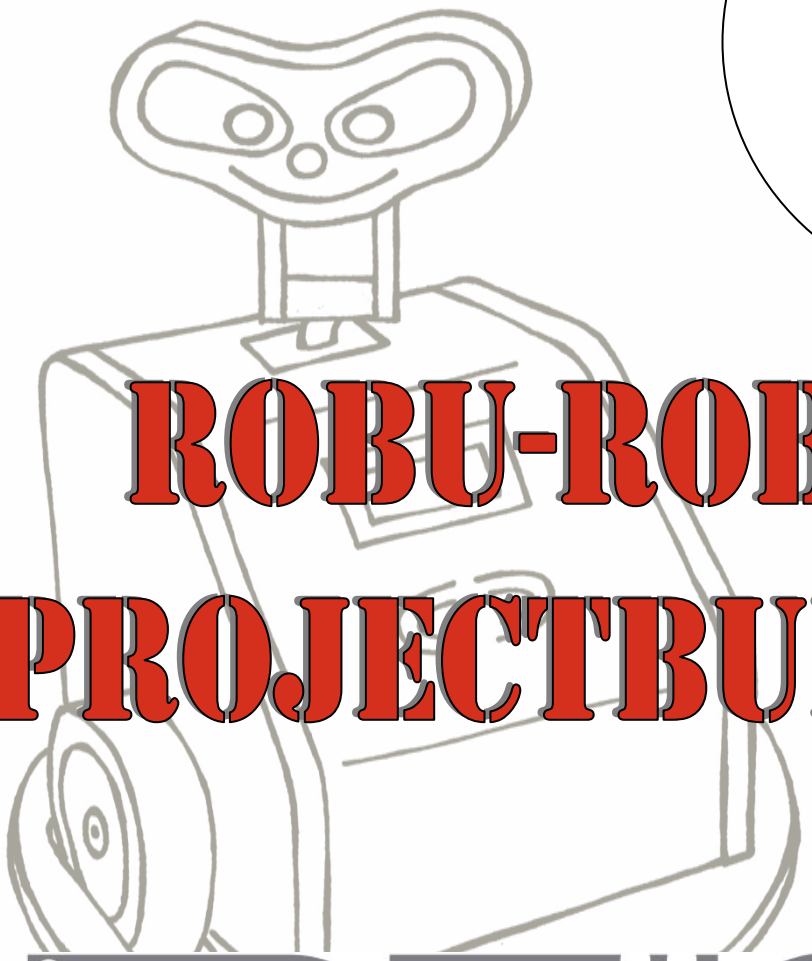
0



ROBU-ROBOT PROJECTBUNDEL

RTC

REGIONAAL TECHNOLOGISCH CENTRUM vzw

Antwerpen

TECHNISCH INSTITUUT St.-Jozef Schoten

techniek werkt!

Projectmap ROBU-ROBOT

1. Inleiding

Het idee achter dit robu-robot project was " We ontwerpen een stevig platform waarin zoveel mogelijk moderne microcontroller technologie – sensoren – actuatoren en bus-sytemen in één systeem wordt samengebracht om studenten 3e graad TSO op een vernieuwende en uitdagende manier kennis te laten maken met deze nieuwe technologieën.

RTC Antwerpen heeft beslist om dit project financieel te ondersteunen en zo konden we in januari 2009 van start gaan met de uitwerking ervan.

Gedurende bijna één jaar is er hard gewerkt en is deze ROBU-ROBOT er als resultaat uit gekomen.

Deze robot omvat:
Actuatoren:

- 2 DC motoren met wheelencoders op de DC motortjes aan de wielen
- Een stappenmotor vooraan
- 2 servo motoren in de nek
- 8 Blauwe leds in mond en 6 voor backlight
- 20 RGB kleuren leds in de ogen
- Een laser in de neus
- Een LCD scherm van 4 x 16 karakters
- Een luidspreker om spraak en muziek te genereren.

Sensoren

- 4 Sharp GP2D12 IR afstandsensoren
- 1 Sharp GP2D120 IR afstandsensor
- 2 LDR licht sensoren
- 1 SRF08 ultrasoon sensor in de ogen
- 1 SRF08 ultrasoon sensor vooraan op de stappenmotor
- CMPS03 kompas sensor
- Wheelencoders op beide wielen

Communicatie

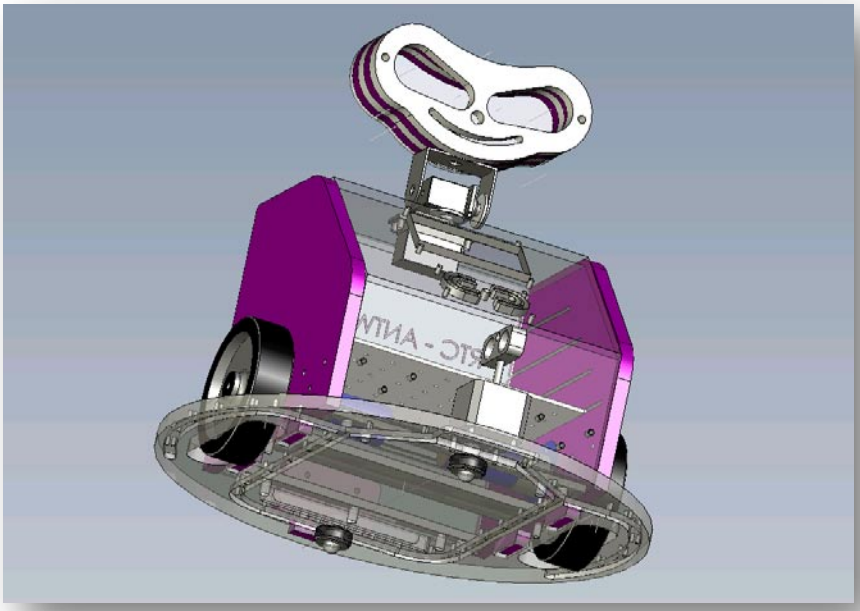
- I2C bus communicatie met de MOTOR driver module
- I2C bus communicatie met de SRF08 sensoren
- I2C bus communicatie met het CMPS03 Compass sensor
- CAN bus communicatie tussen de 3 micro-controllers voor de hoofd-sturing – de body sturing en de muziek sturing.

Microcontrollers

- ECIO 40 pin (PIC) voor sturing body
- ECIO 40 pin (PIC) voor sturing muziek
- ECIO 40 pin (PIC) voor sturing hoofd
- PIC in de twee SRF 08 sensoren
- PIC op de motor driver PCB

Dit volledige project werd gerealiseerd door twee leerkrachten van het St.-Jozefinstituut in Schoten.

Jan Oostvogels is leraar Mechanica en CNC en ontwierp alle mechanische onderdelen. Het idee groeide op de computer en resulteerde na heel veel aanpassingen in een 3D Solidworks tekening.
Op basis van deze tekeningen werden er verschillende prototypes effectief op onze CNC machines geproduceerd, uitgetest en herwerkt waar nodig.
De uiteindelijke productie van de 14 robots is gedaan bij het bedrijf Imatex in Schoten. Jan heeft hierbij gewaakt over de kwaliteit van de productieresultaten.



Bart Huyskens in leraar Elektronica en ICT. Hij ontwierp alle elektronica op basis van E-blocks en andere systemen - testte eerst al deze systemen afzonderlijk uit – tekende 6 verschillende printplaten op maat van dit project en testte deze printplaten in een prototype.

Tijdens de zomermaanden heeft hij alle printplaten gesoldeerd en gemonteerd in de 14 robots.

Daarna volgde het testwerk en het schrijven van de software – samenstellen cursusmateriaal – installeren laptops – in orde maken van de koffers en herwerken van de schema's.

Wij hebben als leerkrachten alvast veel plezier beleefd aan dit project – we hopen dat het voor jullie een even leuke als leerzame ervaring zal zijn.
Bart en Jan

2. Inventaris van één koffer

Omschrijving	Afbeelding	Check
Koffer + deksel		
Robu-Robot		
Laptop + sleeve + adaptor		
Card reader		
Reserve batterij 12V		
USB kabel		
Hoofdtelefoon		
Zaklamp + batterijen		
Projectbundel		
Batterijlader		

3. Algemeen blokschema

Zie schema volgende bladzijde

4. Schema sturing Body

   Zie schema's volgende bladzijden

5. Schema sturing hoofd

Zie schema's volgende bladzijden

6. Schema Muziek

Zie schema's volgende bladzijden

7. Schema Voeding

   Zie schema's volgende bladzijden

8. Schema Hoofd
   Zie schema's volgende bladzijden

9. Schema lijnvolger

   Zie schema's volgende bladzijden

10. Motorsturing MD25

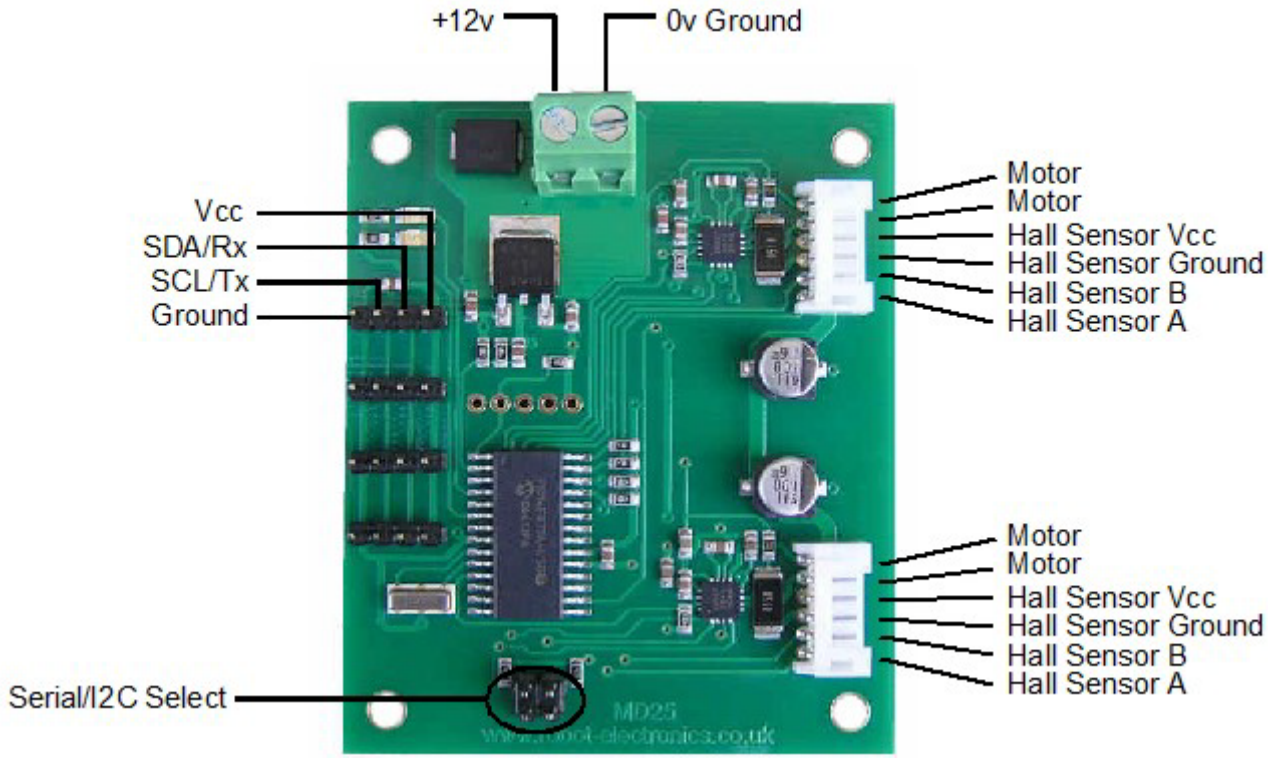
MD25 - Dual 12Volt 2.8Amp H Bridge Motor Drive

Overview

The MD25 is a robust I2C or serial, dual motor driver, designed for use with our EMG30 motors. Main features are:

1. Reads motors encoders and provides counts for determining distance traveled and direction .
2. Drives two motors with independent or combined control.
3. Motor current is readable.
4. Only 12v is required to power the module.
5. Onboard 5v regulator can supply up to 1A peak, 300mA continuously to external circuitry
6. Steering feature, motors can be commanded to turn by sent value.
7. Variable acceleration and power regulation also included

Connections

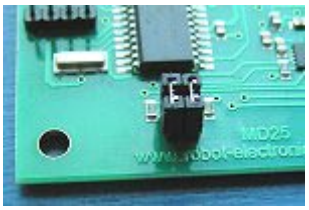


Jumper Selection

I2C mode with no jumpers installed, up to 100 khz clock.
Full Details of I2C Mode is here

Serial mode at 9600 bps, 1 start bit, 2 stop bits, no parity
Full Details of Serial Mode is here

Serial mode at 19200 bps, 1 start bit, 2 stop bits, no parity
Full Details of Serial Mode is here



Serial mode at 38400 bps, 1 start bit, 2 stop bits, no parity
Full Details of Serial Mode is here

Motor Voltage

The MD25 is designed to work with a 12v battery. In practical terms, this means the 9v-14v swing of a flat/charging 12v battery is fine. Much below 9v and the under-voltage protection will prevent any drive to the motors.

Motor Noise Suppression

When using our EMG30 encoded motors, you will find that a 10n noise suppression capacitor has already been fitted. Other motors may require suppression. This is easily achieved by the addition of a 10n snubbing capacitor across the motors. The capacitor should also be capable of handling a voltage of twice the drive voltage to the motor.

Leds

The Red Power Led indicates power is applied to the module.
A Green Led indicates communication activity with the MD25. In I2C mode the green led will also initially flash the address it has been set to. See I2C documentation for further details.

I2C mode documentation

Automatic Speed regulation
By using feedback from the encoders the MD25 is able to dynamically increase power as required. If the required speed is not being achieved, the MD25 will increase power to the motors until it reaches the desired rate or the motors reach there maximum output. Speed regulation can be turned off in the command register.

Automatic Motor Timeout
The MD25 will automatically stop the motors if there is no I2C communications within 2 seconds. This is to prevent your robot running wild if the controller fails. The feature can be turned off, if not required. See the command register.

Controlling the MD25
The MD25 is designed to operate in a standard I2C bus system on addresses from 0xB0 to 0xBE (last bit of address is read/write bit, so even numbers only), with its default address being **0xB2. (In datasheets staat te lezen dat het standard adres 0xB0 is – deze motor drivers hebben in de fabriek standard een verkeerd adres meegekregen nl 0xB2)** This is easily changed by removing the Address Jumper or in the software see Changing the I2C Bus Address.

