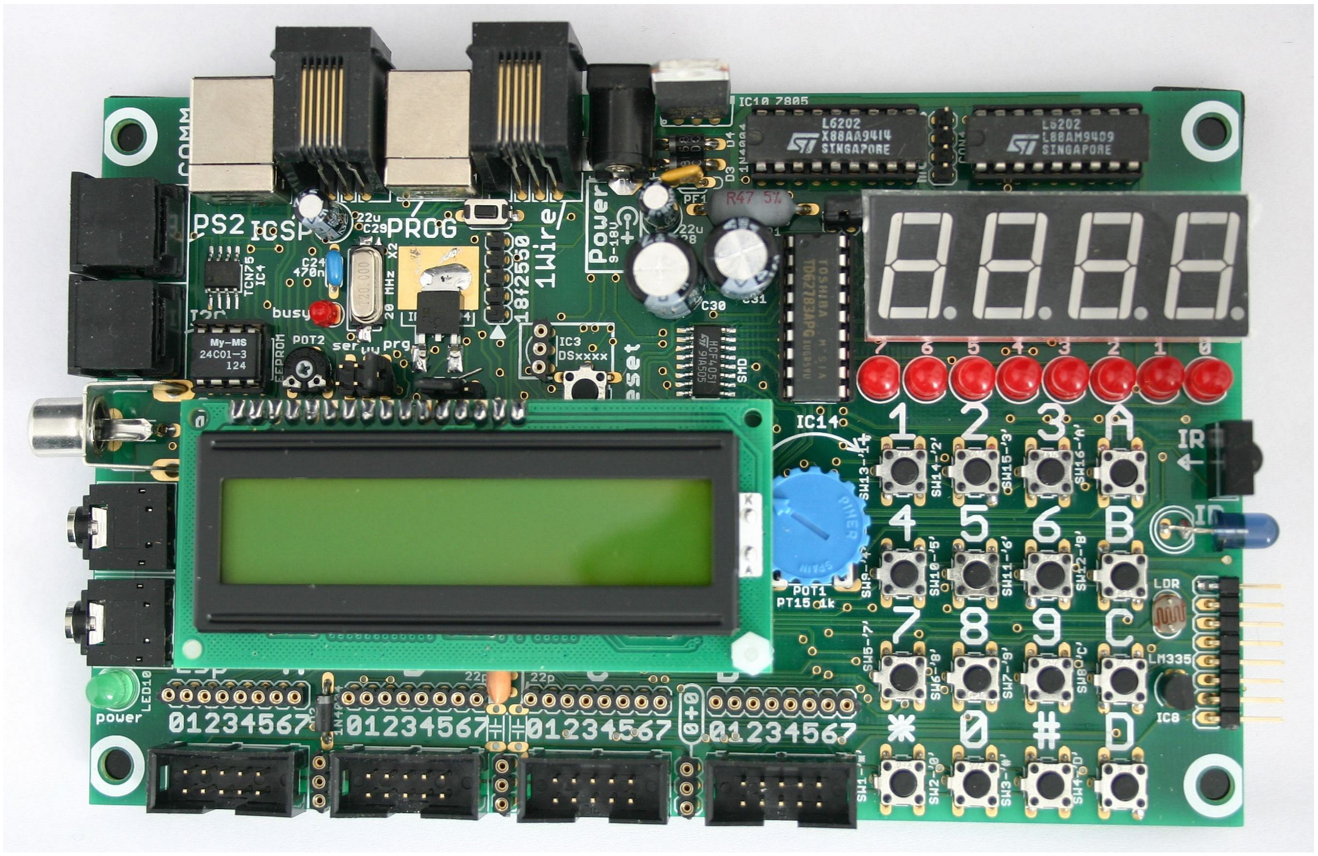
Dwarf Boards

DB038 : A PIC experiments board

(c) Van Ooijen Technische Informatica - Wouter van Ooijen
Version 2.00, last modified 20-apr-2010, for DB038 V1.05



The board is shown here without the character LCD. Near the centre of the board is an empty place where an optional DS18S20 temperature sensor can be fitted.

