  else if(rdata[0] == '1')
                           PORTC.F0 = 1;          // Turn ON LED
         }

         // end of if

         Delay_Ms(100);                           // Wait 100 ms
      }                                           // end of for loop
}                                                 // end of program
```

Figure 11.27 Program listing of node LED

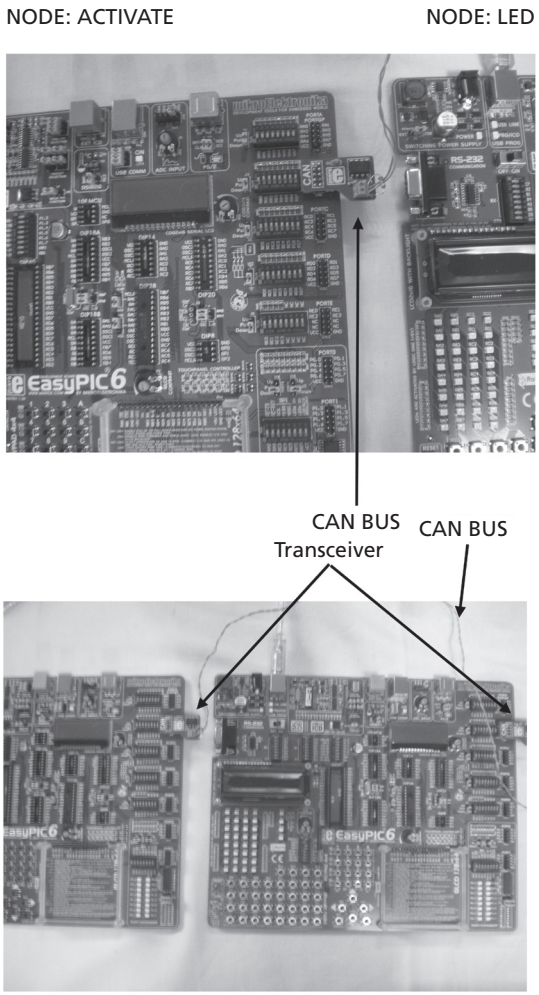


Figure 11.28 The development and testing environment

11.7.22 CAN BUS PROJECT 2

This project is slightly more complex than Project 1. Here, again two nodes are used. Node 1 is called **SENSE**, and node 2 is called **DISPLAY**. A temperature sensor is connected to an analog port of node SENSE. Node DISPLAY is connected to a text based LCD. The operation of the project is such that node SENSE senses the temperatures and sends it every second to node DISPLAY where it is displayed on the LCD.

Figure 11.29 shows block diagram of the project. Again, two PIC18F258 type microcontrollers are used in the project with MCP2551 type CAN bus transceivers.

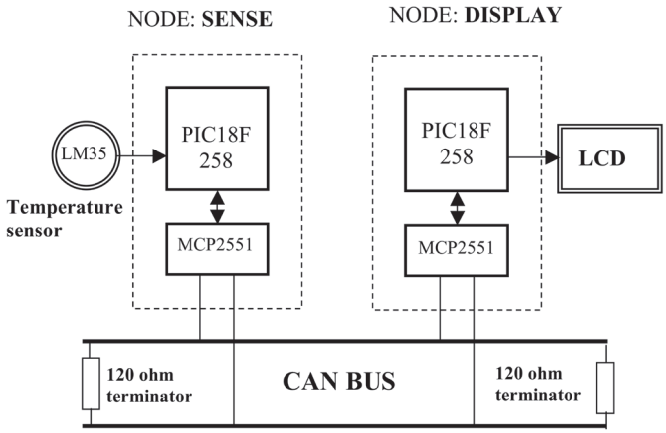


Figure 11.29 Block diagram of the project

The CAN bus timing parameters are assumed to be same as those in Project 1. The circuit diagram of the project is shown in Figure 11.30. The microcontrollers are operated from 8 MHz crystal. External reset is provided by connecting the MCLR inputs to a push-button switch. Analog channel 0 of node SENSE is connected to a LM35DZ type temperature sensor. The output voltage of this sensor is proportional to the temperature and is given by 10mV/°C. Thus, for example, at 20°C the output voltage is 200 mV. A standard text based LCD display is connected to PORT C of node DISPLAY. This node receives the current temperature every second from node SENSE and then displays it on the LCD.

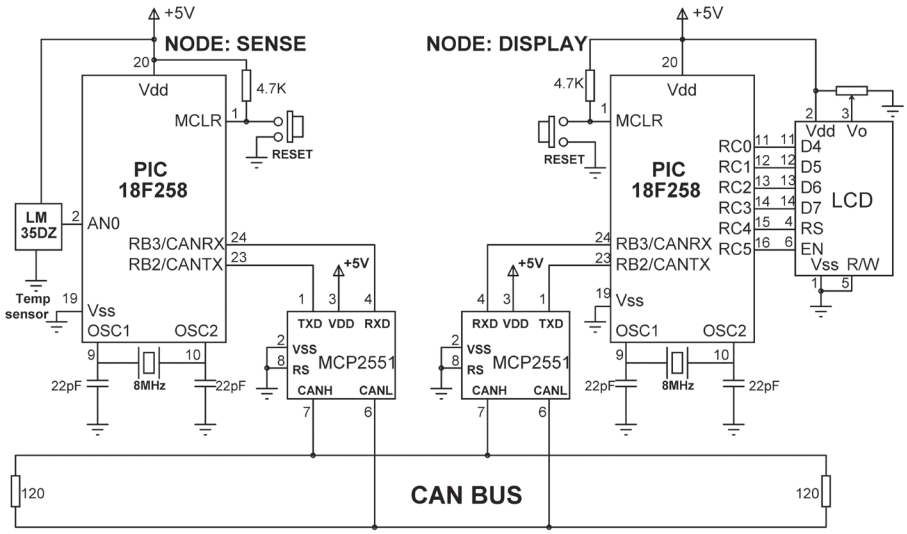


Figure 11.30 Circuit diagram of the project

The operation of the system is described by the Program Description Language (PDL) in Figure 11.31. Node SENSE is given the *message identifier number 3*, and node DISPLAY is given the *message identifier number 5*.

At the beginning, both nodes are initialized and the timing parameters are loaded into the microcontroller CAN modules. Then, the nodes are placed into Configuration Mode, and the acceptance masks and acceptance filters are set as required. Then the nodes are put into normal operation mode. Both nodes have endless loops. Node SENSE sends the temperature data over the bus to node DISPLAY every second.

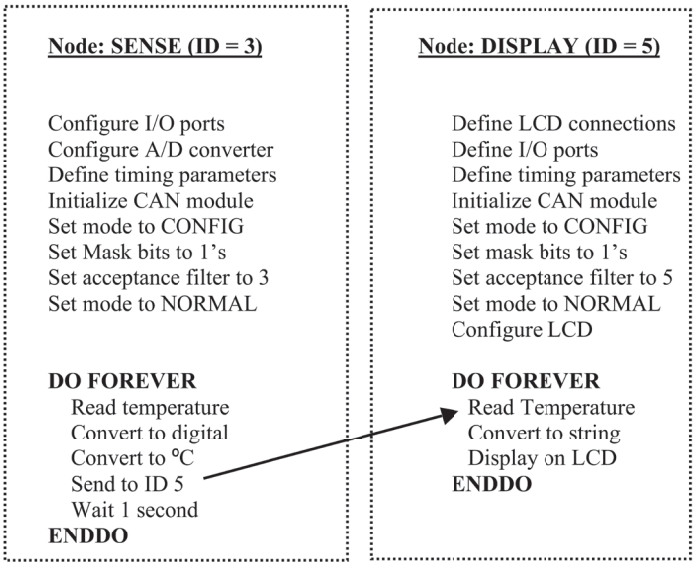


Figure 11.31 Operation of the system

Node SENSE Program

The program listing of node SENSE is shown in Figure 11.32. At the beginning of the program the port directions and the A/D module are configured. Then the CAN bus is initialized, acceptance masks and acceptance filter are set and the CAN module is put into NORMAL operational mode. The rest of the program is an endless loop formed using a for statement. Inside this loop the temperature is read from analog channel 0, converted into millivolts, then to ºC, and transmitted over the CAN bus with message identifier 5. Variables mV and temperature are set to be of type float. Variable mV is the voltage in millivolts output from the temperature sensor. Variable temperature is the actual ambient temperature, obtained by dividing the output voltage of the sensor by 10. The above process is repeated forever with a one second delay between each loop.

In this example, as before, the CAN module is operated in standard protocol mode (STD).

