CAN Node Manual

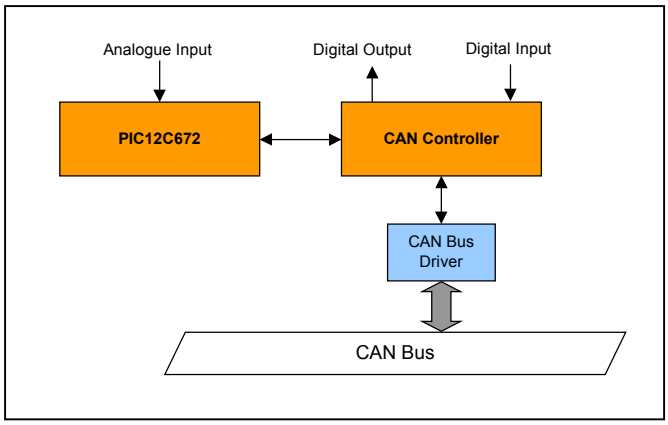
Version 1.0

Introduction

The idea behind this project was to have a simple node that could be placed in the field for remote control and monitoring via a CAN network. The node would have some I/O points together with some analog sensing inputs, for use with temperature sensors, position sensors etc. The node would have an address, to allow a number of different devices on the same network. The solution we have here, will address most of these requirements, together with the ability to further expand and download new programs to the unit.

The node could find many uses, as for example in a home monitoring system. A node could be placed in each room. One digital input, could be used as an input for the infra-red sensor, detecting movement in the room. Another digital input to detect whether the door or window was open. The analog input could be used as temperature monitor for the room. The digital output could be used to turn on the air-conditioning, turn on a light etc. The node would then reduce a lot of the wiring between the various devices and back to the main control point, as all wiring would be taken to the node, which would then connect by a 2 wire CAN bus to the main computer. This CAN bus could be looped around the house, thus minimising and simplify the control network.

Another application is in a greenhouse, where the temperature needs to be monitored at a number of points and the watering system then turned on at certain points based on the temperature or the amount of moisture in the soil.



The applications and ideas are endless, and the system is designed such that more sophisticated nodes with high number of I/O's and accessories can be placed on the same Bus. This is done by assigning each type of node a certain unit type, such that only certain commands specific to that unit type will be recognised. So when we are addressing a node, we first include the Unit type in the address and then the address of that specific unit. Up to 64 of these Basic Nodes can be included on the one CAN Bus network.

CAN Basics

The CAN Bus network is becoming increasingly common, as more and more IC products using this technology become available on the market. The CAN bus was originally designed for automobile applications by automotive system supplier Robert Bosch and is now finding it's way into many industrial and remote control applications. Mainly because of it's reliability and collision detection features.

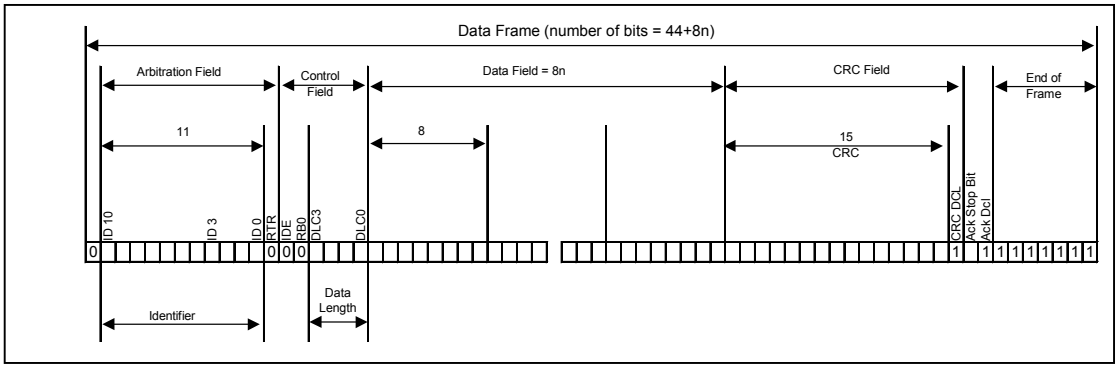
The CAN communication protocol is a CSMA/CD protocol, with the CSMA standing for Carrier Sense Multiple Access, and the CD for Collision Detection. What CSMA means is that every node on the system must monitor the bus for a period of no activity before trying to send a message on the bus. After this period of no activity, if two nodes on the network start transmitting at the same time, the nodes will detect the collision and take the appropriate action.