Table of content

1 Introduction

The DB038 is a PIC experiments board. It combines a PICkit2-compatible USB programmer and a PIC 16F887 target microcontroller (the user can substitute a pin-compatible 40-pins chip) with a wide selection of on-board peripherals and a range of connectors for external peripherals. The board (except for the motor drivers) can be powered from a USB interface, so one cable to a Windows PC[1] is all you need to get going. This is the ideal board for a PIC programming introduction course.

A fully populated DB038 board contains:

- a simplified PICkit2-compatible programmer (connects to a Windows PC via USB)
- a programming connector (RJ 6/6 connector, ICD2 compatible), can be used to program an external chip (this requires that on-board target PIC is removed)
- a connector for a wall-wart, battery, or other power supply (to operate the board without a PC)
- a PIC 16F887 microcontroller with a 20 MHz resonator
- 8 LEDs
- 4 seven-segment LED displays
- a 16-key keypad (an external 4x4 matrix keypad can be connected in parallel)
- a 16 character x 2 line LCD (other character LCDs can be connected)
- two L6202 H bridge motor driver chips, for connecting two small DC motors
- a TSOP 1736, 1737 or 1738 IR remote control receiver chip
- an IR emitter LED
- an TCN75 or TCN75A temperature sensor (I2C interface, compatible with the popular LM75)
- room for a DS18S20 temperature sensor (Dallas one-wire interface), but this sensors is not populated
- an LM335 temperature sensor (analog input)
- a TL431 2.5 Volt voltage reference (analog input)
- a second USB interface, uses an FT232RL chip to connect to the UART pins of the PIC 16F887
- an LDR (analog input)
- a trim potentiometer (analog input)
- a Dallas one-wire interface connector (RJ 6/6 connector)
- a PS/2 connector for connection of a PS/2 mouse or keyboard
- an I2C connector
- an EEPROM chip (I2C interface)
- a (very simple) composite video output (cinch connector)
- a small loudspeaker, and a connector for an external PC-speaker
- a microphone connector (analog input)
- four Dwarf Bus ML10 connectors, with parallel wire cups for the data and power lines (except RA4)

This wide selection of on-board peripherals and interfaces provides a challenge for each programmer, novice and guru alike.

The Introduction section (p 6) contains detailed descriptions of each on-board peripheral and interface.

[1] PCLinux software is available from third parties, but has not been tested with this board.

A DB038 board can be programmed from a Windows XP PC using the stand-alone PICkit2 programming utility (version 1.20 only, later version do not work or have not been tested). A similar program for Linux exists, but has not been tested.

The following additional items might be useful when working with a DB038 board:

- a 12-volt wall-wart (200mA for the board, more when you use motors, type depends on the mains voltage and sockets in your country), and/or a 12-volt battery
- a PC loudspeaker set or headphone
- a (PC) microphone
- two standard USB cables (A-B)
- a cinch-to-SCART or other suitable cable to make a connection to your TV set (and the TV set itself)
- a PC mouse or keyboard (only one can be connected at a time)
- an IR remote control (36 or 38 kHz IR frequency, most remotes use one of these frequencies)
- a television set that can be controlled by IR
- two small DC motors
- DS18S20 and/or other Dallas one-wire peripherals
- I2C peripherals
- A 4x4 matrix keypad

In this document the notation 0b is used to prefix a binary value, 0x is used to prefix a hexadecimal value. Values without are prefix are decimal.