```
/*****************************************************************

                    CAN BUS EXAMPLE - NODE: SENSE
                    =============================

This is node SENSE of the CAN bus example. In this project a
PIC18F258 type microcontroller and an MCP2551 type CAN bus
Transceiver chip is used to connect the microcontroller to the CAN
bus.
The microcontroller is operated from an 8MHz crystal with an exter-
nal reset button.

Pin CANRX and CANTX of the microcontroller are connected to pins RXD
and TXD of the transceiver chip respectively. Pins CANH and CANL of
the transceiver chip are connected to the CAN bus.

An LM35DZ type analog temperature sensor is connected to port AN0
of the microcontroller. The microcontroller reads the temperature
and sendsit every second to the CAN bus. Node DISPLAY receives the
temperature value and displays it on an LCD.

CAN speed parameters are:

  Microcontroller clock: 8MHz
  CAN Bus bit rate:      125 Kb/s
  Sync_Seg:              1
  Prop_Seg:              1
  Phase_Seg1:            3
  Phase_Seg2:            3
  SJW:                   3
  BRP:                   8


Author:                  Dogan Ibrahim
Date:                    October 2011
File:                    SENSE.C

*****************************************************************/
```

```
void main()
{
     unsigned char sdata[8];
     unsigned short init_flag, send_flag;
     char SJW, BRP, Phase_Seg1, Phase_Seg2, Prop_Seg;
     unsigned int temp;
     long SENSE_ID, DISPLAY_ID;
     float mV, temperature;
     long id, mask;

     TRISA = 0xFF;       // PORT A are inputs
     TRISB = 0x08;       // RB2 is output, RB3 is input
     SENSE_ID = 3;
     DISPLAY_ID = 5;

//
// Configure A/D converter
//
     ADCON1 = 0x80;
//
// CAN BUS Timing Parameters
//
     SJW = 3;
     BRP = 8;
     Phase_Seg1 = 3;
     Phase_Seg2 = 3;
     Prop_Seg = 1;

     init_flag = _CAN_CONFIG_SAMPLE_THRICE     &
                 _CAN_CONFIG_PHSEG2_PRG_ON     &
                 _CAN_CONFIG_STD_MSG           &
                 _CAN_CONFIG_DBL_BUFFER_ON     &
                 _CAN_CONFIG_VALID_STD_MSG     &
                 _CAN_CONFIG_LINE_FILTER_OFF;

     send_flag = _CAN_TX_PRIORITY_0            &
                 _CAN_TX_STD_FRAME             &
                 _CAN_TX_NO_RTR_FRAME;
```

```
//
// Initialise CAN module
//
      CANInitialize(SJW, BRP, Phase_Seg1, Phase_Seg2, Prop_Seg,
                    init_flag);
//
// Set CAN CONFIG mode
//
      CANSetOperationMode(_CAN_MODE_CONFIG, 0xFF);

      mask = -1;
//
// Set all MASK1 bits to 1's
//
      CANSetMask(_CAN_MASK_B1, mask, _CAN_CONFIG_STD_MSG);
//
// Set all MASK2 bits to 1's
//
      CANSetMask(_CAN_MASK_B2, mask, _CAN_CONFIG_STD_MSG);
//
// Set id of filter B2_F3 to 3 (SENSE_ID)
//
      CANSetFilter(_CAN_FILTER_B2_F3,
                   SENSE_ID, _CAN_CONFIG_STD_MSG);
//
// Set CAN module to NORMAL mode
//
      CANSetOperationMode(_CAN_MODE_NORMAL, 0xFF);

//
// Program loop. Read the temperature from analog sensor
//

     for(;;)             // Endless loop
     {
         //
         // Now read the temperature
         //
         temp = Adc_Read(0);                  // Read temp
         mV = temp * 5000.0 / 1024.0;         // in mV
         temperature = mV/10.0;               // in degrees C
```

```
        //
        // Send the temperature to Node:Display
        //
        sdata[0] = (unsigned char)temperature;
        CANWrite(DISPLAY_ID, sdata, 1, send_flag);
        Delay_Ms(1000);                          // Wait 1 sec
    }                                            //end of for
}                                                // end of program
```

Figure 11.32 Program listing of node SENSE

Node: DISPLAY Program

The program listing of node DISPLAY is shown in Figure 11.33. At the beginning of the program the LCD connections are defined, and port I/O directions are configured, CAN timing parameters are specified, and the CAN module is initialized. After setting the masks and the acceptance filter, CAN module is put into normal operational mode. Then the LCD is configured and the program enters an endless loop. Inside this loop, the temperature value is read from the CAN bus, converted into string (using the built-in library function ByteToStr), and is displayed on the LCD display. In this example, the CAN module is operated in the standard protocol (STD).

Initially, the LCD displays the string "CAN BUS" for two seconds before displaying the temperature. Then, the temperature is displayed in the first row of the LCD in the format:

Temp = nn

```
/*****************************************************************

                    CAN BUS EXAMPLE - NODE: DISPLAY
                    ===============================

This is the DISPLAY node of the CAN bus example. In this project
a PIC18F258 type microcontroller is used. An MCP2551 type CAN bus
transceiver is used to connect the microcontroller to the CAN bus.
The microcontroller is operated from an 8MHz crystal with an exter-
nal reset button.
```

```
Pin CANRX and CANTX of the microcontroller are connected to pins
RXD and TXD of the transceiver chip respectively. Pins CANH and
CANL of the transceiver chip are connected to the CAN bus.

An LCD is connected to PORT C of the microcontroller. The ambient
temperature is read from node SENSE over the CAN bus and is dis-
played on the LCD.

The LCD is connected to the microcontroller as follows:

Microcontroller      LCD

    RC0              D4
    RC1              D5
    RC2              D6
    RC3              D7
    RC4              RS
    RC5              EN

CAN speed parameters are:

  Microcontroller clock: 8MHz
  CAN Bus bit rate:      125 Kb/s
  Sync_Seg:              1
  Prop_Seg:              1
  Phase_Seg1:            3
  Phase_Seg2:            3
  SJW:                   3
  BRP:                   8

Author:                  Dogan Ibrahim
Date:                    October 2011
File:                    DISPLAY.C\

*******************************************************************/
```

```
//
// LCD module connections
//
sbit LCD_RS at RC4_bit;
sbit LCD_EN at RC5_bit;
sbit LCD_D4 at RC0_bit;
sbit LCD_D5 at RC1_bit;
sbit LCD_D6 at RC2_bit;
sbit LCD_D7 at RC3_bit;
//
// LCD pin directions
//
sbit LCD_RS_Direction at TRISC4_bit;
sbit LCD_EN_Direction at TRISC5_bit;
sbit LCD_D4_Direction at TRISC0_bit;
sbit LCD_D5_Direction at TRISC1_bit;
sbit LCD_D6_Direction at TRISC2_bit;
sbit LCD_D7_Direction at TRISC3_bit;
// End LCD module connections


void main()
{
     unsigned char temperature, rdata[8];
     unsigned short init_flag, len, rd_flag, read_flag;
     char SJW, BRP, Phase_Seg1, Phase_Seg2, Prop_Seg, txt[4];
     long id, mask, SENSE_ID, DISPLAY_ID;

     TRISC = 0;          // PORT C are outputs (LCD)
     TRISB = 0x08;       // RB2 is output, RB3 is input
     SENSE_ID = 3;
     DISPLAY_ID = 5;

//
// CAN BUS Parameters
//
     SJW = 3;
     BRP = 8;
     Phase_Seg1 = 3;
     Phase_Seg2 = 3;
     Prop_Seg = 1;
```