I2C mode allows the MD25 to be connected to popular controllers such as the PICAXE, OOPic and BS2p, and a wide range of micro-controllers like PIC's, AVR's, 8051's etc.
I2C communication protocol with the MD25 module is the same as popular EPROM's such as the 24C04. To read one or more of the MD25 registers, first send a start bit, the module address (0XB0 for example) with the read/write bit low, then the register number you wish to read. This is followed by a repeated start and the module address again with the read/write bit high (0XB1 in this example). You are now able to read one or more registers. The MD25 has 17 registers numbered 0 to 16 as follows;

Register	Name	Read/Write	Description
0	Speed1	R/W	Motor1 speed (mode 0,1) or speed (mode 2,3)
1	Speed2/Turn	R/W	Motor2 speed (mode 0,1) or turn (mode 2,3)
2	Enc1a	Read only	Encoder 1 position, 1st byte (highest), capture count when read
3	Enc1b	Read only	Encoder 1 position, 2nd byte
4	Enc1c	Read only	Encoder 1 position, 3rd byte
5	Enc1d	Read only	Encoder 1 position, 4th (lowest byte)
6	Enc2a	Read only	Encoder 2 position, 1st byte (highest), capture count when read
7	Enc2b	Read only	Encoder 2 position, 2nd byte
8	Enc2c	Read only	Encoder 2 position, 3rd byte
9	Enc2d	Read only	Encoder 2 position, 4th byte (lowest byte)
10	Battery volts	Read only	The supply battery voltage

11	Motor 1 current	Read only	The current through motor 1
12	Motor 2 current	Read only	The current through motor 2
13	Software Revision	Read only	Software Revision Number
14	Acceleration rate	R/W	Optional Acceleration register
15	Mode	R/W	Mode of operation (see below)
16	Command	R/W	Used for reset of encoder counts and module address changes

Speed1 Register
Depending on what mode you are in, this register can affect the speed of one motor or both motors. If you are in mode 0 or 1 it will set the speed and direction of motor 1. The larger the number written to this register, the more power is applied to the motor. A mode of 2 or 3 will control the speed and direction of both motors (subject to effect of turn register).

Speed2/Turn Register
When in mode 0 or 1 this register operates the speed and direction of motor 2. When in mode 2 or 3 Speed2 becomes a Turn register, and any value in this register is combined with the contents of Speed1 to steer the device (see below).

Turn mode
Turn mode looks at the speed register to decide if the direction is forward or reverse. Then it applies a subtraction or addition of the turn value on either motor.

so if the direction is forward
motor speed1 = speed - turn
motor speed2 = speed + turn

else the direction is reverse so
motor speed1 = speed + turn
motor speed2 = speed - turn

If the either motor is not able to achieve the required speed for the turn (beyond the maximum output), then the other motor is automatically changed by the program to meet the required difference.

Encoder registers
Each motor has its encoder count stored in an array of four bytes, together the bytes form a signed 32 bit number, the encoder count is captured on a read of the highest byte (registers 2, 6) and the subsequent lower bytes will be held until another read of the highest byte takes place. The count is stored with the highest byte in the lowest numbered register. The registers can be zeroed at any time by writing 32 (0x20) to the command register.

Battery volts
A reading of the voltage of the connected battery is available in this register. It reads as 10 times the voltage (121 for 12.1v).

Motor 1 and 2 current
A guide reading of the average current through the motor is available in this register. It reads approx ten times the number of Amps (25 at 2.5A).

Software Revision number
This register contains the revision number of the software in the modules PIC16F873 controller - currently 1 at the time of writing.

Acceleration Rate
If you require a controlled acceleration period for the attached motors to reach there ultimate speed, the MD25 has a register to provide this. It works by using a value into the acceleration register and incrementing the power by that value. Changing between the current speed of the motors and the new speed (from speed 1 and 2 registers). So if the motors were traveling at full speed in the forward direction (255) and were instructed to move at full speed in reverse (0), there would be 255 steps with an acceleration register value of 1, but 128 for a value of 2. The default acceleration value is 5, meaning the speed is changed from full forward to full reverse in 1.25 seconds. The register will accept values of 1 up to 10 which equates to a period of only 0.65 seconds to travel from full speed in one direction to full speed in the opposite direction.

So to calculate the time (in seconds) for the acceleration to complete :

if new speed > current speed
steps = (new speed - current speed) / acceleration register

if new speed < current speed
steps = (current speed - new speed) / acceleration register

time = steps * 25ms

For example :

Acceleration register	Time/step	Current speed	New speed	Steps	Acceleration time
1	25ms	0	255	255	6.375s
2	25ms	127	255	64	1.6s
3	25ms	80	0	27	0.675s
5 (default)	25ms	0	255	51	1.275s
10	25ms	255	0	26	0.65s

Mode Register
The mode register selects which mode of operation and I2C data input type the user requires. The options being:

0, (Default Setting) If a value of 0 is written to the mode register then the meaning of the speed registers is literal speeds in the range of 0 (Full Reverse) 128 (Stop) 255 (Full Forward). **Deze mode bij voorkeur gebruiken.**

1, Mode 1 is similar to Mode 0, except that the speed registers are interpreted as signed values. The meaning of the speed registers is literal speeds in the range of -128 (Full Reverse) 0 (Stop) 127 (Full Forward).

2, Writing a value of 2 to the mode register will make speed1 control both motors speed, and speed2 becomes the turn value.
Data is in the range of 0 (Full Reverse) 128 (Stop) 255 (Full Forward).

3, Mode 3 is similar to Mode 2, except that the speed registers are interpreted as signed values.
Data is in the range of -128 (Full Reverse) 0 (Stop) 127 (Full Forward)

Command register

Command		Action
Dec	Hex	
32	20	Resets the encoder registers to zero
48	30	Disables automatic speed regulation
49	31	Enables automatic speed regulation (default)
50	32	Disables 2 second timeout of motors (Version 2 onwards only)
51	33	Enables 2 second timeout of motors when no I2C comms (default) (Version 2 onwards only)
160	A0	1st in sequence to change I2C address
170	AA	2nd in sequence to change I2C address
165	A5	3rd in sequence to change I2C address

Changing the I2C Bus Address

To change the I2C address of the MD25 by writing a new address you must have only one module on the bus. Write the 3 sequence commands in the correct order followed by the address. Example; to change the address of an MD25 currently at 0xB0 (the default shipped address) to 0xB4, write the following to address 0xB0; (0xA0, 0xAA, 0xA5, 0xB4). These commands must be sent in the correct sequence to change the I2C address, additionally, no other command may be issued in the middle of the sequence. The sequence must be sent to the command register at location 16, which means 4 separate write transactions on the I2C bus. Because of the way the MD25 works internally, there MUST be a delay of at least 5mS between the writing of each of these 4 transactions. When done, you should label the MD25 with its address, however if you do forget, just power it up without sending any commands. The MD25 will flash its address out on the green communication LED. One long flash followed by a number of shorter flashes indicating its address. Any command sent to the MD25 during this period will still be received and writing new speeds or a write to the command register will terminate the flashing.

Address		Long Flash	Short Flashes
Decimal	Hex		
176	B0	1	0
178	B2	1	1
180	B4	1	2
182	B6	1	3
184	B8	1	4
186	BA	1	5
188	BC	1	6
190	BE	1	7

Take care not to set more than one MD25 to the same address, there will be a bus collision and very unpredictable results.

EGM30 Motor met Wheelencoder

Connector
The EGM30 is supplied with a 6 way JST connector (part no PHR-6) at the end of approx 90mm of cable as standard. The connections are:

Wire colour Connection

Purple (1) Hall Sensor B Vout
Blue (2) Hall sensor A Vout
Green (3) Hall sensor ground
Brown (4) Hall sensor Vcc
Red (5) + Motor
Black (6) - Motor

note that pull up resistors (we used 4k7) are required on the hall sensor outputs, and wires are the colours from the actual cable.

specification

Rated voltage 12v
Rated torque 1.5kg/cm
Rated speed 170rpm
Rated current 530mA
No load speed 216
No load current 150mA
EMG30 Mounting bracket
Stall Current 2.5A
Rated output 4.22W
Encoder counts per output shaft turn 360

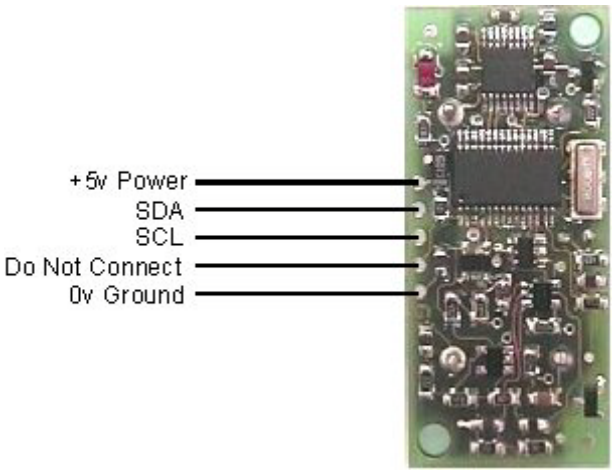
Measured Shaft Speed when used off-load with MD23 and 12v supply.

Minimum Speed 1.5rpm
Maximum Speed 200rpm

11. SRF08 Ultrasoon afstandmeting

Communication with the SRF08 ultrasonic rangefinder is via the I2C bus. This is available on popular controllers such as the OOPic and Stamp BS2p, as well as a wide variety of micro-controllers. To the programmer the SRF08 behaves in the same way as the ubiquitous 24xx series eeprom's, except that the I2C address is different. The default shipped address of the SRF08 is 0xE0. It can be changed by the user to any of 16 addresses E0, E2, E4, E6, E8, EA, EC, EE, F0, F2, F4, F6, F8, FA, FC or FE, therefore up to 16 sonar's can be used. In addition to the above addresses, all sonar's on the I2C bus will respond to address 0 - the General Broadcast address. This means that writing a ranging command to I2C address 0 (0x00) will start all sonar's ranging at the same time. This should be useful in ANN Mode (See below). The results must be read individually from each sonar's real address. We have examples of using the SRF08 module with a wide range of popular controllers.

Connections
The "Do Not Connect" pin should be left unconnected. It is actually the CPU MCLR line and is used once only in our workshop to program the PIC16F872 on-board after assembly, and has an internal pull-up resistor. The SCL and SDA lines should each have a pull-up resistor to +5v somewhere on the I2C bus. You only need one pair of resistors, not a pair for every module. They are normally located with the bus master rather than the slaves. The SRF08 is always a slave - never a bus master. If you need them, I recommend 1.8k resistors. Some modules such as the OOPic already have pull-up resistors and you do not need to add any more.



Registers

The SRF08 appears as a set of 36 registers.

Location	Read	Write
0	Software Revision	Command Register
1	Light Sensor	Max Gain Register (default 31)
2	1st Echo High Byte	Range Register (default 255)
3	1st Echo Low Byte	N/A
~~	~~~~	~~~~
34	17th Echo High Byte	N/A
35	17th Echo Low Byte	N/A

Only locations 0, 1 and 2 can be written to. Location 0 is the command register and is used to start a ranging session. It cannot be read. Reading from location 0 returns the SRF08 software revision. By default, the ranging lasts for 65mS, but can be changed by writing to the range register at location 2. If you do so, then you will likely need to change the analogue gain by writing to location 1. See the **Changing Range** and **Analogue Gain** sections below.

Location 1 is the onboard light sensor. This data is updated every time a new ranging command has completed and can be read when range data is read. The next two locations, 2 and 3, are the 16bit unsigned result from the latest ranging - high byte first. The meaning of this value depends on the command used, and is either the range in inches, or the range in cm or the flight time in uS. A value of zero indicates that no objects were detected. There are up to a further 16 results indicating echo's from more distant objects.

Commands

The are three commands to initiate a ranging (80 to 82), to return the result in inches, centimeters or microseconds. There is also an ANN mode (Artificial Neural Network) mode which is described later and a set of commands to change the I2C address.

Command		Action
Decimal	Hex	
80	0x50	Ranging Mode - Result in inches
81	0x51	Ranging Mode - Result in centimeters
82	0x52	Ranging Mode - Result in micro-seconds
83	0x53	ANN Mode - Result in inches
84	0x54	ANN Mode - Result in centimeters
85	0x55	ANN Mode - Result in micro-seconds
160	0xA0	1st in sequence to change I2C address
165	0xA5	3rd in sequence to change I2C address
170	0xAA	2nd in sequence to change I2C address

Ranging Mode
To initiate a ranging, write one of the above commands to the command register and wait the required amount of time for completion and read as many results as you wish. The echo buffer is cleared at the start of each ranging. The first echo range is placed in locations 2,3. the second in 4,5, etc. If a location (high and low bytes) is 0, then there will be no further reading in the rest of the registers. The default and recommended time for completion of ranging is 65mS, however you can shorten this by writing to the range register before issuing a ranging command. Light sensor data at location 1 will also have been updated after a ranging command.

ANN Mode
ANN mode (Artificial Neural Network) is designed to provide the multi echo data in a way that is easier to input to a neural network, at least I hope it is - I've not actually done it yet. ANN mode provides a 32 byte buffer (locations 4 to 35 inclusive) where each byte represents the 65536uS maximum flight time divided into 32 chunks of 2048uS each - equivalent to about 352mm of range. If an echo is received within a bytes time slot then it will be set to no-zero, otherwise it will be zero. So if an echo is received from within the first 352mm, location 4 will be non-zero. If an object is detected 3m away the location 12 will be non-zero (3000/352 = 8) (8+4=12). Arranging the data like this should be better for a neural net than the other formats. The input to your network should be 0 if the byte is zero and 1 if its non-zero. I have a SOFM (Self Organizing Feature Map) in mind for the neural net, but will hopefully be useful for any type.

Location 4	Location 5	Location 6	Location 7	Locations 8 - 35
0 - 352mm	353 - 705mm	706 - 1057mm	1058 - 1410mm	and so on

Locations 2,3 contain the range of the nearest object converted to inches, cm or uS and is the same as for Ranging Mode.

Checking for Completion of Ranging
You do not have to use a timer on your own controller to wait for ranging to finish. You can take advantage of the fact that the SRF08 will not respond to any I2C activity whilst ranging. Therefore, if you try to read from the SRF08 (we use the software revision number a location 0) then you will get 255 (0xFF) whilst ranging. This is because the I2C data line (SDA) is pulled high if nothing is driving it. As soon as the ranging is complete the SRF08 will again respond to the I2C bus, so just keep reading the register until its not 255 (0xFF) anymore. You can then read the sonar data. Your controller can take advantage of this to perform other tasks while the SRF08 is ranging.