The CAN protocol is also a message based protocol, so instead of sending messages between nodes, messages are placed on the bus and are available to all nodes. The node that is interested in the message will then take the appropriate action. This is the reason for filter's and masks within the Microchip MCP IC. The masks and filters are set to receive and reject certain messages, based on what the user programs the device for.

The CAN protocol defined four different types of messages. The first and most common which we will discuss in more detail later is the Data frame. This is used for placing information on the bus and can be either a standard data frame or an extended data frame. The difference being the length of the identifier for the node, in the standard data frame the identifier is 11 bits, while in the extended data frame the identifier is 11 + 18 bits. Both types of messages allow up to 8 bytes of data to be transmitted in the one message. The second type of message is the Remote Frame, which is a data frame with the RTR bit set to signify a Remote Transmit Request. The other two frame types are for handling of errors. One is the Overload Frame, for a bus overload and the other the Error Frame for a collision error etc.

The data frame is broken up into a number of fields, these are the Arbitration Field, Control Field, Data Field, CRC Field and End of Frame field, this can be seen in the diagram below. The Arbitration field as well as containing the unit address, also serves to prioritise the messages on the bus. With the lower the number meaning the higher the priority. The RTR bit is also included in the Arbitration field. The control field consists of six bits. This defines if the message is a standard or extended identifier as well as the length of the message.

The Data field is next, with the length of up to 8 bytes, determined by the value in the data length code. Next is the CRC field, which is 16 bits long and is used to check for errors. Finally the Acknowledge and End of Frame section defines the end of the message.



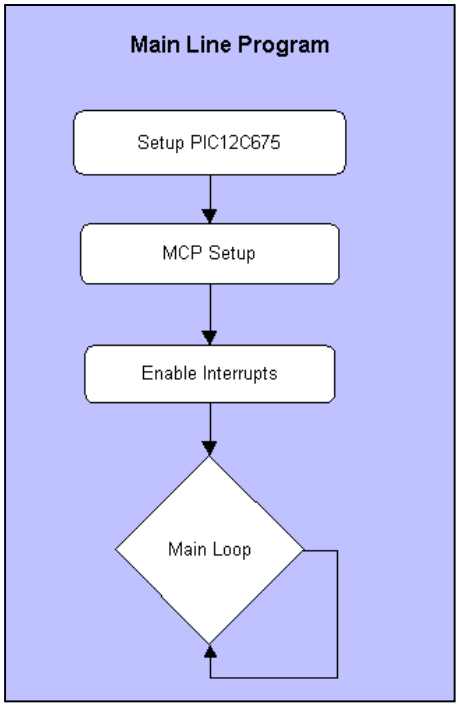
PIC12C675 Program

The program operation is quite simple and a good understanding can be obtained by studying the flow-charts, as well as the code provided on the attached diskette. In this section we will explain the operation in quite a lot of detail. It is not necessary to be able to understand the operation of the program, as the node was designed to operate by sending and receiving various commands. Thus an understanding of the program will only be of benefit if you wish to increase your knowledge of PIC programming and CAN assembly code, and this section can be skipped if this is not important to you.