```
     init_flag = _CAN_CONFIG_SAMPLE_THRICE      &
                 _CAN_CONFIG_PHSEG2_PRG_ON      &
                 _CAN_CONFIG_STD_MSG            &
                 _CAN_CONFIG_DBL_BUFFER_ON      &
                 _CAN_CONFIG_VALID_STD_MSG      &
                 _CAN_CONFIG_LINE_FILTER_OFF;

     read_flag = 0;
//
// Initialise CAN module
//
 CANInitialize(SJW, BRP, Phase_Seg1, Phase_Seg2, Prop_Seg,
                   init_flag);
//
// Set CAN CONFIG mode
//
     CANSetOperationMode(_CAN_MODE_CONFIG, 0xFF);

      mask = -1;
//
// Set all MASK1 bits to 1's
//
     CANSetMask(_CAN_MASK_B1, mask, _CAN_CONFIG_STD_MSG);
//
// Set all MASK2 bits to 1's
//
     CANSetMask(_CAN_MASK_B2, mask, _CAN_CONFIG_STD_MSG);
//
// Set id of filter B2_F3 to 3 (DISPLAY_ID)
//
      CANSetFilter(_CAN_FILTER_B2_F3,
                   DISPLAY_ID, _CAN_CONFIG_STD_MSG);
//
// Set CAN module to NORMAL mode
//
      CANSetOperationMode(_CAN_MODE_NORMAL, 0xFF);


//
// Configure LCD
//
```

```
     Lcd_Init();
     Lcd_Cmd(_LCD_CLEAR);                    // Clear LCD
     Lcd_Out(1,1,"CAN BUS");                 // Display on LCD
     Delay_ms(2000);                         // Wait for 2 second

//
// Program loop. Read the temperature from Node:SENSE and display
// on the LCD
//
     for(;;)                                 // Endless loop
     {
          //
          // Get temperature from node:SENSE
          //
          rd_flag = CANRead(&id, rdata, &len, &read_flag);
          if(rd_flag != 0 && id == DISPLAY_ID)
          {
               Lcd_Cmd(_LCD_CLEAR);
               Lcd_Out(1,1, "Temp = ");
               temperature = rdata[0];
               ByteToStr(temperature,txt);   // Convert to string
               Lcd_Out(1, 8, txt);           // Output to LCD
          }
     }

}
```

Figure 11.33 Program listing of node DISPLAY

Note that the EasyPIC6 development board uses PORT B for the LCD display by default and thus, an external LCD display was connected to PORT C of the development board to display the temperature (it was also possible to use the second LCD on the board which is based on serial data, or to use the on-board graphics display).

11.7.23 CAN BUS PROJECT 3

This project is slightly more complex than the previous two projects. In this project, as before, two nodes are used. The first node, called the **SENSORS** has two sensors connected to it: a temperature sensor, and a pressure sensor. The second node, called the **WEATHER** has an LCD connected to it. Node SENSORS reads the ambient temperature and pressure and sends the readings to node WEATHER whenever data request is made by node WEATHER. Node WEATHER sends character "R" to request the temperature and pressure data. The temperature is displayed on the first row of the LCD, and the pressure is displayed on the second row of the LCD in the following formats:

T = nn
P = nnnn

Figure 11.33 shows the block diagram of the project.

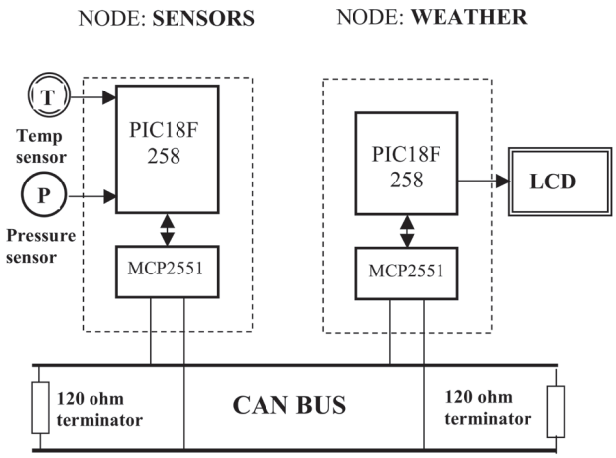


Figure 11.34 Block diagram of the project

The CAN bus timing parameters are assumed to be same as those in Project 1 and Project 2. The circuit diagram of the project is shown in Figure 11.34. The microcontrollers are operated from 8 MHz crystal. External reset is provided by connecting the MCLR inputs to a push-button switch. Analog channel 0 of node SENSORS is connected to a LM35DZ type temperature sensor. The output voltage of this sensor is proportional to the temperature and is given by 10mV/°C. Thus, for example, at 20°C the output voltage is 200 mV. Similarly, Analog channel 1 of node SENSORS is connected to a MPX4115A type pressure sensor chip. This is either a 6-pin or an 8-pin chip. In this project, the 6-pin version is used, having the following pin configuration (see Figure 11.35):

Pin	Description
1	Output voltage
2	Ground
3	+5V supply
4-6	Not used

The output voltage of the pressure sensor is given by:

$$V = 5.0 * (0.009 * kPa - 0.095) \tag{11.1}$$

or

$$kPa = \frac{\frac{V}{5.0} + 0.095}{0.009} \tag{11.2}$$

where,

kPa = atmospheric pressure (kilopascals)
V = output voltage of the sensor (volts)

The atmospheric pressure is usually shown in millibars. At sea level and at 15°C, the atmospheric pressure is 1010.3 millibars. In equation (11.2) the pressure is given in kPa. To convert kPa to millibars we have to multiply Equation (11.2) by 10. Thus,

$$mb = 10x\frac{\frac{V}{5.0} + 0.095}{0.009} \tag{11.3}$$

or,

$$mb = \frac{2.0V + 0.95}{0.009} \tag{11.4}$$

We can use Equation (11.4) to calculate the atmospheric pressure by measuring the output voltage of the pressure sensor. For example, if the output voltage is 4.0V, then the pressure is:

$$mb = \frac{2.0V + 0.95}{0.009} = \frac{2.0x4.0 + 0.95}{0.009} = 994.4\,\text{mb}$$

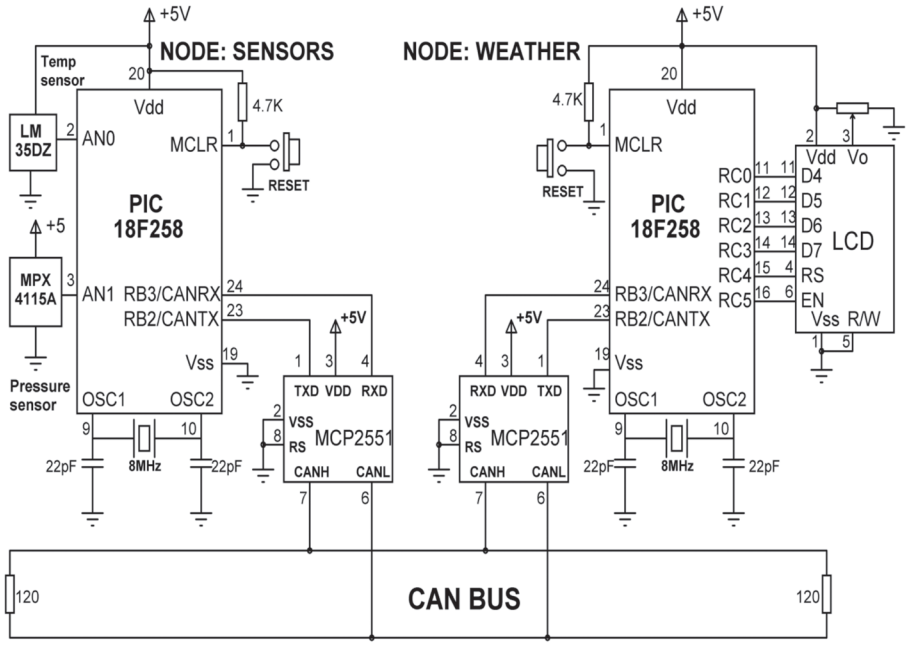


Figure 11.35 Circuit diagram of the project

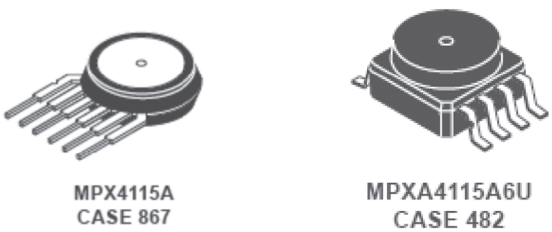


Figure 11.36 MPX4115A pressure sensors

A standard text based LCD display is connected to PORT C of node WEATHER, where the temperature is displayed on the first row and the pressure is displayed on the second row. The data is requested every second by node WEATHER. Note that on the EasyPIC6 development board the LCD is assigned to PORT B pins by default. But as PORT B is used by the CAN I/O functions of the microcontroller, it was necessary to connect an external LCD to port C pins as shown in Figure 11.34.

The operation of the system is described by the Program Description Language (PDL) in Figure 11.36. Node SENSORS is given the *message identifier number 30*, and node WEATHER is given the *message identifier number 50*. At the beginning, both nodes are initialized and the timing parameters are loaded into the microcontroller CAN modules. In addition, the LCD connection pins are defined in node WEATHER. Then, the nodes are placed into Configuration Mode, and the acceptance masks and acceptance filters are set as required. Then the nodes are put into normal operation mode. Both nodes have endless loops. Node SENSORS waits to receive commands from node WEATHER. The command is in the form of a single character. Character "R" sent by node WEATHER is a request for the temperature and pressure data (the request is sent every second). Upon receiving a valid command, node SENSORS reads data from both sensors, converts this data to digital forms, and then sends it to node WEATHER where it is displayed on the appropriate row of the LCD.

Node SENSORS Program

The program listing of node SENSORS is shown in Figure 11.37. At the beginning of the program the port directions and the A/D module are configured. Then the CAN bus is initialized, acceptance masks and acceptance filter are set and the CAN module is put into NORMAL operational mode. The rest of the program is an endless loop formed using a ***for*** statement. Inside this loop the program waits to receive a command from node WEATHER. After receiving command "R", the temperature is read from Analog channel 0, converted into digital, and then to °C and is stored in element 0 of array **srdata**. Similarly, the pressure is read from Analog channel 1, converted into digital, and to mb and is stored in elements 1 and 2 of array **srdata**. Note that the temperature is stored as a single byte as it is always less than 255, but the pressure is stored as an integer in two bytes as it is always greater than 255.

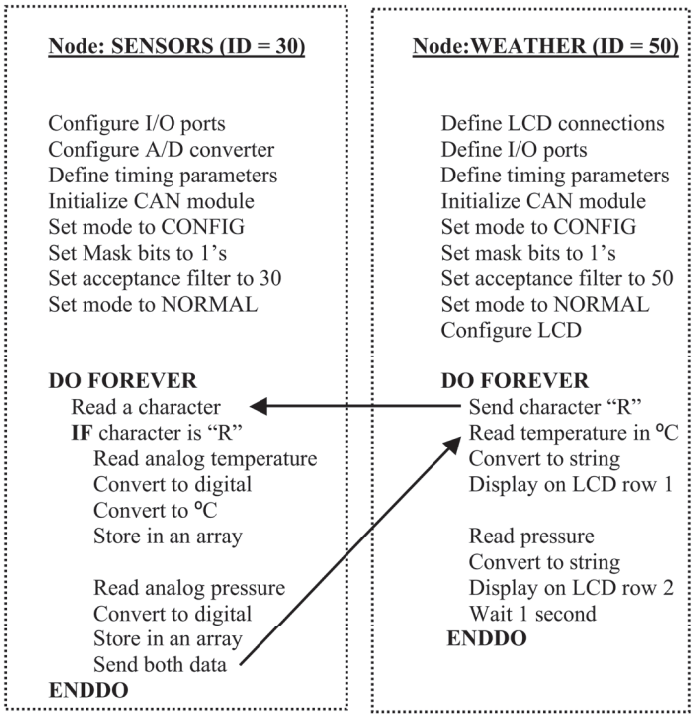


Figure 11.37 Operation of the system