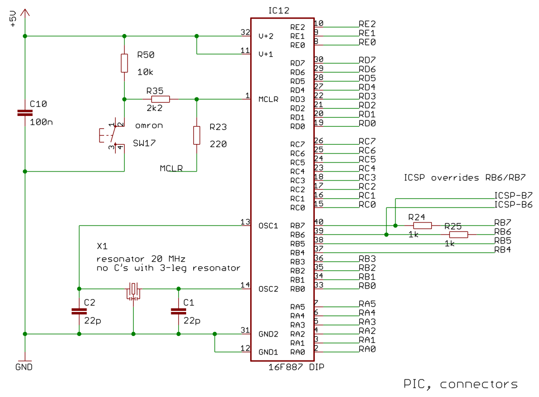
2 Circuit description and interfacing

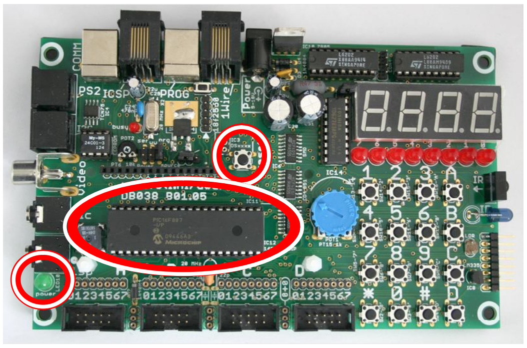
The DB038 circuit is rather complex, so it is discussed piece by piece. The full diagrams can be found in the section Circuit diagrams (P 69). Part of this complexity is caused by the fact that the 40-pins PIC chip does not have enough pins to dedicate separate pins to each peripheral, so only the peripherals that need dedicated PIC pins are connected directly. Other peripherals are interfaced via three CD4051 analog multiplexer chips. Multiplexing is common in real-world applications because the housing (number of pins) of a microcontroller is one of the important factors that determine its price. The DB038 board has much more peripherals than most real-world products; hence it needs a lot of multiplexing.

The default target chip for the DB038 is the 16F887, because this is at the moment the cheapest of the 40-pin 8K instructions flash PICs in the 14-bit core series. Other pin-compatible PICs can be used as well, but the programming details might be a bit different. Examples of alternative chips from the 14-bit core series are the older but well-know 16F877 / 16F877A, and the 16F917. These are all 8K code chips. The 16K instructions 16F1939 is an enhanced-14-bit core. From the 16-bit core series the 18F4520 or the older 18F452 (both 16K instructions) or the newer 18F4620 (32K instructions) could be used.

2.1 *PIC16F887 target chip*

2.1.1 Circuit





The central component is of course a PIC 16F887 chip. This is the target chip that can be programmed by the user, and this chip controls all peripherals. The default clock source for the PIC is a 20 MHz resonator, but the PCB can instead accommodate a crystal and its two accompanying capacitors (the capacitors are not fitted when a resonator is used).

The PICs reset circuit contains a reset switch and a few resistors that prevent damage when the reset switch would be activated while the chip is being programmed. The reset switch is located above the PIC, at the right edge of the LCD connector. Resistor R50 pulls the MCLR line high. This is not needed for a 16F887 (which has an internal MCLR pull-up), but it is needed for some other PIC chips. The RB6 and RB7 I/O pins are used to program the chip. The peripherals and connectors are separated from these two lines of the PIC and the programming circuit by 1kΩ series resistors, to guarantee the programming function. The RB4 and RB5 pins are pulled down by the resistor network of the video interface (p. 29) to prevent enabling of the LVP function, which could cause trouble for the programming function on some PIC chips.

The green LED in the lower-left corner of the board lights up when the target circuit is powered.

2.1.2 16F887 pin usage

The next table shows how the I/O pins of the 16F887 chip are used on the board and the limitations for external use of these pins. In general all pins of the 16F887 can be used externally, provided that no on-board peripherals or connectors are used. The exception is RA4, which is not available on the Dwarf Board connectors or wire cups (it is available on the Dallas 1-wire connector). Beware that using the I/O pins in your own way might cause the on-board peripherals to act funny.

The pins RA0, RA1, RA2 are multiplexed. Their function is set by the value on the pins RE0, RE1, RE2. The jumper COM28 (to the left of the seven-segment displays) can be removed to disable the multiplexers and hence free the RA0, RA1, RA2 pins for other uses. RA4 connects to the on-board socket and the RJ connector, both for Dallas one-wire chips. RA5 connects to the E line of the LCD.