Changing the Range
The maximum range of the SRF08 is set by an internal timer. By default, this is 65mS or the equivalent of 11 metres of range. This is much further than the 6 metres the SRF08 is actually capable of. It is possible to reduce the time the SRF08 listens for an echo, and hence the range, by writing to the range register at location 2. The range can be set in steps of about 43mm (0.043m or 1.68 inches) up to 11 metres.
The range is ((Range Register x 43mm) + 43mm) so setting the Range Register to 0 (0x00) gives a maximum range of 43mm. Setting the Range Register to 1 (0x01) gives a maximum

range of 86mm. More usefully, 24 (0x18) gives a range of 1 metre and 140 (0x8C) is 6 metres. Setting 255 (0xFF) gives the original 11 metres (255 x 43 + 43 is 11008mm). There are two reasons you may wish to reduce the range.
1. To get at the range information quicker
2. To be able to fire the SRF08 at a faster rate.
If you only wish to get at the range information a bit sooner and will continue to fire the SRF08 at 65ms of slower, then all will be well. However if you wish to fire the SRF08 at a faster rate than 65mS, you will definitely need to reduce the gain - see next section.
The range is set to maximum every time the SRF08 is powered-up. If you need a different range, change it once as part of your system initialization code.

Analogue Gain
The analogue gain register sets the *Maximum* gain of the analogue stages. To set the maximum gain, just write one of these values to the gain register at location 1. During a ranging, the analogue gain starts off at its minimum value of 94. This is increased at approx. 70uS intervals up to the maximum gain setting, set by register 1. Maximum possible gain is reached after about 390mm of range. The purpose of providing a limit to the maximum gain is to allow you to fire the sonar more rapidly than 65mS. Since the ranging can be very short, a new ranging can be initiated as soon as the previous range data has been read. A potential hazard with this is that the second ranging may pick up a distant echo returning from the previous "ping", give a false result of a close by object when there is none. To reduce this possibility, the maximum gain can be reduced to limit the modules sensitivity to the weaker distant echo, whilst still able to detect close by objects. The maximum gain setting is stored only in the CPU's RAM and is initialized to maximum on power-up, so if you only want do a ranging every 65mS, or longer, you can ignore the Range and Gain Registers.
Note - Effective in Ranging Mode only, in ANN mode, gain is controlled automatically.

Gain Register		Maximum Analogue Gain
Decimal	Hex	
0	0x00	Set Maximum Analogue Gain to 94
1	0x01	Set Maximum Analogue Gain to 97
2	0x02	Set Maximum Analogue Gain to 100
3	0x03	Set Maximum Analogue Gain to 103
4	0x04	Set Maximum Analogue Gain to 107
5	0x05	Set Maximum Analogue Gain to 110
6	0x06	Set Maximum Analogue Gain to 114
7	0x07	Set Maximum Analogue Gain to 118
8	0x08	Set Maximum Analogue Gain to 123
9	0x09	Set Maximum Analogue Gain to 128
10	0x0A	Set Maximum Analogue Gain to 133
11	0x0B	Set Maximum Analogue Gain to 139
12	0x0C	Set Maximum Analogue Gain to 145
13	0x0D	Set Maximum Analogue Gain to 152
14	0x0E	Set Maximum Analogue Gain to 159
15	0x0F	Set Maximum Analogue Gain to 168

16	0x10	Set Maximum Analogue Gain to 177
17	0x11	Set Maximum Analogue Gain to 187
18	0x12	Set Maximum Analogue Gain to 199
19	0x13	Set Maximum Analogue Gain to 212
20	0x14	Set Maximum Analogue Gain to 227
21	0x15	Set Maximum Analogue Gain to 245
22	0x16	Set Maximum Analogue Gain to 265
23	0x17	Set Maximum Analogue Gain to 288
24	0x18	Set Maximum Analogue Gain to 317
25	0x19	Set Maximum Analogue Gain to 352
26	0x1A	Set Maximum Analogue Gain to 395
27	0x1B	Set Maximum Analogue Gain to 450
28	0x1C	Set Maximum Analogue Gain to 524
29	0x1D	Set Maximum Analogue Gain to 626
30	0x1E	Set Maximum Analogue Gain to 777
31	0x1F	Set Maximum Analogue Gain to 1025

Note that the relationship between the Gain Register setting and the actual gain is not a linear one. Also there is no magic formula to say "use this gain setting with that range setting". It depends on the size, shape and material of the object and what else is around in the room. Try playing with different settings until you get the result you want. If you appear to get false readings, it may be echo's from previous "pings", try going back to firing the SRF08 every 65mS or longer (slower).
If you are in any doubt about the Range and Gain Registers, remember they are automatically set by the SRF08 to their default values when it is powered-up. You can ignore and forget about them and the SRF08 will work fine, detecting objects up to 6 metres away every 65mS or slower.

Light Sensor
The SRF08 has a light sensor on-board. A reading of the light intensity is made by the SRF08 each time a ranging takes place in either Ranging or ANN Modes (The A/D conversion is actually done just before the "ping" whilst the +/- 10v generator is stabilizing). The reading increases as the brightness increases, so you will get a maximum value in bright light and minimum value in darkness. It should get close to 2-3 in complete darkness and up to about 248 (0xF8) in bright light. The light intensity can be read from the Light Sensor Register at location 1 at the same time that you are reading the range data.

LED
The red LED is used to flash out a code for the I2C address on power-up (see below). It also gives a brief flash during the "ping" whilst ranging.

Changing the I2C Bus Address

To change the I2C address of the SRF08 you must have only one sonar on the bus. Write the 3 sequence commands in the correct order followed by the address. Example; to change the address of a sonar currently at 0xE0 (the default shipped address) to 0xF2, write the following to address 0xE0; (0xA0, 0xAA, 0xA5, 0xF2). These commands must be sent in the correct sequence to change the I2C address, additionally, No other command may be issued in the middle of the sequence. The sequence must be sent to the command register at location 0, which means 4 separate write transactions on the I2C bus. When done, you should label the sonar with its address, however if you do forget, just power it up without sending any commands. The SRF08 will flash its address out on the LED. One long flash followed by a number of shorter flashes indicating its address. The flashing is terminated immediately on sending a command the SRF08.

Address		Long Flash	Short flashes
Decimal	Hex		
224	E0	1	0
226	E2	1	1
228	E4	1	2
230	E6	1	3
232	E8	1	4
234	EA	1	5
236	EC	1	6
238	EE	1	7
240	F0	1	8
242	F2	1	9
244	F4	1	10
246	F6	1	11
248	F8	1	12
250	FA	1	13
252	FC	1	14
254	FE	1	15

Take care not to set more than one sonar to the same address, there will be a bus collision and very unpredictable results.

Current Consumption

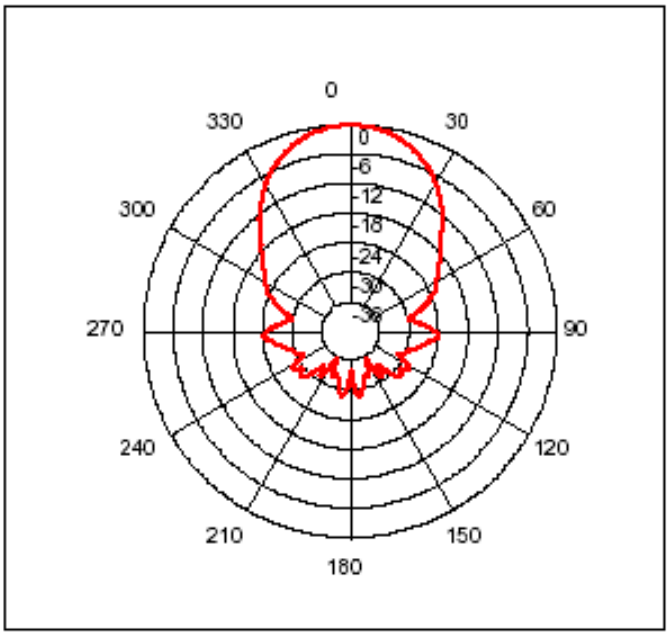
Average current consumption measured on our prototype is around 12mA during ranging, and 3mA standby. The module will automatically go to standby mode after a ranging, whilst waiting for a new command on the I2C bus. The actual measured current profile is as follows;

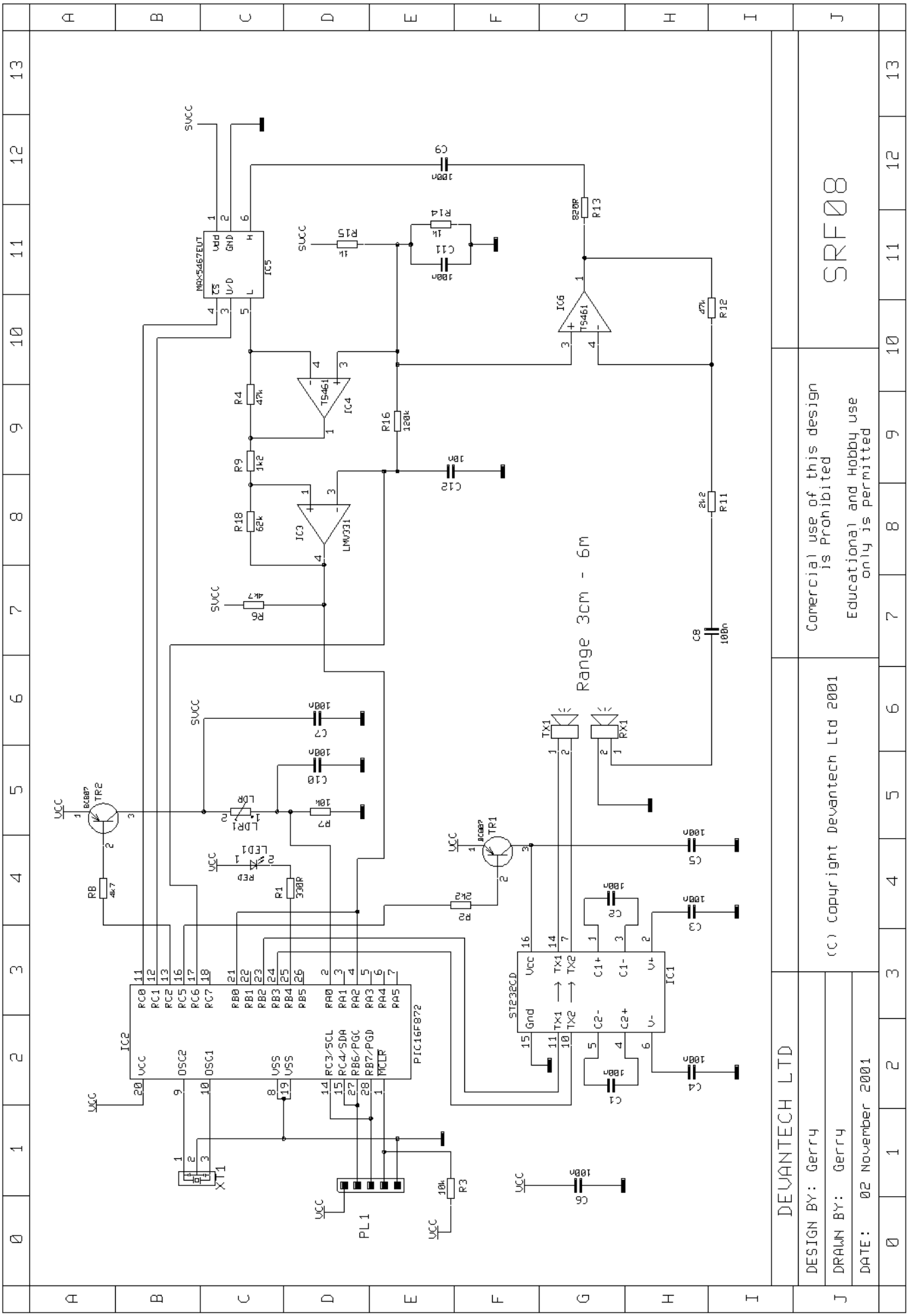
Operation	Current	Duration
Ranging command received - Power on	275mA	3uS
+/- 10v generator Stabilization	25mA	600uS
8 cycles of 40kHz "ping"	40mA	200uS
Ranging	11mA	65mS max
Standby	3mA	indefinite

The above values are for guidance only, they are not tested on production units.

Changing beam pattern and beam width

You can't! This is a question which crops up regularly, however there is no easy way to reduce or change the beam width that I'm aware of. The beam pattern of the SRF08 is conical with the width of the beam being a function of the surface area of the transducers and is fixed. The beam pattern of the transducers used on the SRF08, taken from the manufacturers data sheet, is shown below.