```
/*****************************************************************

                  CAN BUS EXAMPLE - NODE: SENSORS
                 ================================

This is the SENSORS node of the CAN bus example. In this project a
PIC18F258 type microcontroller is used. An MCP2551 type CAN bus
transceiver is used to connect the microcontroller to the CAN bus.
The microcontroller is operated from an 8MHz crystal with an exter-
nal reset button.

Pin CANRX and CANTX of the microcontroller are connected to pins
RXD and TXD of the transceiver chip respectively. Pins CANH and
CANL of the transceiver chip are connected to the CAN bus.
```

```
An LM35DZ type analog temperature sensor is connected to port AN0
of the microcontroller. When a request is received this node reads
the
Temperature, formats it and then sends the temperature value to
Node:WEATHER on the CAN bus.

Similarly, a MPX4115A type atmospheric pressue sensor chip is
Connected to port AN1 of the microcontroller. When a request is re-
ceived
This node reads the pressure, formats it and then sends the pres-
sure value to node WEATHER on the CAN bus.

CAN speed parameters are:

  Microcontroller clock: 8MHz
  CAN Bus bit rate:      125 Kb/s
  Sync_Seg:              1
  Prop_Seg:              1
  Phase_Seg1:            3
  Phase_Seg2:            3
  SJW:                   3
  BRP:                   8


Author:                  Dogan Ibrahim
Date:                    January, 2011
File:                    SENSORS.C

************************************************************/

void main()
{
     unsigned char msb,lsb,temperature, srdata[8];
     unsigned short init_flag, send_flag, dt, len, read_flag;
     char SJW, BRP, Phase_Seg1, Phase_Seg2, Prop_Seg, txt[4];
     unsigned int temp;
     unsigned int pressure;
     long id, SENSORS_ID, WEATHER_ID, mask;
     float mV, V, Pmb;
```

```
     TRISA = 0xFF;       // PORT A are inputs
     TRISB = 0x08;       // RB2 is output, RB3 is input
     SENSORS_ID = 30;
     WEATHER_ID = 50;

//
// Configure A/D converter
//
     ADCON1 = 0x80;
//
// CAN BUS Timing Parameters
//
     SJW = 3;
     BRP = 8;
     Phase_Seg1 = 3;
     Phase_Seg2 = 3;
     Prop_Seg = 1;

     init_flag = _CAN_CONFIG_SAMPLE_THRICE      &
                 _CAN_CONFIG_PHSEG2_PRG_ON      &
                 _CAN_CONFIG_STD_MSG            &
                 _CAN_CONFIG_DBL_BUFFER_ON      &
                 _CAN_CONFIG_VALID_STD_MSG      &
                 _CAN_CONFIG_LINE_FILTER_OFF;

     send_flag = _CAN_TX_PRIORITY_0             &
                 _CAN_TX_STD_FRAME              &
                 _CAN_TX_NO_RTR_FRAME;

     read_flag = 0;
//
// Initialise CAN module
//
     CANInitialize(SJW, BRP, Phase_Seg1, Phase_Seg2, Prop_Seg,
                   init_flag);
//
// Set CAN CONFIG mode
//
     CANSetOperationMode(_CAN_MODE_CONFIG, 0xFF);

     mask = -1;
```

```
//
// Set all MASK1 bits to 1's
//
     CANSetMask(_CAN_MASK_B1, mask, _CAN_CONFIG_STD_MSG);
//
// Set all MASK2 bits to 1's
//
     CANSetMask(_CAN_MASK_B2, mask, _CAN_CONFIG_STD_MSG);
//
// Set id of filter B2_F3 to 30 (SENSORS_ID)
//
     CANSetFilter(_CAN_FILTER_B2_F3, SENSORS_ID,
                            _CAN_CONFIG_STD_MSG);
//
// Set CAN module to NORMAL mode
//
     CANSetOperationMode(_CAN_MODE_NORMAL, 0xFF);