Pins RB6 and RB7 are connected to the on-board programmer and the pins RB4 and RB5 are used to create a composite video output.

Port C connects to the two H-bridge (motor driver) chips (RC0 .. RC5), and to the FT232RL USB-to-serial interface chip.

Port D serves three purposes:

- When RA2 and RE0, RE1, and RE2 enable the LEDs or one of the seven-segment displays, a low value on one of the RD pins enables the corresponding LED or segment.
- A high value on one of RD0, RD1, RD2, and RD3 actives one of the rows of the keypad. RE0, RE1, and RE2 are used to select one of the columns, and RA1 is used to read whether the button at the intersection is pressed.
- RD3 connects to the RS line of the optional LCD, RD4... RD7 connect to the D4... D7 lines. These lines do not influence the LCD unless line E is toggled. Line E connects to pin RA5, which is dedicated to this purpose.

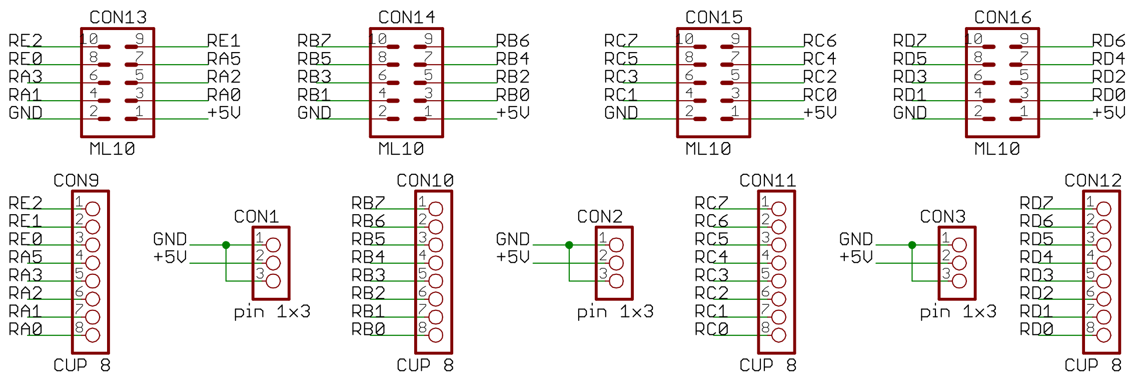
A number of pins (RB4, RB5, and RC7) are driven by on-board peripherals via a resistor. These pins can be used externally, but when used as inputs the external circuit must drive the pin from a sufficiently low impedance to overrule the on-board peripheral. When used as output the PIC will easily supply this drive, but the on-board peripheral might cause some extra current to be drawn.

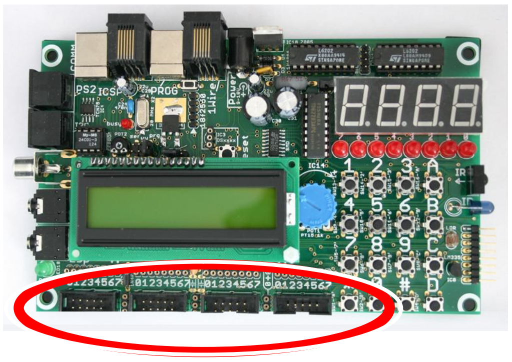
Otherwise the table should be self-explaining.