Firstly the main program includes a number of attached project modules. These modules provide the basic control for the MCP IC and also SPI communication which is used to send and receive commands between the 12C675 and the MCP2510. The MCP modules are broken up into two modules. The first basic module simply sends data bytes to and from the microcontroller and the MCP, here we are using basic bit/banging techniques for the SPI communication. The higher level MCP module includes the set-up procedures, together with the commands for turning on and off various data pins on the MCP. The send command for sending data on the CAN bus is also included in this higher level module. The "comm_new.inc" module, contains the various commands for the node, and their execution, if it was necessary to add new commands, this could be done quite simply by adding the commands within this module.

The final two included modules are for the data eeprom on the PIC12C675, to write and read from this EEPROM. The node address is stored in this EEPROM, such that when the unit is re-addressed the address value is retained once the power is removed. The "ad12c674.inc" is the analogue conversion routine for reading each analogue input channel.

Now to have a look in detail at the main program, we need to also study the flowchart, as seen below.



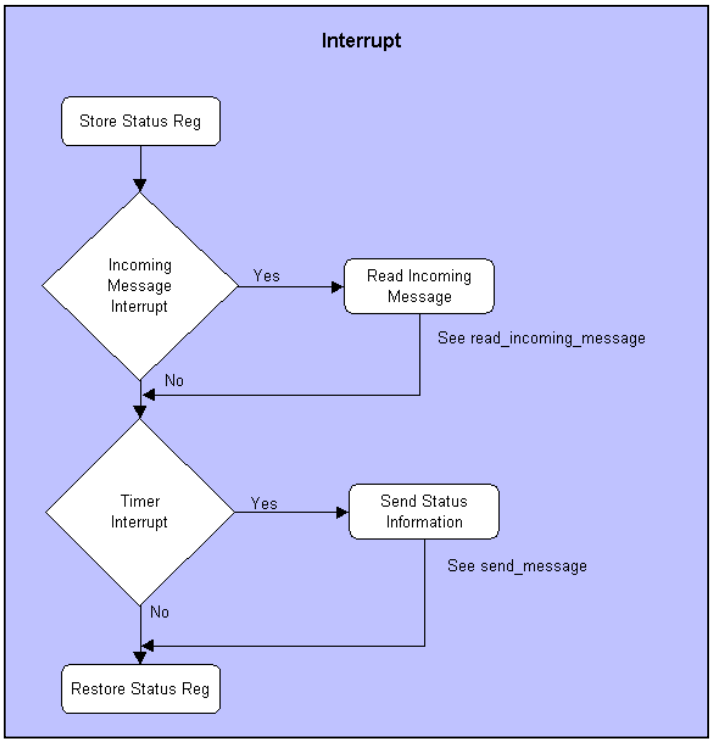
The first section simply sets up the PIC12C675, to enable to various I/O pins and the analogue convertor for use later. The Unit address is also loaded, from the EEPROM.

Next the MCP is made ready for operation. Firstly a reset is carried out to get the MCP ready to receive instructions. Following this a set-up is carried out. During this set-up, the type of CAN bus is specified. This includes such things as the speed, phase segment etc. Then the various I/O pins are enabled for use. The masks and filters are then set, such that only messages with the unit address will be received. Also such that the group commands will be received and placed in a certain receive register. When the set-up is complete, the MCP is placed in normal mode to receive and send messages. There are a number of modes of operation available for the MCP, to learn more about these, please refer to the MCP data sheet.

Once the MCP is ready, the interrupts are then enabled, to wake up the microcontroller should a message be received or the timer overflows. The timer is set to rollover every 62mS, this is further reduced to every 500mS, by using a counter. This will reduce the amount of traffic on the CAN bus, but on the other hand will reduce the response time to a change in the input. This can be changed based on the response time required, however if there are a large number of nodes on the bus care should be taken to avoid data overflows with what may be superfluous data.