//
// Program loop.
//
     for(;;)                                   // Endless loop
     {
          //
          // Wait until a request (R) is received
          //
          dt = CANRead(&id, srdata, &len, &read_flag);
          if(dt != 0 && id == SENSORS_ID)
          {
               if(srdata[0] == 'R')            // Data request
               temp = Adc_Read(0);             // Read temp
               mV = temp * 5000.0 / 1024.0;    // in mV
               temperature = (int)(mV/10);     // in degrees C
               //
               // Put temperature into srdata[0]
               //
               srdata[0] = temperature;
               //
               // Now Pressure
               //
```

```
            temp = Adc_Read(1);
            mV = temp * 5000.0/1024.0;      // in mV
            V = mV /1000.0;
            Pmb = (2.0*V + 0.95) / 0.009;
            pressure = (unsigned int)Pmb;
            msb = pressure/256;
            lsb = pressure % 256;           // As an integer

        //
        // Put the Pressure into srdata[1] and srdata[2]
        //
            srdata[1] = msb;
            srdata[2] = lsb;
        //
        // Now send the temperature+pressure to node WEATHER
        //
            CANWrite(WEATHER_ID, srdata, 3, send_flag);
        }
}
```

Figure 11.38 Program listing of node SENSORS

Node WEATHER Program

The program listing of node WEATHER is shown in Figure 11.38. At the beginning of the program the port directions and the A/D module are configured. The LCD pin connections are defined and then the CAN bus is initialized, acceptance masks and acceptance filter are set and the CAN module is put into NORMAL operational mode. The rest of the program is an endless loop formed using a ***for*** statement. Inside this loop the program sends a command for data request (command "R") to node SENSORS. Both the temperature and the pressure data are received at the same time. After receiving the temperature data, it is converted into string and is displayed on the first row of the LCD (as "T = nn"). Similarly, the pressure data, it is converted into string and is displayed on the second row of the LCD (as "P = nnnn"). The above process is repeated forever after one second delay. Note that when the function *IntToStr* is used to convert a long variable to a string, the output array is 7 bytes long, filled with leading spaces. Function *Ltrim* is used to remove these leading spaces.

```
/*****************************************************************

                    CAN BUS EXAMPLE - NODE: WEATHER
                    ===============================

This is the node WEATHER of the CAN bus example. In this project
a PIC18F258 type microcontroller is used. An MCP2551 type CAN bus
transceiver is used to connect the microcontroller to the CAN bus.
The microcontroller is operated from an 8MHz crystal with an exter-
nal reset button.

Pin CANRX and CANTX of the microcontroller are connected to pins
RXD and TXD of the transceiver chip respectively. Pins CANH and
CANL of the transceiver chip are connected to the CAN bus.

An LCD is connected to PORT C of the microcontroller. The ambient
temperature is read from node SENSORS over the CAN bus and is
displayed on the LCD.

The LCD is connected to the microcontroller as follows:

Microcontroller      LCD

    RC0              D4
    RC1              D5
    RC2              D6
    RC3              D7
    RC4              RS
    RC5              EN

CAN speed parameters are:

    Microcontroller clock:  8MHz
    CAN Bus bit rate:       125 Kb/s
    Sync_Seg:               1
    Prop_Seg:               1
    Phase_Seg1:             3
    Phase_Seg2:             3
    SJW:                    3
    BRP:                    8
```

```
Author:   Dogan Ibrahim
Date:     October 2011
File:     WEATHER.C

*******************************************************************/

//
// LCD module connections
//
sbit LCD_RS at RC4_bit;
sbit LCD_EN at RC5_bit;
sbit LCD_D4 at RC0_bit;
sbit LCD_D5 at RC1_bit;
sbit LCD_D6 at RC2_bit;
sbit LCD_D7 at RC3_bit;
//
// LCD pin directions
//
sbit LCD_RS_Direction at TRISC4_bit;
sbit LCD_EN_Direction at TRISC5_bit;
sbit LCD_D4_Direction at TRISC0_bit;
sbit LCD_D5_Direction at TRISC1_bit;
sbit LCD_D6_Direction at TRISC2_bit;
sbit LCD_D7_Direction at TRISC3_bit;
// End LCD module connections


void main()
{
     unsigned char temperature, srdata[8];
     unsigned short init_flag, len, rd_flag, read_flag, send_flag;
     char SJW, BRP, Phase_Seg1, Phase_Seg2, Prop_Seg, txt[4], op[7];
     long id, mask, SENSORS_ID, WEATHER_ID;
     unsigned int pressure;
     char *res;

     TRISC = 0;          // PORT C are outputs (LCD)
     TRISB = 0x08;       // RB2 is output, RB3 is input
     SENSORS_ID = 30;
     WEATHER_ID = 50;
```

```
//
// CAN BUS Parameters
//
     SJW = 3;
     BRP = 8;
     Phase_Seg1 = 3;
     Phase_Seg2 = 3;
     Prop_Seg = 1;

     init_flag = _CAN_CONFIG_SAMPLE_THRICE     &
                 _CAN_CONFIG_PHSEG2_PRG_ON     &
                 _CAN_CONFIG_STD_MSG           &
                 _CAN_CONFIG_DBL_BUFFER_ON     &
                 _CAN_CONFIG_VALID_STD_MSG     &
                 _CAN_CONFIG_LINE_FILTER_OFF;

     send_flag = _CAN_TX_PRIORITY_0            &
                 _CAN_TX_STD_FRAME             &
                 _CAN_TX_NO_RTR_FRAME;

     read_flag = 0;
//
// Initialise CAN module
//
     CANInitialize(SJW, BRP, Phase_Seg1, Phase_Seg2, Prop_Seg,
                   init_flag);
//
// Set CAN CONFIG mode
//
     CANSetOperationMode(_CAN_MODE_CONFIG, 0xFF);

     mask = -1;
//
// Set all MASK1 bits to 1's
//
     CANSetMask(_CAN_MASK_B1, mask, _CAN_CONFIG_STD_MSG);
//
// Set all MASK2 bits to 1's
//
      CANSetMask(_CAN_MASK_B2, mask, _CAN_CONFIG_STD_MSG);
```

```
//
// Set id of filter B2_F3 to 3 (DISPLAY_ID)
//
     CANSetFilter(_CAN_FILTER_B2_F3,
                  WEATHER_ID, _CAN_CONFIG_STD_MSG);
//
// Set CAN module to NORMAL mode
//
     CANSetOperationMode(_CAN_MODE_NORMAL, 0xFF);

//
// Configure LCD
//
     Lcd_Init();
     Lcd_Cmd(_LCD_CLEAR);                        // Clear LCD
     Lcd_Out(1,1,"CAN BUS");                     // Display on LCD
     Delay_ms(2000);                             // Wait for 2 second

//
// Program loop. Read the temperature from Node:SENSORS and display
// on the LCD
//
     for(;;)                                     // Endless loop
     {
          //
          // Request from node SENSORS
          //
          srdata[0] = 'R';
          // Command R
          CANWrite(SENSORS_ID, srdata, 1, send_flag);
          Delay_Ms(1000);
          //
          // Get data from node:SENSORS
          //
          rd_flag = CANRead(&id, srdata, &len, &read_flag);
          if(rd_flag != 0 && id == WEATHER_ID)
          {
```

```
        Lcd_Cmd(_LCD_CLEAR);
        temperature = srdata[0];
        ByteToStr(temperature,txt);        // Convert to string
        Lcd_Out(1,1,"T = ");
        Lcd_Out(1, 5, txt);                // Output to LCD
        //
        // Now Pressure
        //
        pressure = 256*srdata[1]+srdata[2];// Pressure as long
        IntToStr(pressure, op);            // Convert to string
        res = Ltrim(op);                   // remove spaces
        Lcd_Out(2,1,"P = ");
        Lcd_Out(2, 5, res);
        // Output to LCD
    }