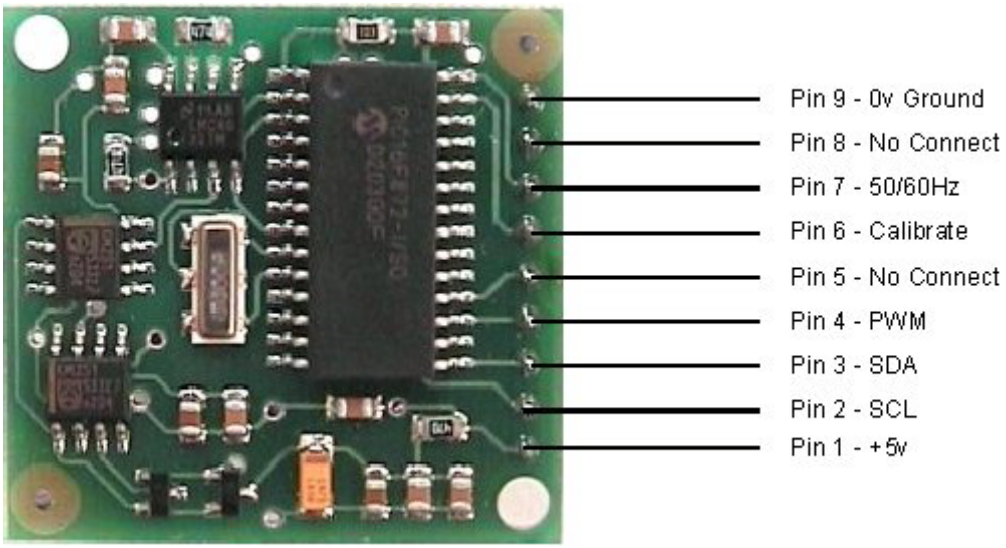

SRF08
DEVANTECH LTD
DESIGN BY: Gerry
DRAWN BY: Gerry
DATE: 02 November 2001
(c) Copyright Devantech Ltd 2001
Comercial use of this design is Prohibited
Educational and Hobby use only is permitted
Range 3cm - 6m
SVCC
VCC
MAX5467EUT
TS461
LMV331
PIC16F872
ST232CD
BC847
LDR
LED
XTAL
PL1
TX1
RX1

12. CMP03 Kompas sensor

Features and Benefits

- Voltage - 5v only required
- Current - 20mA Typ.
- Resolution - 0.1 Degree
- Accuracy - 3-4 degrees approx. after calibration
- Output 1 - Timing Pulse 1mS to 37mS in 0.1mS increments
- Output 2 - I2C Interface, 0-255 and 0-3599 SCL speed up to 1MHz
- Small Size - 32mm x 35mm
- Low Cost - Best Price Compass Module Available

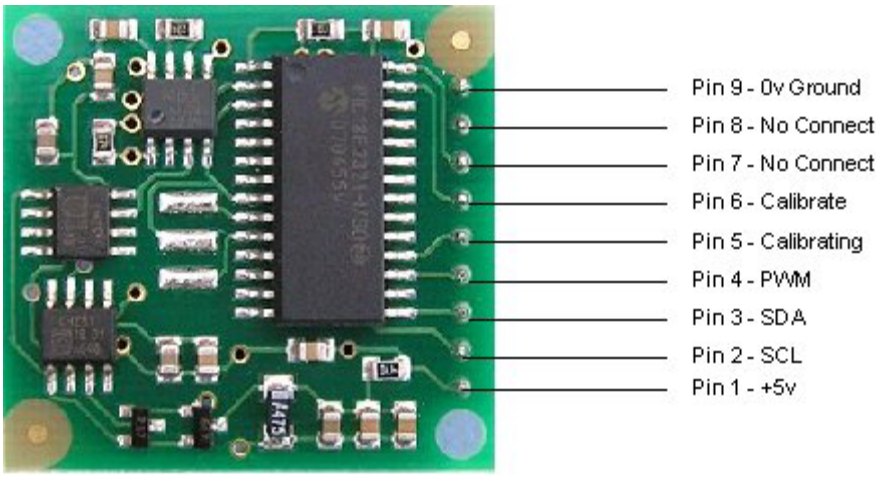
Compass Pinout



Earlier versions can be identified by the presence of the silver 8MHz ceramic resonator in the middle of the PCB, this has been removed on new modules.
Rev14 was released March 2007

Overview
This compass module has been specifically designed for use in robots as an aid to navigation. The aim was to produce a unique number to represent the direction the robot is facing. The compass uses the Philips KMZ51 magnetic field sensor, which is sensitive enough to detect the Earths magnetic field. The output from two of them mounted at right angles to each other is used to compute the direction of the horizontal component of the Earths magnetic field. We have examples of using the Compass module with a wide range of popular controllers.

Connections to the compass module



Connections
Pin 1, +5v. The compass module requires a 5v power supply at a nominal 25mA.
There are two ways of getting the bearing from the module. A PWM signal is available on pin 4, or an I2C interface is provided on pins 2,3.

Pins 2,3 are the I2C interface and can be used to get a direct readout of the bearing. If the I2C interface is not used then these pins should be pulled high (to +5v) via a couple of resistors. Around 47k is ok, the values are not at all critical.

Pin 4. The PWM signal is a pulse width modulated signal with the positive width of the pulse representing the angle. The pulse width varies from 1mS (0°) to 36.99mS (359.9°) – in other words 100uS/° with a +1mS offset. The signal goes low for 65mS between pulses, so the cycle time is 65mS + the pulse width - ie. 66ms-102ms. The pulse is generated by a 16 bit timer in the processor giving a 1uS resolution, however I would not recommend measuring this to anything better than 0.1° (10uS). Make sure you connect the I2C pins, SCL and SDA, to the 5v supply if you are using the PWM, as there are no pull-up resistors on these pins.

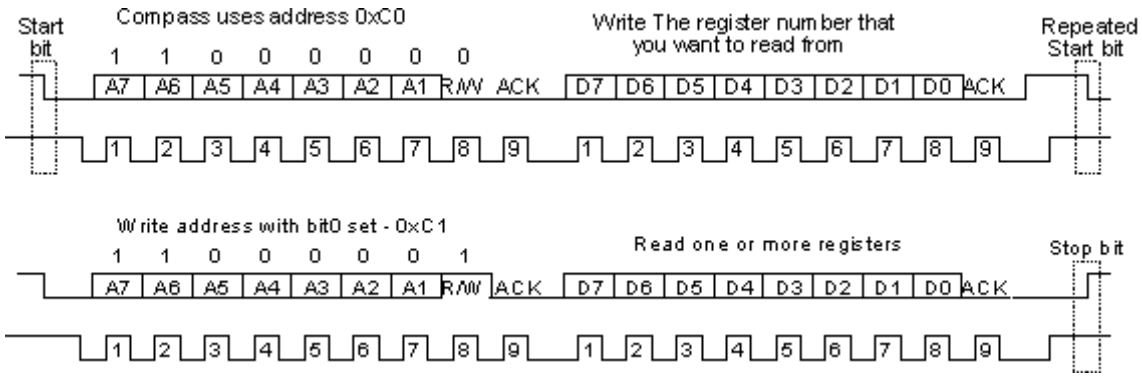
Pin 5 is used to indicate calibration is in progress (active low). You can connect an LED from this pin to +5v via a 390 ohm resistor if you wish.

Pin 6 is one of two ways to calibrate the compass, the other is writing 255 (0xFF) to the command register. Full calibration instructions are further down this page. The calibrate input has an on-board pull-up resistor and can be left unconnected after calibration.

Pins 7 and 8 are currently unused. They have on-board pull-up resistors and should be left unconnected.

Pin 9 is the 0v power supply.

I2C interface.
I2C communication protocol with the compass module is the same as popular eeprom's such as the 24C04.



First send a start bit, the module address (0XC0) with the read/write bit low, then the register number you wish to read. This is followed by a repeated start and the module address again with the read/write bit high (0XC1). You now read one or two bytes for 8bit or 16bit registers respectively. 16bit registers are read high byte first. The compass has a 16 byte array of registers, some of which double up as 16 bit registers as follows;

Register	Function
0	Software Revision Number, Rev14 or higher - for earlier Revisions click here
1	Compass Bearing as a byte, i.e. 0-255 for a full circle
2,3	Compass Bearing as a word, i.e. 0-3599 for a full circle, representing 0-359.9 degrees.
4,5	Internal Test - Sensor1 processed difference signal - 16 bit signed word
6,7	Internal Test - Sensor2 processed difference signal - 16 bit signed word
8,9	Internal Test - Sensor1 raw data - 16 bit signed word
10,11	Internal Test - Sensor2 raw data - 16 bit signed word
12	Unlock code1 - Unlock codes are required for I2C address change or restoring factory calibration
13	Unlock code2
14	Unlock code3
15	Command Register - See text.

Register 0 is the Software revision number (14 at the time of writing). Register 1 is the bearing converted to a 0-255 value. This may be easier for some applications than 0-360 which requires two bytes. For those who require better resolution registers 2 and 3 (high byte first) are a 16 bit unsigned integer in the range 0-3599. This represents 0-359.9°. Registers 4 to 11 are internal test registers. Registers 8,9 and 10,11 contain the raw sensor data. These are the signals coming directly from the sensors, and are the starting point for all the internal calculations which produce the compass bearing. Registers 12,13 and 14 are for writing the unlock codes for I2C address change or restoring factory calibration. Register 15 is the command Register.

The I2C interface does not have any pull-up resistors on the board, these should be provided elsewhere, most probably with the bus master. They are required on both the SCL and SDA lines, but only once for the whole bus, not on each module. I suggest a value of 1k8 if you are going to be working up to 400KHz and 1k2 or even 1k if you are going up to 1MHz. The compass is designed to work at up to the standard clock speed (SCL) of 100KHz, however the

clock speed can be raised to 1MHZ providing the following precaution is taken;
At speeds above around 160KHz the CPU cannot respond fast enough to read the I2C data. Therefore a small delay of 50uS should be inserted either side of writing the register address. No delays are required anywhere else in the sequence. By doing this, I have tested the compass module up to 1.3MHz SCL clock speed. There is an example driver here using the HITECH PICC compiler for the PIC16F877. Note that the above is of no concern if you are using popular embedded language processors such as the OOPic. The compass module always operates as a slave, its never a bus master.

Command Register
Register 15 is the command Register. There are very few commands - 0xC0-CE for I2c address change and 0xF2 for restoring factory calibration - these require unlock codes, see below. Also 255 (0xFF) is the calibrate command. There are no unlock codes required for this.

Changing the I2C address from factory default of 0xC0
With Rev14 onwards, it is now possible to change the I2C address to any of 8 addresses 0xC0, 0xC2, 0xC4, 0xC6, 0xC8, 0xCA, 0xCC or 0xCE. You do this by writing unlock codes to registers 12,13 and 14 and the new address to register 15. Note that the unlock codes are different to the ones which restore factory calibration.

Reg 12	Reg 13	Reg 14	Reg 15
0xA0	0xAA	0xA5	0xC2

The above example will change the address to 0xC2 and the new address will be effective immediately. Don't forget to label you CMPS03 with the new address. You can do this in one I2C transaction, setting the register address to 12 and writing the four bytes. The internal register pointer is incremented automatically.

Restoring Factory Calibration
With Rev14 onwards, it is now possible to restore the factory calibration settings. You do this by writing unlock codes to registers 12,13 and 14 and the restore command (0xF2) to register 15. Note that the unlock codes are different to the ones which used for changing the I2C address.

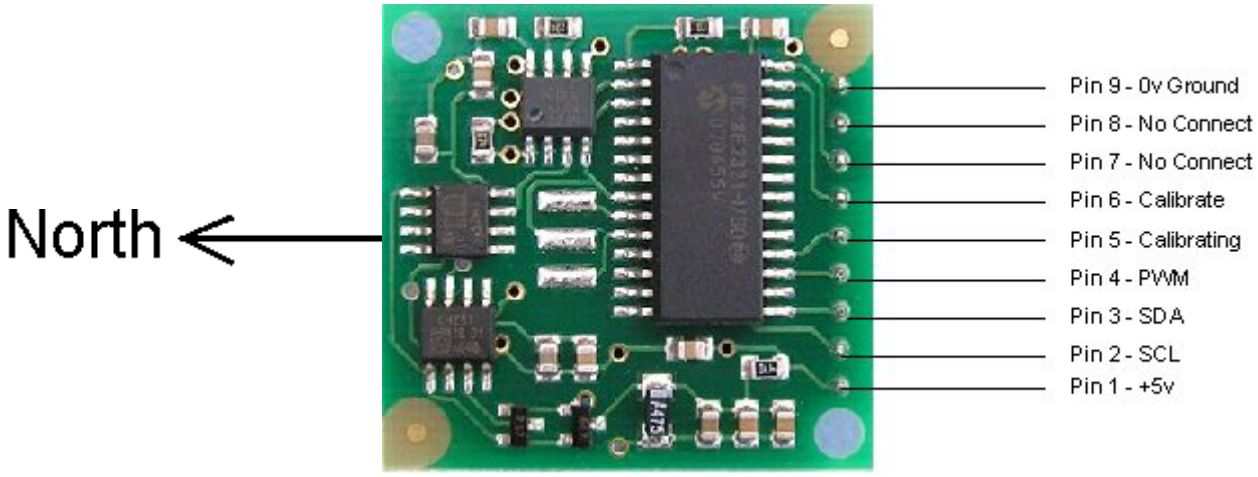
Reg 12	Reg 13	Reg 14	Reg 15
0x55	0x5A	0xA5	0xF2

You can do this in one transaction, setting the register address to 12 and writing the four bytes. The internal register pointer is incremented automatically.

Calibration
Do not attempt this until you have your compass working! Especially if your using the I2C interface - get that fully working first. The module has already been calibrated in our workshop for our inclination, which is 67 degrees. If your location is close to this, you may like to try the compass without re-calibrating at all. Calibration only needs to be done once -

the calibration data is stored in EEPROM on the PIC18F2321 chip. You do not need to re-calibrate every time the module is powered up.

Compass module orientation to produce 0 degrees reading.



Before calibrating the compass, you must know **exactly** which direction is North, East, South and West. Remember these are the magnet poles, not the geographic poles. Don't guess at it. Get a magnetic needle compass and check it. When calibrating, make sure the compass is horizontal at all times with components upwards, don't tilt it. Keep all magnetic and ferrous materials away from the compass during calibration - including your wristwatch.

I2C Method

To calibrate using the I2C bus, you only have to write 255 (0xff) to register 15, once for each of the four major compass points North, East, South and West. The 255 is cleared internally automatically after each point is calibrated. The compass points can be set in any order, but all four points must be calibrated. For example

1. Set the compass module flat, pointing North. Write 255 to register 15, Calibrating pin (pin5) goes low.
2. Set the compass module flat, pointing East. Write 255 to register 15,
3. Set the compass module flat, pointing South. Write 255 to register 15,
4. Set the compass module flat, pointing West. Write 255 to register 15, Calibrating pin (pin5) goes high.

That's it.

Pin Method

Pin 6 is used to calibrate the compass. The calibrate input (pin 6) has an on-board pull-up resistor and can be left unconnected after calibration. To calibrate the compass you only have to take the calibrate pin low and then high again for each of the four major compass points North, East, South and West. A simple push switch wired from pin6 to 0v (Ground) is OK for this. The compass points can be set in any order, but all four points must be calibrated. For example

1. Set the compass module flat, pointing North. Briefly press and release the switch, Calibrating pin (pin5) goes low.
2. Set the compass module flat, pointing East. Briefly press and release the switch,

3. Set the compass module flat, pointing South. Briefly press and release the switch,
4. Set the compass module flat, pointing West. Briefly press and release the switch,
Calibrating pin (pin5) goes high.
That's it.

One point which must be emphasized. ***The calibration must be done in exactly four steps***, once for each of the four major compass points North, East, South and West. Previous versions performed part of the calibration at each step and you could actually go back and do a point again, taking 5 or more steps. Only the most recent reading from each point was used. Rev 14 onwards works differently. The 1st step (pulling pin 6 low or writing 255 to register 15) initializes internal construction registers and collects the 1st data set. The remaining steps only collect data. After the final 4th step, multiple processing stages generate and store in EEPROM a number of internal calibration values. When you perform the 1st step, Pin 5 will go low. After the 4th step it will go high again. You can connect an LED from pin 5 to 5v via a 390ohm resistor to indicate calibration is in progress. It should be high (Led off) before you start.