Now to look into more detail at the interrupt procedure. This basically identifies the interrupt and sends the program to the appropriate subroutine based on that interrupt. This can be seen below, the flow-chart for this section. If the interrupt is from the MCP, it is signalling there has been a message

received and is requiring this message to be cleared from it's registers. Alternatively a time interrupt will get the microcontroller to check the status of the various ports and send this information onto the CAN bus.

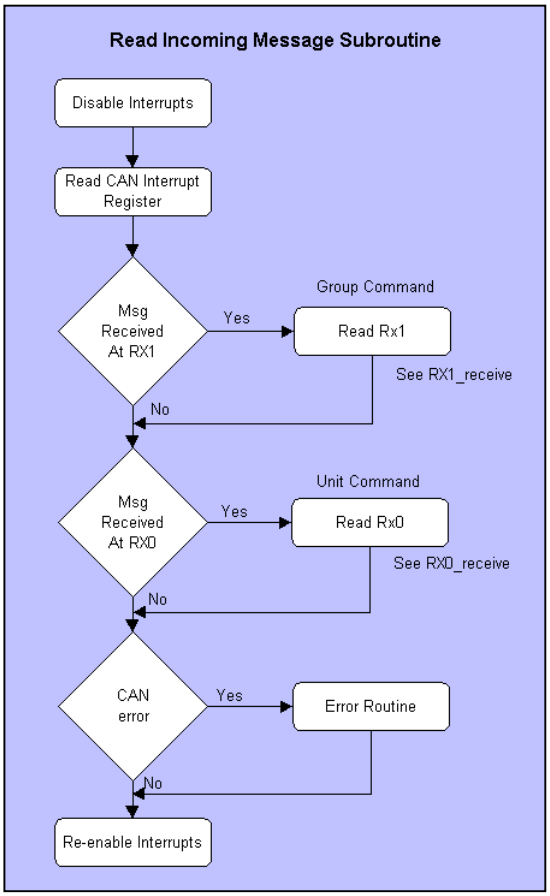


Looking into each of these procedures further, we will look at the read incoming message subroutine first. The basic operation of this procedure can be seen in the flow-chart below.

Firstly, the interrupts are enabled, such that the message can be sent without a fatal interrupt occurring within the routine. The MCP CAN Interrupt register is then read to identify which register the message is at. The group commands are placed in RX1 when received and the specific unit commands are placed in RX0. The error flag is then checked to ensure no error has occurred during receiving the message.

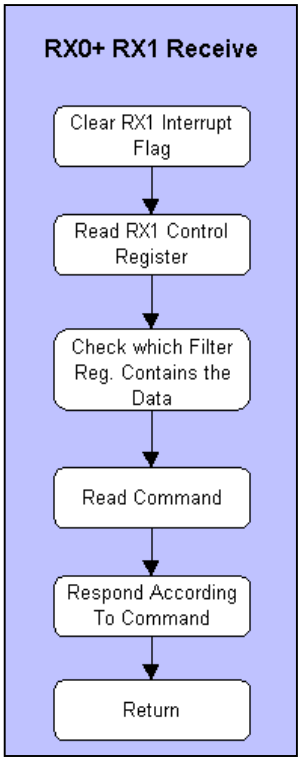
The various receive subroutines will be explained in further detail in a later section. Now to have a look at the send message subroutine.