    Delay_Ms(1000);                        // Wait 1 sec
}
```

Figure 11.39 Program listing of node WEATHER

TESTING

Testing the project is as in Project 1. The project was built using the mikroElektronika CAN Bus Communications Kit. After compiling and loading the programs into the program memories of the two EasyPIC6 boards, power was applied to both boards through the USB ports. The temperature and the pressure were displayed on the LCD as shown in Figure 11.40 (an external LCD was used since the EasyPIC6 development board uses the PORT B pins for the LCD and these pins are required for the CAN I/O interface).

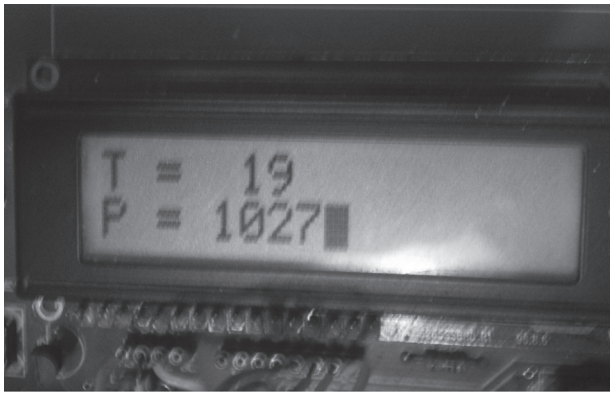


Figure 11.40 Displaying the temperature and pressure

11.8 Summary

This Chapter has described the basic architecture of the PIC microcontrollers. PIC18F452 is taken as an example microcontroller. The reset circuitry, clock circuitry, and the parallel I/O ports of the microcontroller have been explained.

In addition, the basic principles of the mikroC language has been described with reference to the variable types, arrays, structures, pointers, user functions, and built-in LCD functions. An example simple flashing LED project is given to show how a project can be developed and how the program code can be compiles and tested on a microcontroller development board.

The Chapter has also described the basic features of the PIC18F258 microcontroller which has a built-in CAN module, as this is the microcontroller used in the example projects. mikroC language CAN bus library functions are very important for developing CAN bus based projects using microcontrollers. The important CAN bus library functions have been described.

Finally, example built and working CAN bus based projects are given in the Chapter where two nodes exchange data using the PIC18F258 microcontroller.

Chapter 12. On Board Diagnostics (OBD)

On Board Diagnostics (OBD), when used in automotive context, refers to a vehicle's self-diagnostic, where the vehicle owner (or a repair technician) can access and monitor the state of various modules within the vehicle. Although the early OBD was very limited, the modern OBD tools enable a person to get real-time information about the state of various modules in a vehicle. An OBD basically communicates with the on-board vehicle computers and displays the state of various parameters. For example, the engine rpm, engine temperature, oxygen sensor, fuel type and so on. Most OBD functions are read-only where the state of a parameter is searched and its value is displayed, usually in HEX.

First OBD tools appeared in late 1980s with the requirement of the California Air Resources Board, stating that all new vehicles sold in California should have OBD capabilities. This was later named as OBD I. Later in early 1990s, The Society of Automotive Engineers (SAE) recommended a standard diagnostic connector and a set of diagnostic test signals. In 1996 the importance of OBD was realized and an updated OBD, the OBD II specification, became mandatory for all cars sold in the United States. It was in the year 2001 that the OBD became mandatory for all vehicles sold in the European Union (the version of OBD II in Europe is called EOBD).

Although the current OBD is at level II, its development went through the levels OBD I, and OBD 1.5. In this Chapter we shall be looking at the OBD II specifications and some of the tools (called "scan" tools) that can be used for OBD II analysis.

12.1 OBD II

OBD II standard specifies the type of connector used and its pin configuration, the electrical signalling protocols available, and the messaging formats. In addition, the OBD II standard provides a list of DTCs (Diagnostic Trouble Codes).

There are basically five non-compatible standards used in OBD II. These are:

- J1850 PWM
- J1850 VPW
- ISO 9141-2 (similar to LIN bus)
- KWP 2000 (or Keyword Protocol)
- ISO 15765 (CAN bus)

Most OBD scan tools are compatible with all the five standards, but it is always advisable to check the compatibility before using a scan tool.

The OBD II connector (see Figure 12.1) is a 16-pin female J1962 type connector, and is usually located within 2 feet of the steering wheel. Different pins are used for the five standards, although the power and the ground pins are common to all standards. The pin layout of the connector is:

1. –
2. Bus positive (J1850 PWM and VPW)
3. –
4. Chassis ground
5. Signal ground
6. CAN_H
7. K line (ISO 9141-2 and KWP 2000)
8. –
9. –
10. Bus negative (J1850 PWM)
11. –
12. –
13. –
14. CAN_L
15. L line (ISO 9141-2 and KWP 2000)
16. Battery voltage

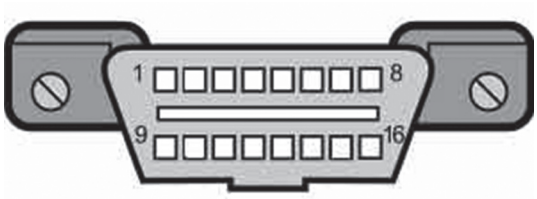


Figure 12.1 OBD II connector

The pins used by each standard can be summarized as follows:

SAE J1850 PWM

Pin 2: BUS +
Pin 5: Ground
Pin 10: BUS-
Pin 16: Battery

SAE J1850 VPW

Pin 2: BUS+
Pin 5: Ground
Pin 16: Battery
ISO 9141-2
Pin 5: Ground
Pin 7: K-line
Pin 15: L-line
Pin 16: Battery

KWP 2000

Pin 5: Ground
Pin 7: K-line
Pin 15: L-line (optional)
Pin 16: Battery

CAN

Pin 5: Ground
Pin 6: CAN_H
Pin 14: CAN_L
Pin 16: Battery

In general, we can tell what type of standard a given vehicle is using by looking at the OBD connector. Table 12.1 shows the pin that must be present for a given standard.

The SAE J1979 standard defines a method for requesting various diagnostics data from the engine control unit using standard parameters. A list of these parameters is known as the PIDs (Parameter Identification Numbers), which are defined in J1979. Note that manufacturers are not required to implement all the PIDs. For example, some of the PID codes are given below. Note that the PIDs are in two groups: Generic PIDs, and manufacturer specific PIDs. Generic PIDs usually start with letters P0, P2, P3, or U0, while the manufacturer specific PIDs start with letter P1, P3, or U1. Details of generic PIDs are available on the internet, while company specific PIDs can either be obtained from manufacturers' data sheets or some of them are available on the internet:

- P0001 Fuel Volume Regulator Control Circuit/Open
- P0002 Fuel Volume Regulator Control Circuit Range/Performance
- P0003 Fuel Volume Regulator Control Circuit Low
- P0004 Fuel Volume Regulator Control Circuit High
- P0005 Fuel Shutoff Valve "A" Control Circuit/Open
- P0006 Fuel Shutoff Valve "A" Control Circuit Low
- P0007 Fuel Shutoff Valve "A" Control Circuit High
- P0008 Engine Positions System Performance Bank 1
- P0009 Engine Position System Performance Bank 2

Table 12.1 OBD II connector identification (in addition to pin 5 and pin 16)

Standard	Pin 2	Pin 6	Pin 7	Pin 10	Pin 14	Pin 15