Rev 15 Firmware - April 2007
The default internal scan time of the CPMS03 is 100mS, or 10 updates per second 100mS is a common denominator of 50Hz/20mS (5*) and 60Hz/16.66mS (6*).
In Rev15 we have provided two further options. A 33mS scan time which is 30 updates per second, and 300mS which is 3.3 updates per second. The 33ms scan time can be used for applications that require fast updating. The penalty is that fewer samples can be taken in the available time, so there will be a slightly higher variation in the output angle. A 300mS scan time will increase the stability of the reading slightly and is suitable where the update rate is less important. Most users will not need to change this from the 100mS default.
You change the scan time by writing unlock codes to registers 12,13 and 14 and the new scan command to register 15. The scan commands are:

Scan Period	Command
300mS	0x10
100mS	0x11
33mS	0x12

For example, to change to 33mS scan time, write the following to registers 12,13,14 and 15.

Reg 12	Reg 13	Reg 14	Reg 15
0x55	0x5A	0xA5	0x12

The new scan period is stored in EEPROM so you only need to do it once, not every time the module is powered up. When the scan time is changed, the calibration is affected so you will need to recalibrate the compass. If you restore factory calibration then the scan time is reset to 100mS.

CMPS03 - Compass Module FAQ

Q. Will your compass tell me which direction is north?
A. No.

Q. Ok, will your compass tell me which way the magnetic north pole is?
A. No.

Q. So what does it do?
A. It gives you the direction of the horizontal component of the prevailing magnetic flux. OK, so I'm making a point here and it's this: The magnetic field in a building can vary hugely. Don't expect it to be always pointing north. By walking around my workshop with a standard magnetic needle compass I can make it point in any direction I like by moving past various machines, and I can do the same thing at home by going near the fridge or even the central heating radiators. The compass module will give the same results. It can only provide information about the field at its current location.

Q. Is the Compass affected by Motors, Magnets, Ferrous objects etc?
A. Yes. Anything that affects the local magnetic field will affect the Compass module. Motors in particular contain strong magnets. The only solution is to mount the compass as far away from magnetic/ferrous objects as is practicable. Also see the previous questions and answers above

Q. If the accuracy of the compass is 3-4° , how can you provide a resolution of 0.1° ?
A. Accuracy and resolution are not the same thing. A 3.5 digit multi-meter has a resolution of 1 in 2000 or 0.05%, yet the accuracy on some ranges can be 5% - a hundred time worse. A robot can use the resolution to detect small changes in direction even though there is uncertainty about the absolute direction. Also see previous answer.

Q. How do I select between I2C and PWM outputs?
A. No need to. The PWM output is always present whether you use it or not and does not require a trigger signal. The I2C interface is also always available – just connect it up and start talking.

Q. How many degrees can the module be tilted before the readings become inaccurate?
A. None. Moving the compass off horizontal will result in increasing error. The sensors are sensitive to the vertical component of the Earths magnetic field as well. The angle of the Earths field is not horizontal, it dives into the ground at an angle which varies according to location. It is this which produces an inherent error in the reading, and makes calibration of the compass required at the point where it is to be operated. After calibration you can expect 3-4° accuracy if you keep it horizontal.

Q. When measuring the PWM signal the maximum reading (which should be 359 degrees) is actually 357 degrees (or similar). Why is this?
A. This is because of slight differences in CMPS01/03 and Controller oscillators and the internal software generating and measuring the pulse. If a 359 degree roll-over is important, then you can add a "fiddle factor" to the calculation or alternatively, use the I2C Bus.

Q. Which way up should the Compass be?
A. The Compass module should be horizontal, with the PCB parallel to the earths surface. The IC's should be on top and (for the CMPS01) the Sensors underneath.

Q. What is the difference between the CMPS01 and the CMPS03?
A. For most purposes, not a lot really. The coils around the KMZ10 sensors on the underside were proving way to expensive to assemble and still sell the module at such a low cost. When Philips produced the KMZ51 sensor, an 8 pin surface mount chip with the coils built in, it was time to change. The PCB is the same physical size as the CMPS01 with the same connections and uses the same software. Calibrating the CMPS03 is the same as CMPS01 Rev7.

Q. Can the Compass be mounted near to the speaker on the SP03 speech synthesizer?
A. The speaker magnet will effect the Compass. The effects reduce with distance and are negligible at 10-12 inches

Q. My software master I2C code does not read correct data from the compass, but its works fine with an I2C EEPROM chip. Why is this?
A. The most likely cause is the master code not waiting for the I2C bus hold. This is where the slave can hold the SCL line low until it is ready. When the master releases SCL (remember it's a passive pull-up, not driven high) the slave may still be holding it low. The master code should then wait until it actually does go high before proceeding.

13. GP2D120 IR afstandmeting

Zie PDF document

14. GP2D12 IR afstandmeting

Zie PDF document

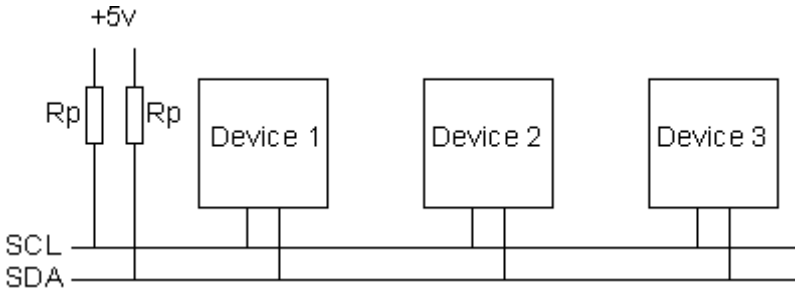
15. TCRT5000 Line follow sensor

    Zie datasheet in bijlage

16. I2C bus communicatie

It is quite clear that the I2C bus can be very confusing for the newcomer. I have lots of examples on using the I2C bus on the website, but many of these are using high level controllers and do not show the detail of what is actually happening on the bus. This short article therefore tries to de-mystify the I2C bus, I hope it doesn't have the opposite effect!

The physical I2C bus
This is just two wires, called SCL and SDA. SCL is the clock line. It is used to synchronize all data transfers over the I2C bus. SDA is the data line. The SCL & SDA lines are connected to all devices on the I2C bus. There needs to be a third wire which is just the ground or 0 volts. There may also be a 5volt wire is power is being distributed to the devices. Both SCL and SDA lines are "open drain" drivers. What this means is that the chip can drive its output low, but it cannot drive it high. For the line to be able to go high you must provide pull-up resistors to the 5v supply. There should be a resistor from the SCL line to the 5v line and another from the SDA line to the 5v line. You only need one set of pull-up resistors for the whole I2C bus, not for each device, as illustrated below:

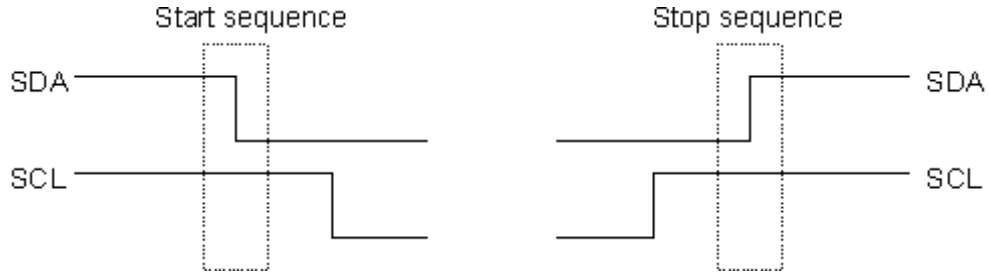


The value of the resistors is not critical. I have seen anything from 1k8 (1800 ohms) to 47k (47000 ohms) used. 1k8, 4k7 and 10k are common values, but anything in this range should work OK. I recommend 1k8 as this gives you the best performance. If the resistors are missing, the SCL and SDA lines will always be low - nearly 0 volts - and the I2C bus will not work.

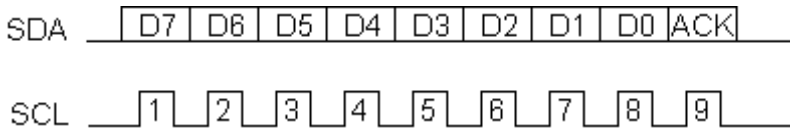
Masters and Slaves
The devices on the I2C bus are either masters or slaves. The master is always the device that drives the SCL clock line. The slaves are the devices that respond to the master. A slave cannot initiate a transfer over the I2C bus, only a master can do that. There can be, and usually are, multiple slaves on the I2C bus, however there is normally only one master. It is possible to have multiple masters, but it is unusual and not covered here. On your robot, the master will be your controller and the slaves will be our modules such as the SRF08 or CMPS03. Slaves will never initiate a transfer. Both master and slave can transfer data over the I2C bus, but that transfer is always controlled by the master.

The I2C Physical Protocol
When the master (your controller) wishes to talk to a slave (our CMPS03 for example) it begins by issuing a start sequence on the I2C bus. A start sequence is one of two special sequences defined for the I2C bus, the other being the stop sequence. The start sequence

and stop sequence are special in that these are the only places where the SDA (data line) is allowed to change while the SCL (clock line) is high. When data is being transferred, SDA must remain stable and not change whilst SCL is high. The start and stop sequences mark the beginning and end of a transaction with the slave device.



Data is transferred in sequences of 8 bits. The bits are placed on the SDA line starting with the MSB (Most Significant Bit). The SCL line is then pulsed high, then low. Remember that the chip cannot really drive the line high, it simply "lets go" of it and the resistor actually pulls it high. For every 8 bits transferred, the device receiving the data sends back an acknowledge bit, so there are actually 9 SCL clock pulses to transfer each 8 bit byte of data. If the receiving device sends back a low ACK bit, then it has received the data and is ready to accept another byte. If it sends back a high then it is indicating it cannot accept any further data and the master should terminate the transfer by sending a stop sequence.

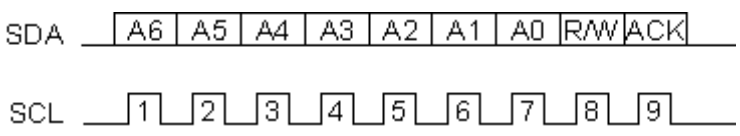


How fast?

The standard clock (SCL) speed for I2C up to 100KHz. Philips do define faster speeds: Fast mode, which is up to 400KHz and High Speed mode which is up to 3.4MHz. All of our modules are designed to work at up to 100KHz. We have tested our modules up to 1MHz but this needs a small delay of a few uS between each byte transferred. In practical robots, we have never had any need to use high SCL speeds. Keep SCL at or below 100KHz and then forget about it.

I2C Device Addressing

All I2C addresses are either 7 bits or 10 bits. The use of 10 bit addresses is rare and is not covered here. All of our modules and the common chips you will use will have 7 bit addresses. This means that you can have up to 128 devices on the I2C bus, since a 7bit number can be from 0 to 127. When sending out the 7 bit address, we still always send 8 bits. The extra bit is used to inform the slave if the master is writing to it or reading from it. If the bit is zero are master is writing to the slave. If the bit is 1 the master is reading from the slave. The 7 bit address is placed in the upper 7 bits of the byte and the Read/Write (R/W) bit is in the LSB (Least Significant Bit).



The placement of the 7 bit address in the upper 7 bits of the byte is a source of confusion for the newcomer. It means that to write to address 21, you must actually send out 42 which is 21 moved over by 1 bit. It is probably easier to think of the I2C bus addresses as 8 bit addresses, with even addresses as write only, and the odd addresses as the read address for the same device. To take our CMPS03 for example, this is at address 0xC0 ($C0). You would uses 0xC0 to write to the CMPS03 and 0xC1 to read from it. So the read/write bit just makes it an odd/even address.

The I2C Software Protocol
The first thing that will happen is that the master will send out a start sequence. This will alert all the slave devices on the bus that a transaction is starting and they should listen in incase it is for them. Next the master will send out the device address. The slave that matches this address will continue with the transaction, any others will ignore the rest of this transaction and wait for the next. Having addressed the slave device the master must now send out the internal location or register number inside the slave that it wishes to write to or read from. This number is obviously dependant on what the slave actually is and how many internal registers it has. Some very simple devices do not have any, but most do, including all of our modules. Our CMPS03 has 16 locations numbered 0-15. The SRF08 has 36. Having sent the I2C address and the internal register address the master can now send the data byte (or bytes, it doesn't have to be just one). The master can continue to send data bytes to the slave and these will normally be placed in the following registers because the slave will automatically increment the internal register address after each byte. When the master has finished writing all data to the slave, it sends a stop sequence which completes the transaction.

So to write to a slave device:
1. Send a start sequence
2. Send the I2C address of the slave with the R/W bit low (even address)
3. Send the internal register number you want to write to
4. Send the data byte
5. [Optionally, send any further data bytes]
6. Send the stop sequence.

As an example, you have an SRF08 at the factory default address of 0xE0.

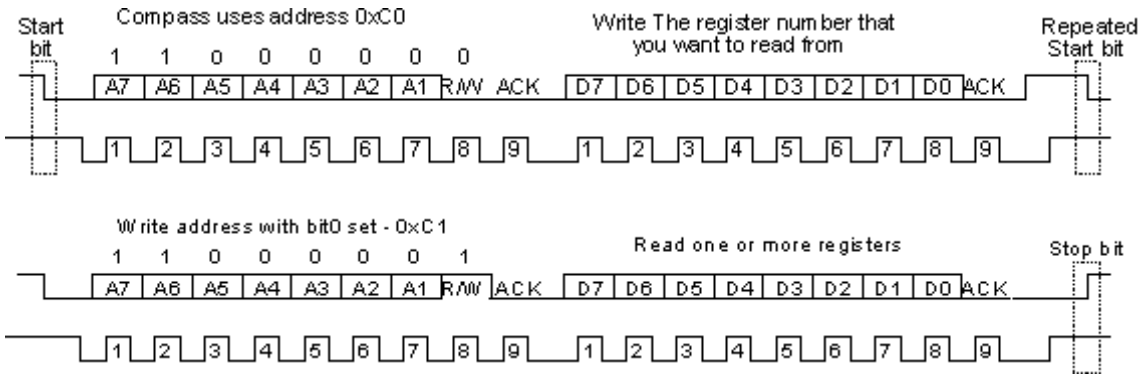
To start the SRF08 ranging you would write 0x51 to the command register at 0x00 like this:
1. Send a start sequence
2. Send 0xE0 (I2C address of the SRF08 with the R/W bit low (even address)
3. Send 0x00 (Internal address of the command register)
4. Send 0x51 (The command to start the SRF08 ranging)
5. Send the stop sequence.