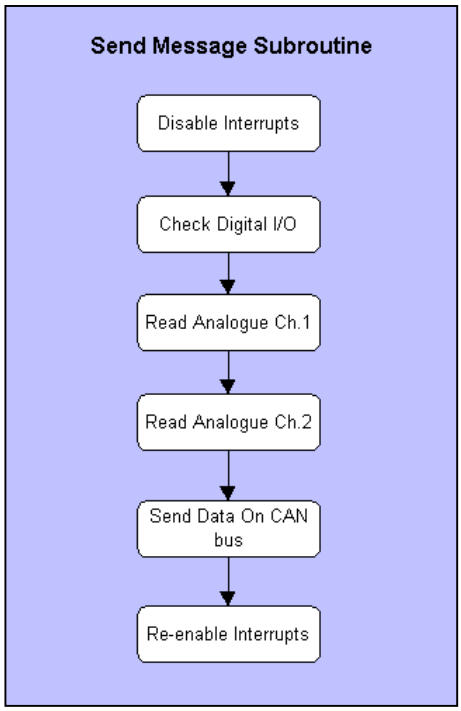
To reduce the number of data messages on the CAN bus, we have added a counter to this procedure such that a status message is only sent every 500mS. This status message, contains the status of the I/O pins and also the value of the 2 analogue channels. To go further into detail about this procedure, we first disable the interrupts as in the previous routine. The digital I/O is then checked and placed in the status register. This value is then sent with the address of the unit. Each analogue channel is then read, using the standard read_analogue routine. These values are stored and sent accordingly. Following completion, the interrupts are re-enabled.



Finally to go further into the RX0 and RX1 receive subroutines. Firstly we need to clear the MCP flag that indicates that a message has been received. We then read the specific control register to determine, which of the filters contains the received data. Which filter receives the message is determined by the various masks we can apply during the set-up operation. Thus different message types can be placed in different filters according to some set criteria. The particular filter is then read and the command and specific data is loaded into the PIC. The command is then checked and action taken according to the command. Only a few

basic commands are included for this Node, in the case of a more complex node, the number of commands can be expanded accordingly.





Circuit Details

The circuit design is rather straightforward. Firstly for the PIC12C675, the internal oscillator has been used, thus these pins are available as I/O pins. The SPI connection has been done using a 2 wire connection in lieu of the standard 3 wire connection. This can be achieved by alternatively switching GP4, between an input and output. The 4.7k resistor is included as a pull up resistor for the input. The analogue connections are direct to the analogue pins on the PIC12C675. Additional protection should be included if these connections are brought in from a remote location. Connections are also provided for in-circuit downloading via a special programmer.

For the MCP2510, this is connected in the standard configuration, with a 8MHz oscillator used to provide timing for the IC. The output is then connected to a standard CAN bus controller. Although the PCA82C251 is used here, a number of equivalent IC's can be used.

Construction

Construction can be completed in 1-2 hours. IC sockets are recommended for the IC's, and these should be installed first. The resistors can then be installed followed by the capacitors and the crystal. Care should be taken with the electrolytic capacitor, that the orientation is correct. The 78L05 should then be installed checking the orientation again. Finally the pcb terminals and header strips.

After the construction is complete, Before installing the IC's, connect the power supply and check the +5V at the various points on the board, including each IC socket. If this is all okay, disconnect the supply again and insert each of the IC's. The Node is now ready to be placed on a CAN bus and set-up as per the following section.

Setting Up and Using The CAN Node

To set-up the CAN Node, it is first necessary to connect it to a standard CAN bus, operating at 125kbit/s. The bus timing should be Q=8, S1=5, S2=3, SJW=1 Although the unit can operate at other settings, this is the recommended values.

Then 'ping' the unit, by issuing a 'f0h' instruction. The unit should then respond with it's current address. If this works okay, then set the unit to the required address by sending the unit address and the '08h' instruction, together with the data byte containing the new address value. This can be from 0 to 63, ensure however that no other unit will have the same address on the bus.

If this is okay, send a turn on command and check the data output at the particular output using a voltmeter. The analogue and digit inputs can also be checked, if a suitable CAN bus monitoring tool is available.

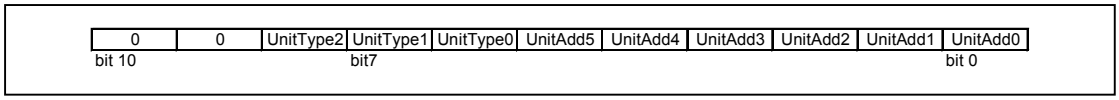
Commands

The group of commands are broken up into two areas. Firstly the specific unit commands, which are used to monitor and change the state of the various output pins. Secondly, the group commands which can be used to shutdown all units, in the case of an emergency. A good understanding of standard CAN message format, would be useful here, and we recommend reading the CAN basic section first.