J1850 PWM	X			X		
J1850 VPW	X					
ISO 9141-2			X			X
KWP 2000						
CAN		X			X	

Notice that some PID codes refer to BANK numbers. BANK 1 is where cylinder No 1 is, or the driver's side (left), and Bank 2 is the opposite side (right).

Currently OBD-II defines 9 modes of operation. For example, in Mode 1, data is requested by PID. Sending 01XX(CR), and the vehicle computer returns XX = PID.

Table 12.2 shows (see SAE J1979 document for further details) the data returned with their min and max values for some of the PID codes (Note that the PID codes are specified in HEX).

For example, to get the current Fuel Pressure value, we have to send the PID code 0A. In return, we receive a single byte response. This response value should be multiplied by 3 to find the actual fuel pressure in kPa. The min and max values of the fuel pressure are 0 and 765 kPa respectively.

Table 12.2 Some of the data returned with min and max values

PID (hex)	Data bytes returned	Description	Min value	Max value	Units	Formula
03	2	Fuel system status				Bit encoded.
04	1	Calculated engine load value	0	100	%	A*100/255
05	1	Engine coolant temperature	-40	215	°C	A-40
06	1	Short term fuel % trim—Bank 1	-100 (Rich)	99.22 (Lean)	%	(A-128) * 100/128
07	1	Long term fuel % trim—Bank 1	-100 (Rich)	99.22 (Lean)	%	(A-128) * 100/128
08	1	Short term fuel % trim—Bank 2	-100 (Rich)	99.22 (Lean)	%	(A-128) * 100/128
09	1	Long term fuel % trim—Bank 2	-100 (Rich)	99.22 (Lean)	%	(A-128) * 100/128
0A	1	Fuel pressure	0	765	kPa (gauge)	A*3

It is easy to tell whether or not a vehicle is OBD II compliant. In general, all new cars sold in America since 1996, and all new cars sold in Europe since 2001 should be OBD II compliant. The OBD II compliance can also be checked by locating the metal *Vehicle Emission Control Information* label located under the hood. The top right hand corner of this label should say if the car is OBD II compliant (OBD II certified).

The OBD II connector is normally located inside the car, usually at the driver's side, usually a few feet away from the instrument panel. The exact location of the conenctor can be found on the internet or from manufacturers' data sheets.

12.2 Hand-Held OBD II SCAN Tools

There are many hand-held OBD II SCAN tools available. Two examples are shown in Figures 12.2 and 12.3. Note that the scan tools are supplied with OBD II connectors. The operation of these tools differ depending upon the model and the cost of the tool. Some models have several buttons where a PID code can be entered and its value can then be displayed on an LCD. In some cheaper and simpler models, a scroll button and an LCD are provided and the diagnostic trouble codes can be displayed by scrolling through a menu. Using an expensive professsional hand-held scan tool, the OBD II operation is simply as follows:

- The technician enters the PID
- The scan tool sends it to the vehicle's bus
- A device on the bus recognizes the PID as one it is responsible for, and reports the value for that PID to the bus
- The scan tool reads the response, and shows it to the technician

Simple OBD II scan tools are mostly aimed at the non-professional hobby and consumer market. They may read simple error codes, possibly without translating the meaning, and reset some error codes.
Professional hand-held OBD II scan tools may possess more advanced functions, such as accessing advanced diagnostics, setting engine control unit parameters, real-time monitoring of engine parameters, and advanced diagnostics using graphs.
An example use of an OBD II scan tool is given later in this Chapter.

12.3 PC Based OBD II Scan Tools

A PC-based OBD II scan tool consists of a small hardware device and a dedicated software. One end of the device is connected to the OBD II plug of the vehicle, and the other end is connected to the USB port of a PC. The software receives and decodes the diagnostic data, and usually provides a display of the vehicle status on the screen. Some sophisticated tools provide graphical data which is easier to interpret. PC based OBD II tools are professional tools and they are usually much more expensive than the hand-held scan tools. Some companies offer free OBD II analysis software. e.g. Freediag, Opendiag, pyOBD, and so on. Figure 12.4 shows a screen shot from the pyOBD software.

Figure 12.2 Hand-held OBD II scan tool (Memoscanner U480)

Figure 12.3 Another hand-held OBD II scan tool

12.4 Data Logging

OBD II data loggers are designed to capture vehicle data while the vehicle is in normal operation. The collected data is normally analyzed later, offline. Data logging can be useful for monitoring and tuning the engine while it is running in normal operational mode. In addition, the logged data can be used as a black-box where the driver and vehicle conditions are monitored constantly. This could be useful to analyze the vehicle movements after an accident. Some insurance companies offer reduced premiums if OBD II data logging is used on a vehicle.

12.5 Example Using a Hand-Held OBD II Scan Tool

In this section the operation of a low-cost OBD II scan tool is demonstrated. The tool chosen is the Memoscanner U480. Figure 12.5 shows parts of the U480, and their functions are given below:

LCD DISPLAY: This is a 2-row 8-character display that indicates the test results.

ENTER BUTTON: Used to make a selection from the menu.

SCROLL BUTTON: This button is used to scroll through menu items, or to cancel a selection.

OBD II CONNECTOR: This is the standard OBD II connector (male plug).

The features of the U480 are:

- Works with all cars since 1996
- Supports CAN, VPW, PWM, ISO 9141-2, and KWP2000 protocols
- Reads and clears generic and manufacturer specific diagnostic trouble codes.
- Clears Malfunction Indicator Lamp (MIL), or sometimes called the CHECK ENGINE lamp.
- Displays Vehicle Identification Number (VIN)
- No battery is needed as the power is taken from the OBD II socket of the car

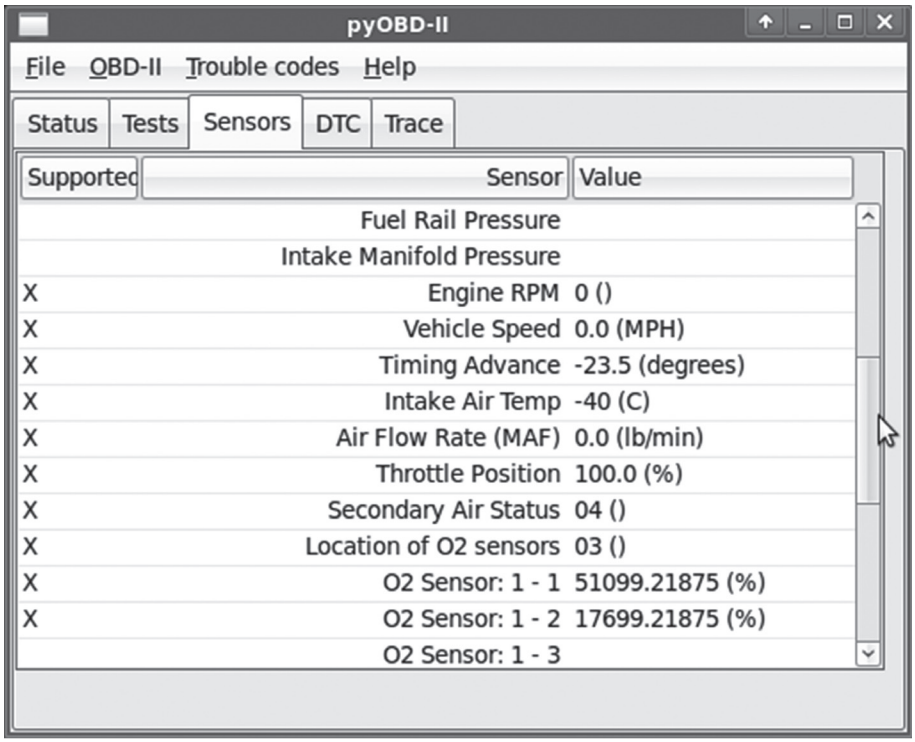


Figure 12.4 Screen shot from pyOBD software

Before going into details of using the U480 tool, it is worthwhile to look at some of the important parameters that can be read, set, or cleared by this tool.

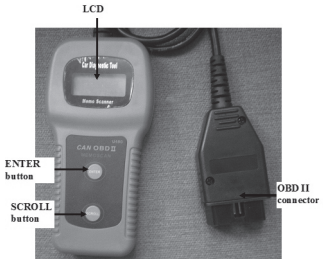


Figure 12.5 The U480 scan tool used

The Vehicle Identification Number

A Vehicle Identification Number (VIN) is a unique serial number used by the manufacturers to identify individual motor vehicles. A VIN consists of 17 characters, excluding characters I, i, O, o, Q, q to avoid confusion with numbers and letters. VIN is based on two standards, ISO 3779, and ISO 3780. Some manufacturers in European Union and USA use a somewhat modified standard. In general, the fields of a VIN are:

Digit 1 – 3: World Manufacturer ID (WMI)
Digit 4 – 9: Vehicle Descriptor Section (VDS)
Digit 9: Check digit
Digit 10 – 17: Vehicle Identifier Section (VIS)

The WMI identify the manufacturer of the vehicle using a code. The Society of Automotive Engineers (SAE) asigns WMI codes to countries and manufacturers.

The VDS is used to identify the vehicle type, model, body style and so on. In general each manufacturer has a unique method for using this field.

The check digit is used to validate a VIN.

The VIS is used to identify an individual vehicle, and in most cases consists of a simple serial number. In USA, the last five digits of VIS are numeric.

There are online sites on the internet where a given VIN can be decoded
(e.g. http://www.autocalculator.org).

Malfunction Indicator Lamp (MIL)

The MIL lamp (also called the CHECK ENGINE light) is an on-board diagnosic lamp used to notify the driver that there could be a serious problem with the vehicle. The MIL is found on the instrument panel and, when lit, it is either an amber or red colour. This light has two stages: steady, and flashing. A steady light indicates a minor fault, such as a failing sensor), while a flashing light indicates a severe fault (e.g. a misfire). The fault code related to the malfunction can be retrieved and also cleared using a scan tool. The MIL is usually indicated by a picture of the engine (see Figure 12.6), or a text such as "Service Engine Soon", or "Check Engine".

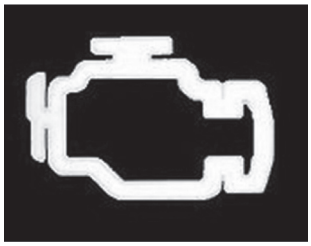


Figure 12.6 MIL lamp

Using the U480 Scan Tool

In this section the use of the U480 scan tool is demonstrated. The tool is connected to the OBD II connector of a BMW 320I SE model passenger car. The location of the OBD II connector is in the driver's area, just above the lever used to open the bonnet, and is covered with a plastic cover as shown in Figure 12.7. The steps are given below:

- Turn the ignition off
- Connect the scan tool to the OBD II connector (see Figure 12.7)
- Wait until the scan tool is initialized and the following message is displayed on the LCD:

MEMOSCAN
U480

- Turn the ignition on (do not start the engine)
- The scan tool will go through and display all the recognized protocols until the one used by the vehicle is identified.

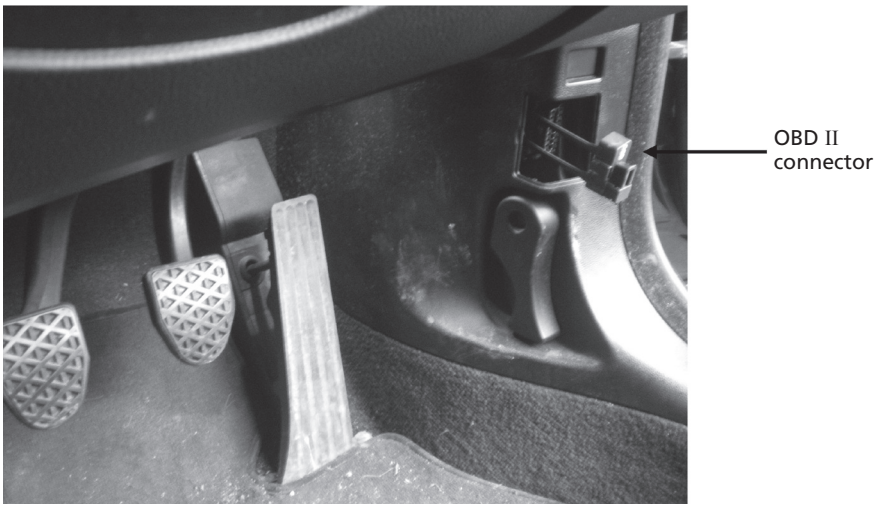


Figure 12.7 Location of the OBD II connector on BMW 320I SE

- When the display is idle, press ENTER. The scan tool will display the total number of diagnostic trouble codes (DTCs) and the overall I/M status (Inspection and Maintenance program legislated by the Government to meet clean-air standards). In this example, the following is displayed on the LCD to show that there are no trouble codes present:

DTC: 0
IM: YES

- To display the VIN, press SCROLL button until menu option 4 is displayed:

- Press the ENTER button. Part of the VIN will be displayed.
  Press the SCROLL button to see the remainder (in 3 screens):

The required VIN is: **WBAVA72030AH23957**

Notice that this VIN translates to:

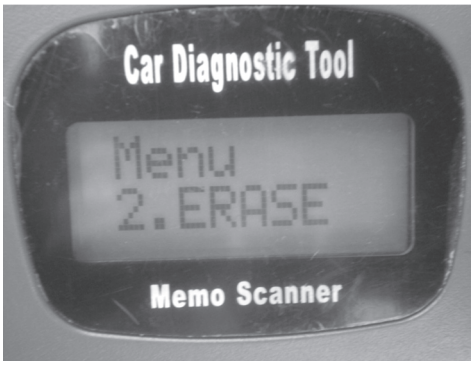
WMI= WBA (code for BMW, Munich)
VDS= VA720 (Manual)
Check digit = 3
VIS= 0AH23957 (European model, manufactured in Munich, Serial number = H23957)

Usually, one can find web sites on the internet that decode specific manufacturers' VINs. For example, the site http://www.bmw-z1.com/VIN/VINdecode-e.cgi decodes all BMW vehicle VINs.

- Check status of the MIL lamp. Scroll to MIL and press ENTER

- Rescan for any faults again if required. Scroll to menu RESCAN (menu item 5) and press ENTER

- Using the scan tool any fault codes can be erased if desired (menu option 2):



12.6 Summary

On Board Diagnostics has become an integral part of every vehicle. This Chapter has given the basic OBD II standards, and has also demonstrated how to use a hand-held OBD II scan tool. Examples of hand-held OBD II scan tools, and freely available PC software have been given.

Using a hand-held scan tool, or a PC based software and hardware, a novice or a professional technician can easily carry out diagnostic tests on a vehicle. PC based OBD II tools have the advantages that the display is much bigger, graphics data can easily be displayed, the memory is virtually unlimited, and vehicle data can continuously be logged while the vehicle is moving.

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Z

PRINCIPLES, PROJECTS, PROGRAMMING

CONTROLLER AREA NETWORK PROJECTS

Dogan Ibrahim

Dogan Ibrahim is a Fellow of the Institution of Electrical Engineers. He is the author of over 60 technical books, published by international famous publishers, such as Wiley, Butterworth, and Newnes. In addition, he is the author of over 250 technical papers, published in journals, and presented in seminars and conferences.

The Controller Area Network (CAN) was originally developed to be used as a vehicle data bus system in passenger cars. Today, CAN controllers are available from over 20 manufacturers, and CAN is finding applications in other fields, such as medical, aerospace, process control, automation, and so on.

This book is written for students, for practising engineers, for hobbyists, and for everyone else who may be interested to learn more about the CAN bus and its applications. The aim of this book is to teach you the basic principles of CAN networks and in addition the development of microcontroller based projects using the CAN bus. In summary, this book enables the reader to:

- Learn the theory of the CAN bus used in automotive industry
- Learn the principles, operation, and programming of microcontrollers
- Design complete microcontroller based projects using the C language
- Develop complete real CAN bus projects using microcontrollers
- Learn the principles of OBD systems used to debug vehicle electronics

You will learn how to design microcontroller based CAN bus nodes, build a CAN bus, develop high-level programs, and then exchange data in real-time over the bus. You will also learn how to build microcontroller hardware and interface it to LEDs, LCDs, and A/D converters.

The book assumes that the reader has some knowledge on basic electronics. Knowledge of the C programming language will be useful in later chapters of the book, and familiarity with at least one member of the PIC series of microcontrollers will be an advantage, especially if the reader intends to develop microcontroller based projects using the CAN bus.

ISBN 978-1-907920-04-2

9 781907 920042

Elektor International Media BV

www.elektor.com

elektor