Reading from the Slave
This is a little more complicated - but not too much more. Before reading data from the slave device, you must tell it which of its internal addresses you want to read. So a read of the slave actually starts off by writing to it. This is the same as when you want to write to it: You send the start sequence, the I2C address of the slave with the R/W bit low (even address) and the internal register number you want to write to. Now you send another start sequence

(sometimes called a restart) and the I2C address again - this time with the read bit set. You then read as many data bytes as you wish and terminate the transaction with a stop sequence.

So to read the compass bearing as a byte from the CMPS03 module:
1. Send a start sequence
2. Send 0xC0 (I2C address of the CMPS03 with the R/W bit low (even address)
3. Send 0x01 (Internal address of the bearing register)
4. Send a start sequence again (repeated start)
5. Send 0xC1 (I2C address of the CMPS03 with the R/W bit high (odd address)
6. Read data byte from CMPS03
7. Send the stop sequence.

The bit sequence will look like this:



Wait a moment
That's almost it for simple I2C communications, but there is one more complication. When the master is reading from the slave, its the slave that places the data on the SDA line, but its the master that controls the clock. What if the slave is not ready to send the data! With devices such as EEPROMs this is not a problem, but when the slave device is actually a microprocessor with other things to do, it can be a problem. The microprocessor on the slave device will need to go to an interrupt routine, save its working registers, find out what address the master wants to read from, get the data and place it in its transmission register. This can take many uS to happen, meanwhile the master is blissfully sending out clock pulses on the SCL line that the slave cannot respond to. The I2C protocol provides a solution to this: the slave is allowed to hold the SCL line low! This is called clock stretching. When the slave gets the read command from the master it holds the clock line low. The microprocessor then gets the requested data, places it in the transmission register and releases the clock line allowing the pull-up resistor to finally pull it high. From the masters point of view, it will issue the first clock pulse of the read by making SCL high and then check to see if it really has gone high. If its still low then its the slave that holding it low and the master should wait until it goes high before continuing. Luckily the hardware I2C ports on most microprocessors will handle this automatically.

Sometimes however, the master I2C is just a collection of subroutines and there are a few implementations out there that completely ignore clock stretching. They work with things

like EEPROM's but not with microprocessor slaves that use clock stretching. The result is that erroneous data is read from the slave. Beware!

Example Master Code

This example shows how to implement a software I2C master, including clock stretching. It is written in C for the PIC processor, but should be applicable to most processors with minor changes to the I/O pin definitions. It is suitable for controlling all of our I2C based robot modules. Since the SCL and SDA lines are open drain type, we use the tristate control register to control the output, keeping the output register low. The port pins still need to be read though, so they're defined as SCL_IN and SDA_IN. This definition and the initialization is probably all you'll need to change for a different processor.

```
#define SCL     TRISB4 // I2C bus
#define SDA     TRISB1 //
#define SCL_IN  RB4    //
#define SDA_IN  RB1    //
```

To initialize the ports set the output resisters to 0 and the tristate registers to 1 which disables the outputs and allows them to be pulled high by the resistors.

```
SDA = SCL = 1;
SCL_IN = SDA_IN = 0;
```

We use a small delay routine between SDA and SCL changes to give a clear sequence on the I2C bus. This is nothing more than a subroutine call and return.

```
void i2c_dly(void)
{
}
```

The following 4 functions provide the primitive start, stop, read and write sequences. All I2C transactions can be built up from these.

```
void i2c_start(void)
{
  SDA = 1;          // i2c start bit sequence
  i2c_dly();
  SCL = 1;
  i2c_dly();
  SDA = 0;
  i2c_dly();
  SCL = 0;
  i2c_dly();
}

void i2c_stop(void)
{
  SDA = 0;          // i2c stop bit sequence
  i2c_dly();
  SCL = 1;
```

```
 i2c_dly();
 SDA = 1;
 i2c_dly();
}

unsigned char i2c_rx(char ack)
{
char x, d=0;
 SDA = 1;
 for(x=0; x<8; x++) {
  d <<= 1;
  do {
   SCL = 1;
  }
  while(SCL_IN==0);    // wait for any SCL clock stretching
  i2c_dly();
  if(SDA_IN) d |= 1;
  SCL = 0;
 }
 if(ack) SDA = 0;
 else SDA = 1;
 SCL = 1;
 i2c_dly();            // send (N)ACK bit
 SCL = 0;
 SDA = 1;
 return d;
}

bit i2c_tx(unsigned char d)
{
char x;
static bit b;
 for(x=8; x; x--) {
  if(d&0x80) SDA = 1;
  else SDA = 0;
  SCL = 1;
  d <<= 1;
  SCL = 0;
 }
 SDA = 1;
 SCL = 1;
 i2c_dly();
 b = SDA_IN;         // possible ACK bit
 SCL = 0;
 return b;
}
```

The 4 primitive functions above can easily be put together to form complete I2C transactions. Here's and example to start an SRF08 ranging in cm:

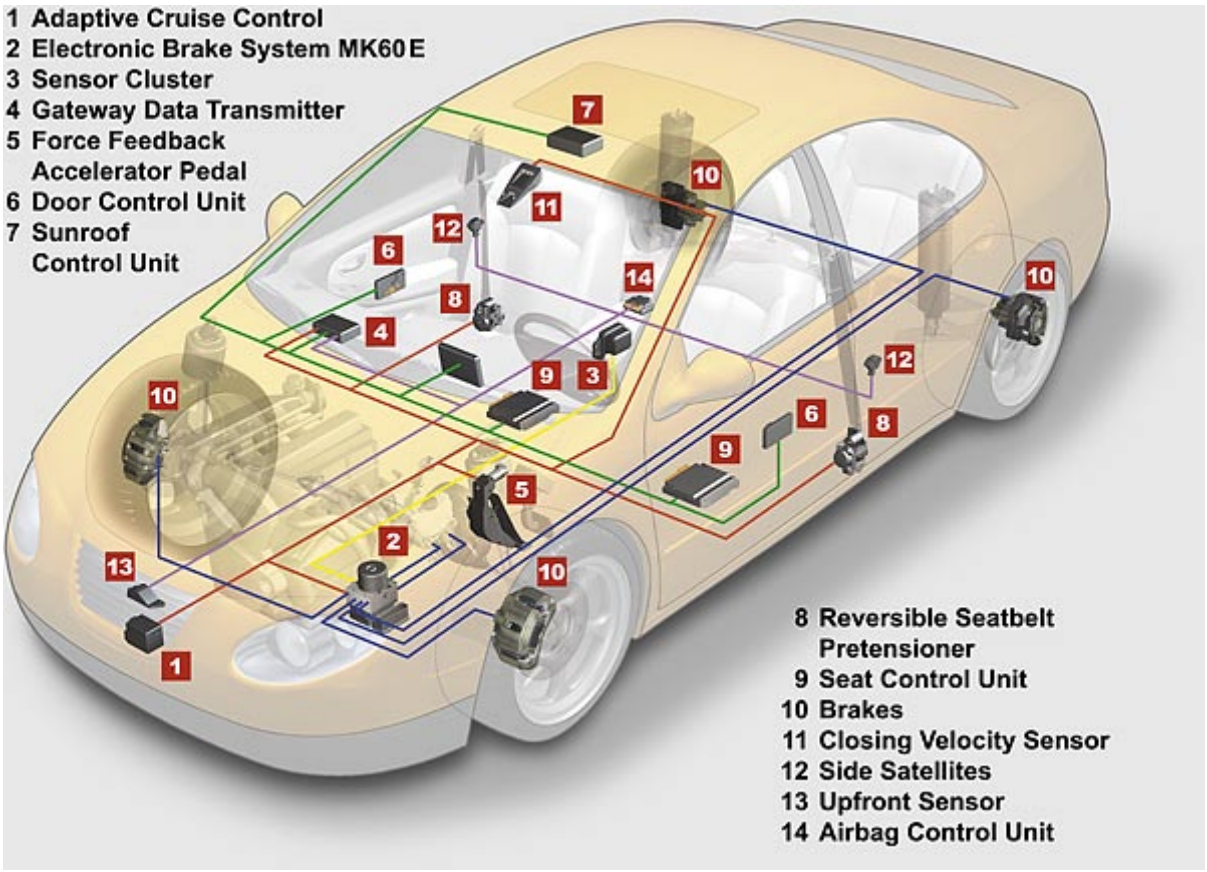
```
i2c_start();          // send start sequence
i2c_tx(0xE0);           // SRF08 I2C address with R/W bit clear
i2c_tx(0x00);           // SRF08 command register address
i2c_tx(0x51);           // command to start ranging in cm
i2c_stop();           // send stop sequence
```

Now after waiting 65mS for the ranging to complete (I've left that to you) the following example shows how to read the light sensor value from register 1 and the range result from registers 2 & 3.

```
i2c_start();          // send start sequence
i2c_tx(0xE0);           // SRF08 I2C address with R/W bit clear
i2c_tx(0x01);           // SRF08 light sensor register address
i2c_start();          // send a restart sequence
i2c_tx(0xE1);           // SRF08 I2C address with R/W bit set
lightsensor = i2c_rx(1);  // get light sensor and send acknowledge. Internal register address will increment automatically.
rangehigh = i2c_rx(1);    // get the high byte of the range and send acknowledge.
rangelow = i2c_rx(0);     // get low byte of the range - note we don't acknowledge the last byte.
i2c_stop();           // send stop sequence
```

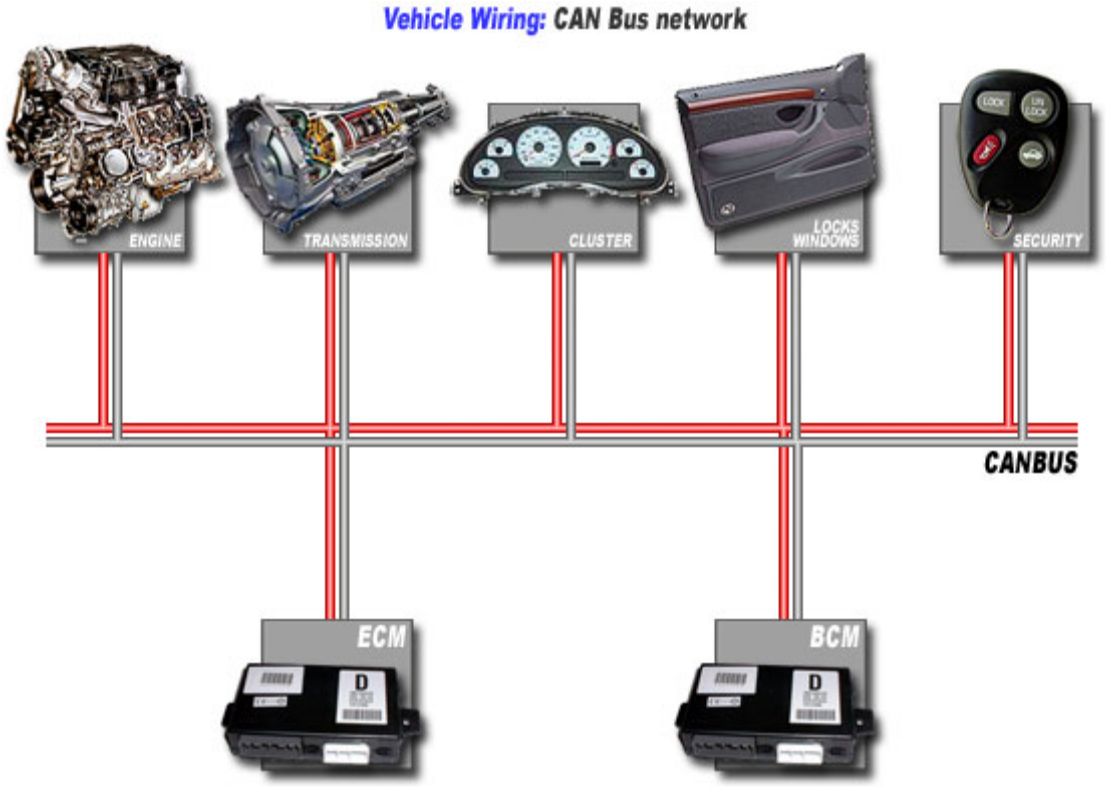
Easy isn't it?

17. CAN-bus communicatie



CAN BUS staat voor Controller Area Network. Dit bussysteem is speciaal ontworpen voor gebruik in wagens maar is door z'n grote storings-ongevoeligheid ook populair als industriëel bussysteem.

Een moderne wagen heeft enorm veel contacten – schakelaars – lampjes en sensoren die allemaal op één of andere manier met de centrale boordcomputer moeten verbonden worden. Dat zorgt voor een enorm gewicht aan koperdraad wat op zijn beurt weer voor meer verbruik en vervuiling zorgt.

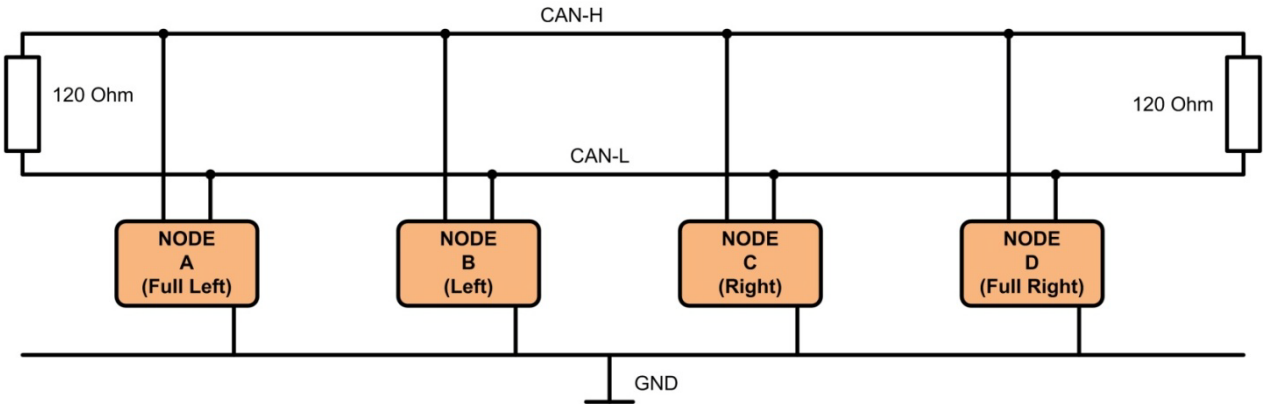


Bij een CAN bus systeem worden er enkele tientallen microcontrollers over de verschillende delen van de wagen verdeeld. Elke schakelaar en sensor in de buurt van deze microcontroller wordt hiermee verbonden met korte draden. Alle microcontrollers worden via een CAN bus systeem met elkaar verbonden en kunnen zo de noodzakelijke gegevens uiwisselen. Dit can bus netwerk heeft slechts 2 draden nodig en zo besparen we gewicht.