The unit commands are as shown below,

Unit Commands	
Command	Description
01h	Turn On RXB0
02h	Turn Off RXB0
03h	Turn On RXB1
04h	Turn Off RXB1
05h	Check I/O Status
06h	Read Analogue Ch0
07h	Read Analogue Ch1
08h	Change Unit Address

To send a unit command, we need to first set the identifier for the specific unit we wish to control. The standard CAN identifier is 11 bits long and we have broken these up as follows,



The last 6 bits are for the unit address, thus up to 64 nodes of a certain unit type can be used on the CAN bus. Bit 6 to Bit 8 specify the unit type, in this case the Unit type is 001b, different Unit types can be added to the same bus, these other Unit types can also have their own specific group of instructions. Bit 9 and 10 are unused and should be set to 0.

Thus to send a turn on command to RXB0 at Unit Address 1, we would do the following,

1. Set the CAN identifier to 00+001+000001 (65 decimal)
2. Set the data length to 1
3. Load the command into the first data field
4. Send the message

Now the status of each of the I/O points is contained in the Status register within the PIC. To obtain this status of a particular unit we can send a 05h command similar to as above, or wait until the status information is routinely sent on the bus, this occurs every 500ms. This status will arrive as a data byte contained within a message, first containing the unit identifier, then the command and then the data byte containing the status register information. Thus the data length in this case will be 2, the first byte is the command and the second byte the information or data.

The I/O status is broken down as follows,

Status Register - Command 05

bit 7	Output status of RX1
bit 6	Output status of RX0
bit 5	Unimplemented
bit 4	Unimplemented
bit 3	Unimplemented
bit 2	Input Status of TX2
bit 1	Input Status of TX1
bit 0	Input Status of TX0

This byte can then be broken down to find the status of a particular pin. If for example 01h was received. This would indicate only TX0 is a 1 or ON and all other pins are at 0. A byte value of 82h would indicate RX1 was ON and also the Input of TX1 was high.

Finally to send a group command, it is necessary to send the identifier with just the Unit type. Thus the procedure would be as follows,

1. Set the CAN identifier to 00+001+000000 (64 decimal)
2. Set the data length to 1
3. Load the command into the first data field
4. Send the message

There are two group commands.

Group Commands	
Command	Description
f0h	Identify Unit
f1h	Shutdown

The identify unit command, will cause all the current nodes of the same unit type to respond, that they are on the bus. Thus if two nodes, of unit type 1 were on the bus. Following sending the f0 Identify Unit command, two messages would be returned, with the Unit Identifier and the command value. Thus the data information would appear as follows,

Identifier	Length	Data
64	1	f0
65	1	f0
66	1	f0

This would indicate that Node Address 1 and Node Address 2 were currently on the CAN bus.

For the shutdown command this will simply turn both RXB0 and RXB1 off on every unit of the same type. As previously mentioned, this could be used for an emergency or fire situation, were rapid shutdown is required.

Summary

Once your CAN node is up and running, the fun then begins. The unit can be used in many applications and with a base unit connected to a computer, it opens up many opportunities for control via the computer, over the Internet or through a modem. The digital outputs can also easily be connected to mains control equipment, to turn on Lights, TV etc. The base unit will also allow you to write Visual Basic programs for control and monitoring, of the unit. It is also possible to add control logic within the sensor by modifying the assembly code accordingly. This will reduce the traffic on the CAN bus and allow the unit to operate without a base unit. For example a light sensor could be used to detect a certain level of light and turn on the output lamp, once the light level decreases below a set point. The possibilities are endless, and with an in-expensive node such as this, it is possible to control your home or office at an economical price.