Eigenschappen:

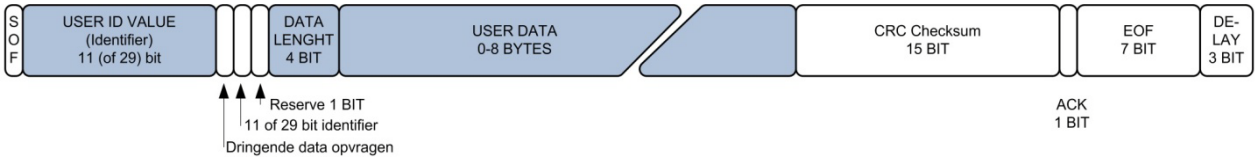
- Controller Area Network (Data link layer)
- Differential 2 wire (STP, UTP, Ribbon Cable) (Physical layer)
- NRZ (Non return to zero) + Bit stuffing
- Max 1Mbps (40m) – 50kbps (1km)
- Can bus E-block = 125kbps (300-600m)
- Hamming distance of 6 – 5 random bit errors will be detected

De verschillende microcontrollers of nodes in een can bus netwerk takken éénder waar op de twee draden af. Er dient wel op gelet te worden dat de twee uiteinden afgesloten worden met weerstanden van 120 Ohm om reflecties in de kabels te vermijden. We mogen ook niet vergeten om de GND van alle nodes door te verbinden. We hebben dus eigenlijk een 3-draads netwerk (Of tweedraads in wagens vermits het metalen chassis als gemeenschappelijke ground dienst doet).



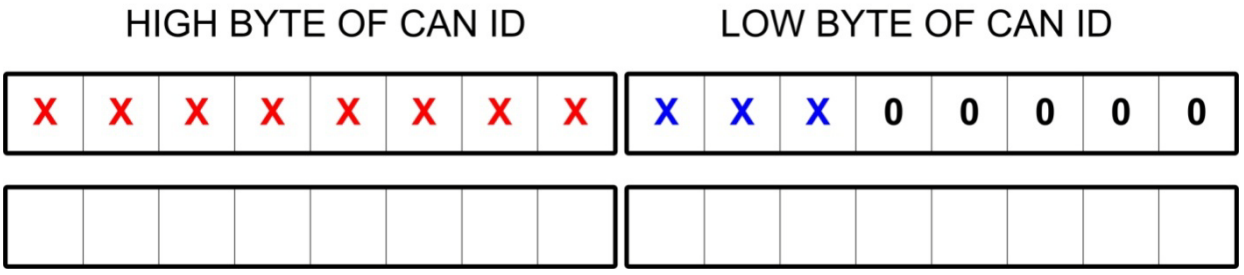
Het CAN systeem werk niet met adressering. De verschillende nodes in het systeem hebben geen uniek adres. Wat er wel wordt gedaan is dat elk bericht een unieke identifier mee krijgt. Het bericht dat van de sensor onder de rempedaal komt heeft een andere identifier dan het bericht dat van de schakelaar voor de achterruitverwarming komt. Elke node beslist zelf of het bericht dat voorbij komt nut heeft voor deze node. Een bericht dat van de rempedaal komt zal worden opgepikt door de microcontrollers van de linker achterlichten – de rechter achterlichten – het derde stoplicht en van de remsystemen. De microcontroller in één van de protieren heeft niets aan dit bericht en zal dit bericht dan ook niet binnen pakken.

Zo'n identifier kon oorspronkelijk bestaan uit 11 bits wat betekent dat er binnen één can bus systeem maximaal 2048 verschillende identifiers mogelijk zijn. Voor vrachtwagens en landbouwvoertuigen bleek dit niet te volstaan en daarvoor heeft men een nieuwe can bus standaard ontworpen die 29 bit identifiers toelaat wat betekent dat er 536 870 912 verschillende indentifiers mogelijk zijn binnen elk can bus systeem. Dat zou even moeten volstaan.



Zoals hierboven te zien is is zo'n CAN bus bericht samengesteld uit verschillende delen. Wij moeten enkel de donkere delen invullen – de rest wordt aangevuld door de CAN IC's.

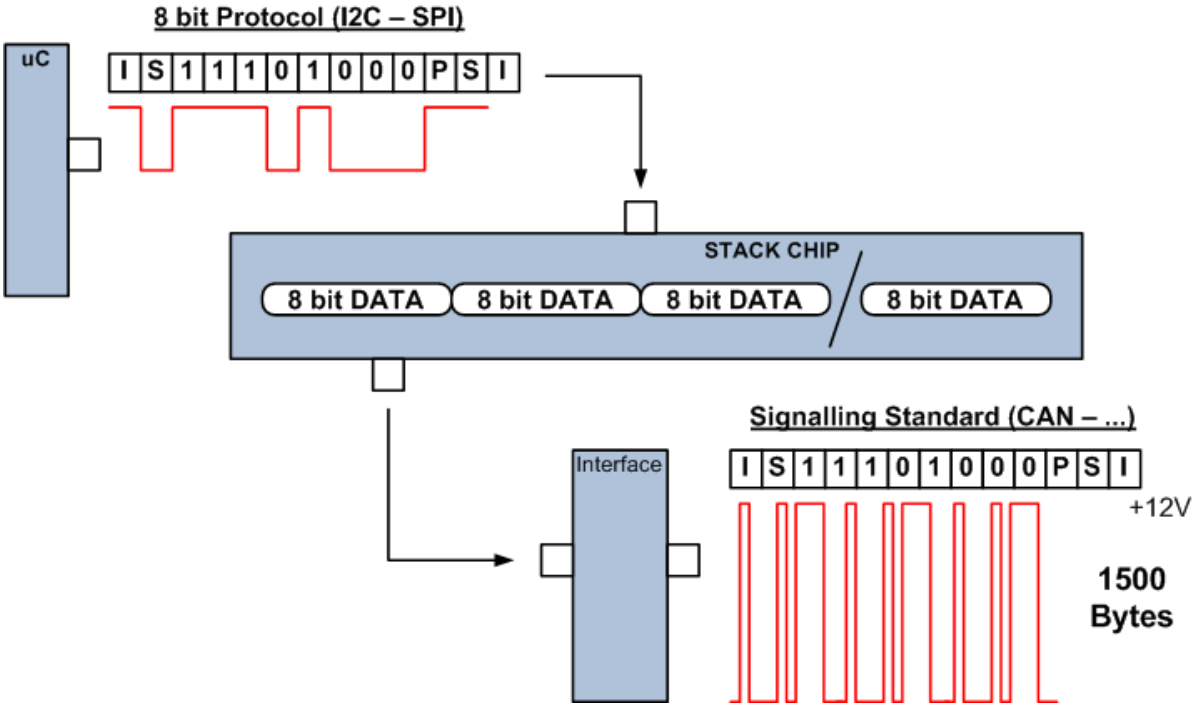
11 bit identifier:



Ten eerste de identifier. Vermits dit een 11 bit waarde is moeten we hiervoor 2 bytes gebruiken. Een High en een LOW byte. Let er op dat de hoogste 3 bits van de low byte de 3 LSB's zijn van de identifier.

Vervolgens een dada lengte. Je geeft mee hoeveel bytes date er zullen worden mee gestuurd. Dit kan variëren tussen 0 en 8 bytes.

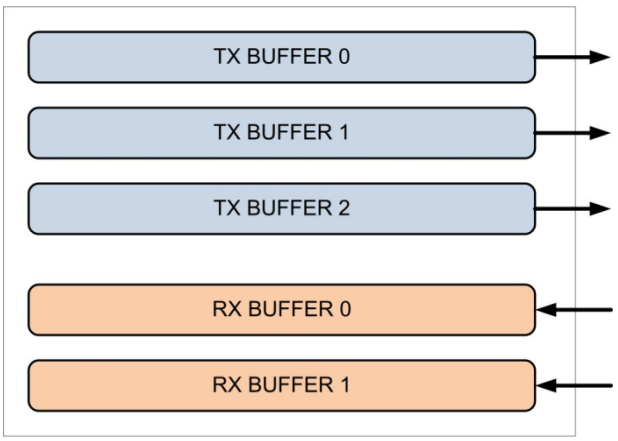
Tot slot het bericht – dit kan bestaan uit minimaal 0 tot maximaal 8 bytes.



Vermits we slechts een 8 bit microcontroller hebben en het CAN bus bericht is opgebouwd uit meerdere bytes data, moeten we dit bericht eerst samen stellen. Dat samenstellen gebeurt in een externe IC – de zogenaamde CAN STACK IC.

Van het moment dat het volledige bericht byte voor byte is opgebouwd – dan pas sturen we dit bericht als één geheel door naar buiten. Een interface IC zorgt er nog voor dat de spanningsniveau's ed worden aangepast aan de CAN BUS signaal standaard.

De stack IC kan heeft verschillende zend en ontvangstbuffers waarin gelijktijdig verschillende berichten in kunnen worden bewaard.



Init

CAN INSTRUCTIES IN FLOWCODE

Sets up the CAN component. Needs to be placed at the beginning of the program for the CAN component to work.

SendBuffer
This macro sends the information currently held in the specified <Tx_buffer> onto the CAN bus network.

CheckRx
This macro returns a value to indicate whether a message has been received by the specified <Rx_buffer>. A zero value indicates that a message has not been received, while a non-zero value indicates that a message has been received.

GetRxDataCount(buffer)
This macro returns a value that specifies how many data bytes are contained in the message received via the specified receive buffer. This macro would only be used if the user did not know how many bytes were contained in the message or if the message had a varying length.

GetRxData(buffer, index)
Returns a data byte from the location index in the specified buffer.

SetTxData(buffer, data_cnt, d0, d1, d2, d3, d4, d5, d6, d7)
This macro configures the Data for the specified buffer.
buffer is the Tx buffer to be used (0-2).
data_cnt is the number of bytes of data in the message.
IMPORTANT - This macro changes property values. The values listed on the property pages will then no longer be accurate. You will need to track the changed values yourself to ensure they are correct.

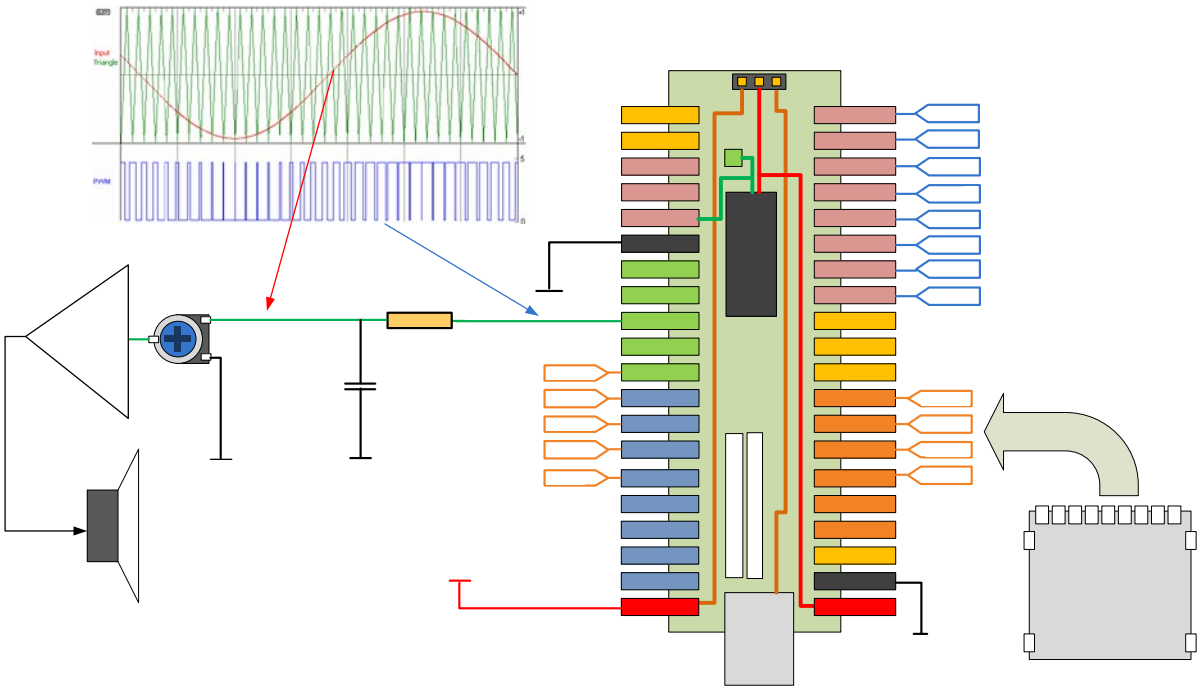
SetTxID(buffer, hi, lo)
Sets the Message ID value.
The Message ID value is a number between 0 and 2047 (0x00 and 0x7FF). However PICmicro's use 8 bit numbers 0-255 (0x00-0xFF) and so must use a set of two ID values to send the whole ID value.
The hi parameter is used for High byte of the Message ID value.
The lo parameter is used for Low byte of the Message ID value.

char GetRxIDLo(buffer)
Gets the Low byte of the Message ID.
This can be used along with the Hi byte of the Message ID to determine if the Message ID is to be enacted upon.

char GetRxIDHi(buffer)
Gets the High byte of the Message ID.

This can be used along with the Low byte of the Message ID to determine if the Message ID is to be enacted upon.

18. Geluid afspelen



De Ecio kan in Flowcode geprogrammeerd worden om WAV files – die je vooraf op de SD card hebt gezet om te zetten in geluid.

De wav files worden met SPI protocol ingelezen uit de SD card. De gegevens bestaan uit 8 bit samples die tegen een snelheid van 8000 bytes per seconde uit de SD card gehaald worden.

Deze samples worden gebruikt om de duty cycle van de PWM uitgang op pin C2 te beïnvloeden. Dit zeer snelle PWM signaal wordt via een eenvoudig RC laagdoorlaatfilter omgezet naar een geluidssignaal.

Dit geluidssignaal wordt vervolgens versterkt en dan naar de luidspreker gestuurd.

Verschillende files kunnen worden afgespeeld – elke file krijgt een unieke naam die ook in Flowcode dient gebruikt te worden.

Er mogen maximaal SD kaarten van 512Mb gebruikt worden.

Uitleg bij Module:

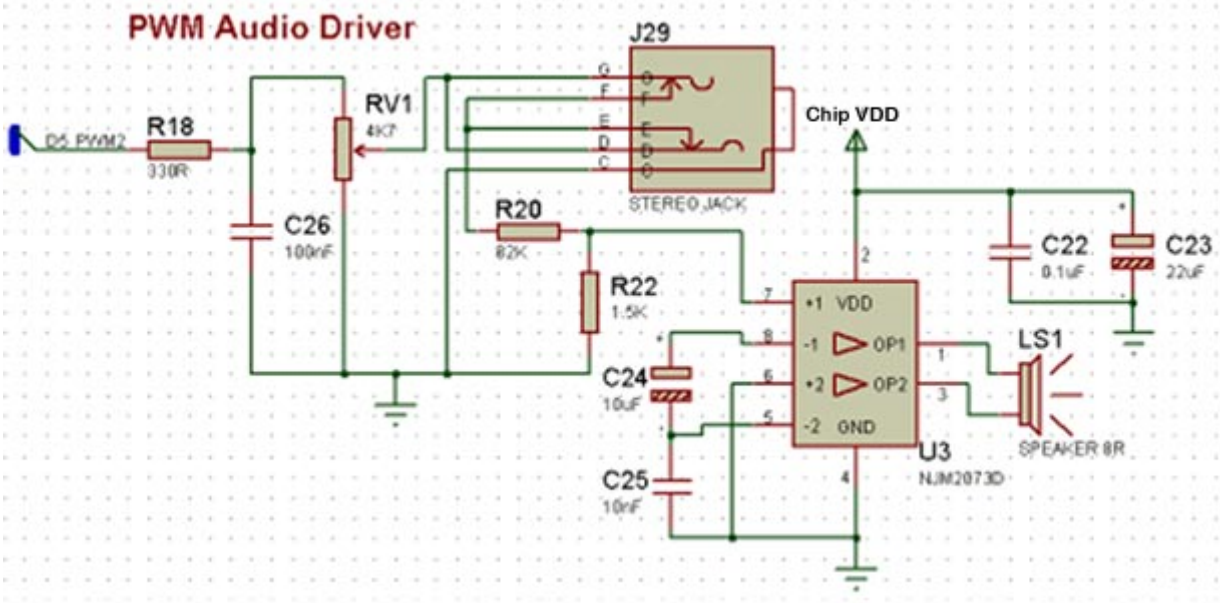
MP3 players are very popular in today's technology market and come in many shapes and sizes. Using the MMC/SD card reader E-block EB037 it is possible to incorporate files straight into the microcontroller devices ready for streaming out to a pair of headphones or a loudspeaker. To save having to buy an expensive dedicated MP3 chip we will instead be using WAV files as they are uncompressed and easy to stream out via a DAC or PWM output. It would not be much of a push to forward the audio data onto an MP3 chip once the

operation of the FAT16 code is understood. The FAT16 driver will only work with the PIC18 series and the ARM devices. AVR device support is on route.

The WAV files can be created using Windows Sound Recorder, Winamp, Audio Recorder for Free plus many other free audio software packages. The format we need is either an 8-bit @ 8khz mono for the PICmicro example. The WAV files are then placed onto the MMC/SD ready for streaming.

Both the PICmicro and the ARM Flowcode examples employ a buffering scheme to allow the audio data to be streamed without causing ticks or glitches in the playback audio signal. The files are read from the memory card in chunks of 512 bytes called sectors. Two neighbouring sectors of the file may not be located one after another on the physical card memory so there is a small amount of time to seek out and read the next sector for the card. Don't worry this is all handled automatically by the Flowcode FAT16 driver. However it does mean that the previously mentioned buffering technique needs to be used to keep the audio streaming correctly. As each set of 512 bytes is loaded there is a timer interrupt that starts streaming out the data. While this is happening a section at the end of the 512 bytes is stored into a local array. This is done so that when the timer interrupt is nearing the end of the 512 bytes it can stream the audio from the local buffer while the FAT routine is moving and collecting the next file sector.

The audio output can be driven by using an 8-bit R2R Ladder DAC connected to one of the chips ports or by using a single PWM output as the speaker device will remove all of the high frequency noise created. Both the example files use the PWM method as this seems to be less susceptible to noise and it only uses one I/O pin instead of eight. To take the design further an audio amplifier chip can be used connected to the PWM output to push and pull the signal high and low with more power. The NJM2073D is a 2 Watt audio amplifier that can be purchased from Rapid for £0.79 order code 82-2154.


PWM Audio Driver
J29
STEREO JACK
Chip VDD
D5 PWM2
R18
330R
RV1
4K7
C26
100nF
R20
82K
R22
1.5K
C24
10uF
C25
10nF
+1 VDD
-1 OP1
+2 OP2
-2 GND
U3
NJM2073D
C22
0.1uF
C23
22uF
LS1
SPEAKER 8R

19. Mogelijke oefeningen

1) Laat alle leds van de mond van de robot aan en uit flikkeren.
2) Maak een looplichtje op de leds van de mond
3) Zet de rode leds van het linker oog aan.
4) Zet de blauwe leds van het rechter oog aan.
5) Zet de laser van de neus aan en uit.
6) Maak een flikkerlicht met de blauwe backlight leds in het hoofd van de robot.
7) Genereerd een PWM signaal op alle 6 kanalen van de RGB leds in de ogen en meng zo de kleuren tot je de paarse kleuren van de robot evenaart.
8) Lees de waarde van de LDR in het linkse oog in en visualiseer deze waarde op de 8 leds van de mond.
9) Lees de waarde van de LDR in het rechts oog in en visualiseer deze waarde op de 8 leds van de mond in VU meter stijl. Dus niet in binaire vorm maar nu gaat voor elke stijging in lichtintensiteit één led meer aan.
10) bestuur de servo motoren en laat de robot Ja-knikken
11) bestuur de servo motoren en lat de robot nee-knikken.
12) Lees de eerste echo van de SRF08 Ultrasoon sensor in de ogen in en visualiseer deze waarde op de leds in de mond.
13) Lees de tweede echo in en visualiseer op de leds van de mond.
14) Lees de waarde van één van de SHARP IR sensoren in en visualiseer deze waarde op de LCD
15) Lees de waarde van alle SHARP IR sensoren in en visualiseer deze waarden mooi op de LCD.
16) Lees de waarde van de Compas sensor in en visualiseer op de LCD
17) Lees de eerste, de tweede en de derde echo van de SRF08 Ultrasoon sensor op de stappenmotor in en visualiseer op de LCD.
18) Bestuur de stappenmotor zo, zodat de SRF08 sensor een gezichtsveld van 1800 heeft – visualiseer de eerste en de tweede echo op de LCD.
19) Zet een voorwerp in het gezichtsveld van de SRF08 sensor – mbv de stappenmotor ga je eerst alle waarden inlezen – daarna draait de

stappenmotor de sensor terug in de richting van het dichtstbijzijnde voorwerp.

20) Laat de robot rond z'n eigen as draaien (linkse wiel traag vooruit – rechter wiel traag achteruit)
21) Gebruik de MD25 motor driver om de batterijspanning en de stroom op de LCD te visualiseren op een correcte manier.
22) Gebruik de wheelencoder om het linker wiel telkens met stappen van exact 90 graden vooruit en terug achteruit te laten draaien.
23) Gebruik de wheelencoder om de volledige robot telkens exact 90° rond zijn eigen as te laten draaien en dan telkens 3 seconden stil te staan. Dit is als voorbereiding om een bocht van exact 90° te maken.
24) Gebruik de wheelencoder om de robot exact 1 meter vooruit te laten rijden – dan 180° te draaien en terug 1 meter vooruit te rijden en weer 180° te draaien. De robot moet terug op de plaats van vertrek staan nu.
25) Laat de robot rond rijden zonder ooit obstakels te raken of zonder ooit te stoppen met rijden.
26) Laat de robot een muur volgen zonder deze te raken.
27) Laat de robot een zwarte lijn op een witte achtergrond volgen.
28) Gebruik CAN bus om de meetwaarde van de SRF08 sensor op de stappenmotor te visualiseren op de leds in de mond van de robot.
29) Gebruik CAN bus om de meetwaarde van de SRF08 sensor in de ogen van het hoofd zichtbaar te maken op de LCD
30) Laat een zelf-opgenomen muziekfile horen op de luidspreker. (neem een wav file op met 8 bit sample diepte en een sample frequentie van 8KHz en geef deze dezelfde naam als je in het programma gebruikt)
31) Gebruik de ECIO op de sturing van de body als master. Deze geeft aan de Muziek en het hoofd door (via CAN) wat en wanneer zij iets moeten uitvoeren. Bouw zo stap voor stap een hele show op.

20. FAQ

1) **Ik wil mijn ECIO programmeren, maar het lukt niet. Ik heb nochtans de robot wel aan gezet.**
    - Heb je op de reset schakelaar van de ECIO gedrukt. De ECIO moet in bootloader mode staan om een nieuw programma te kunnen inladen.
    - Heel soms blijft de ECIO in bootloader mode staan – haal de ecio er dan uit – verzet de jumper naar USB – en probeer zo het programma in te laden – soms moet je dit meerdere malen terug proberen. Vergeet niet om nadien de jumper terug te verzetten.
    - Is de batterijspanning van de robot nog hoog genoeg.

2) **Wanneer moet ik de batterij terug opladen?**
    - Als de batterijspanning – aangegeven op de spanningsmeter aan de zijkant – daalt onder de 11,5Volt.

3) **Hoe lang kan ik met volle batterijen 'spelen'?**
    - Dat hangt natuurlijk af van uw verbruik. De grootste verbruikers zijn de RGB leds in de ogen (60 leds x 20mA) – de DC motoren voor de wielen en de Stappenmotor als die stilstaat (niet in rust).

21. Voorbeeldprogramma's

1) Leds in mond
2) Kleuren in ogen
3) Laser
4) LDR in ogen
5) Backlight in hoofd
6) SRF08 sensor in hoofd
7) LCD
8) Compas sensor
9) SHARP sensoren
10) SRF08 sensor
11) Stappenmotor
12) DC motoren
13) Spanningsmeting met MD25
14) CAN bus communicatie
15) Geluid van een SD card afspelen
16) 3 Testprogramma's alle sensoren en actuatoren
    - Test sturing body
    - Test sturing head
    - Test sturing muziek

22. Materiaallijsten componenten:

Active Robots

RD01 Complete robot drive system
GP2D12 distance sensor 10-80
GP2D120 distance sensor 4-30
Sharp GP2Y0A21YK Sensor w/lead
SRF08 sensor only SRF08
CMPS03 compas module
MRM-O17 pair of Ball casters (niet meer OK)
16x4 LCD display module 16X4-LCD
BPT-KT LYNX Pan and tilt kit
Servomotors HS-422

FARNELL

HEAD:	Aantal	bestelcode
Laser PLP6501FR	1	1209950
LDR	2	1652638
Resistor SMD 1206 120Ohm	100	9337059
Resistor SMD 1206 100Ohm	50	9336990
led blue true hole	10	8576769
led blue voor mond smd	10	1058403
led SMD RGB Ogen	30	1618713
header 36 pin recht naar boven	2	1097954
mosfet - 1583665	10	1583665

VOEDING:	Aantal	bestelcode
Battery 2,3ah	2	174786
DCDC convertor 12-5V Ten 25-1211	1	1204988
Fuse holder	4	146123
Diode 10A	1	9099352
7806 6V voltage regulator (TO220)	1	1468763
led 5mm blue TH	2	8576769
10uF Elko 16V 1,5mm pitch	1	3017357
100nF 5mm pitch	1	1457699
3300uf 16V Axial	2	1165378
Batterijlader	1	1205745
Voltmeter - batterijspanning	1	9932712

MUZIEK:	Aantal	bestelcode
SD card reader	1	9186158
MAX3002 level convertor (via samples)	1	
LM1117 3,3V regulator	2	1469056
R 220 Ohm	4	9339299
R 10 Kohm	2	9339060
R 120 Ohm	2	9339116
MCP2515 CAN bus	1	1439391
MCP2551 CAN bus	1	1292240
X-tal 20MHz	3	9712879
16 pin IC voet	2	1101347
8 pin IC voet	2	1101345
18 pin IC voet	2	1101348
40 pin IC voet	2	1101352
potmeter 5KOhm	1	108238
led 5mm blue TH	1	8576769
100nF 5mm pitch	5	1457699
10uF Elko 16V 1,5mm pitch	2	3017357
470 uF Elko 2,5 pitch	1	8767041
47 uf 2mm pich	1	9451072
100uF 3,5mm pitch	3	9451285
speaker	1	1683898
Header 36 pin 90°	2	9729100
TDA7267A (via Spoerle)		order ID: 4536

STURING BODY	Aantal	bestelcode
8 pin IC voet	2	1101345
18 pin IC voet	2	1101348
MCP2515 CAN bus	1	1439391
MCP2551 CAN bus	1	1292240
X-tal 20MHz	1	9712879
potmeter 5KOhm	1	108238
40 pin IC voet	1	1101352
100uF 3,5mm pitch	2	9451285
L293D Motor driver IC	2	9589619
10uF Elko 16V 1,5mm pitch	2	3017357
Header 36 pin 90°	3	9729100
Stappenmotor	1	9598642
16 pin IC voet	2	1101347
Afstandbusjes (pakje 25 st)	1	666865

STURING HOOFD:	Aantal	bestelcode
8 pin IC voet	1	1101345
18 pin IC voet	1	1101348
MCP2515 CAN bus	1	1439391
MCP2551 CAN bus	1	1292240
X-tal 20MHz	1	9712879
40 pin IC voet	1	1101352
potmeter 5KOhm	1	108238
100uF 3,5mm pitch	3	9451285
R 82 Ohm	8	513763
Header 36 pin 90°	1	9729100
led 5mm blue TH	10	8576769
kabelhoes klein	1	1416082
kabelhoes groot	1	1416083
krimpkous 1,2m	2	1210430
krimpkous 5m	1	1008465

LIJNVOLGER	Aantal	bestelcode
TCRT5000	3	1470066
TAOS RGB Sensor TCS230	2	4891211

23. LOGBOEK

Gelieve in dit logboek duidelijk bij te houden wat er met deze robot gebeurt.

Datum	Omschrijving

Datum	Omschrijving