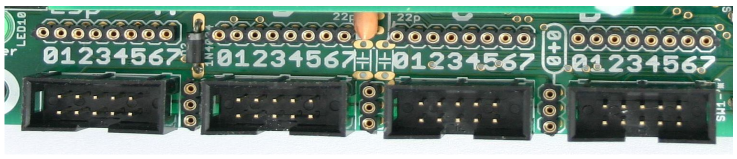
16F887 I/O pin	DB038 use			External use
RA0		RE0 .. RE1:		Depending on the value of RE0… RE2 various on-board peripherals can drive pin RA0, RA1, and RA2. Better not use these three pins externally.
	Input and open-drain output	0b000 : F0	PS/2 – SDA	
		0b001 : F1	I2C – SDA	
	Analog input	0b010 : F2	Microphone input	
	Analog input	0b011 : F3	Battery voltage	
	Analog input	0b100 : F4	Potentiometer	
	Analog input	0b101 : F5	LDR	
	Analog input	0b110 : F6	LM335 temperature sensor	
	Analog input	0b111 : F7	TL431 2.50 Volt analog reference	
RA1		RE0 .. RE1:		
	Input and open-drain output	0b000 : G0	PS/2- SCL	
		0b001 : G1	I2C – SCL	
	Output	0b010 : G2	Loudspeaker output	
	-	0b011 : G3	Not used	
	Digital input	0b100 : G4	Read keypad column 1 (leftmost)	
	Digital input	0b101 : G5	Read keypad column 2	
	Digital input	0b110 : G6	Read keypad column 3	
	Digital input	0b111 : G7	Read keypad column 4 (rightmost)	
RA2		RE0 .. RE1:		
	Output	0b000 : H0	High to enable digit 4 (leftmost)	
	Output	0b001 : H1	High to enable digit 3	
	Output	0b010 : H2	High to enable digit 2	
	Output	0b011 : H3	High to enable digit 1 (rightmost)	
	Output	0b100 : H4	High to enable LEDs	
	Output	0b101 : H5	High to enable IR LED	
	Digital input	0b110 : H6	TSOP1737 IR receiver	
	-	0b111 : H7	Not used	
RA3				OK
RA4	Output	Dallas one-wire bus		OK (open drain!)
RA5	Output	E line of the LCD		The LCD might show garbage; otherwise OK
RA6	These pins are not available as I/O because their primary function (Xtal) is used			
RA7				
RB0	Pins RB0 .. RB7 default to analog			OK
RB1				OK
RB2				OK
RB3	Output	LCD backlight enable (high enables)		OK
RB4	Output	Composite video output; also serves as weak pull-down to prevent inadvertent activation of LVP		For chips with LVP function: the LVP pin must not be driven high during reset; these pins have a
RB5	Output			

16F887 I/O pin	DB038 use				External use
					weak pull-down as part of the composite video circuit; otherwise OK
RB6	Output	HVP Programming interface pins, will be driven by the on-board programmer during programming; 1k series resistors between the pins and the I/O connectors to ensure programmability.			Will be driven by the on-board programmer during programming; otherwise OK
RB7	Output				
RC0	Output	Motor driver			The motor driver might be activated; no problem if the motor driver is not used
RC1	Output				
RC2	Output				
RC3	Output				
RC4	Output				
RC5	Output				
RC6	Output	Asynchronous serial output to FT232RL (second USB)			OK
RC7	Input	Asynchronous serial input from FT232RL (second USB)			FT232RL drives this line via 1kΩ resistor, otherwise OK
RD0	Output	Activate individual LEDs and segments of the selected seven-segment display (active low)	Activate keypad row 1 (lowest)		LEDs, seven-segment display (and LCD when RA5 is also used) might be enabled and show garbage; otherwise OK
RD1	Output		Activate keypad row 2		
RD2	Output		Activate keypad row 3		
RD3	Output		Activate keypad row 4 (highest)	RS line of the LCD	
RD4	Output			D4 .. D7 lines of the LCD	
RD5	Output				
RD6	Output				
RD7	Output				
RE0	Output	Address for the three CD4051 multiplexers: determines sub-function of RA0 ... RA2			OK when the multiplexers are disabled (jumper).
RE1	Output				
RE2	Output				
RE3	This pin is not available as I/O because its primary function (MCLR) is used				

2.2 *Dwarf Bus connectors and wire cups*







Most I/O pins of the chip are (besides being used for the peripherals) also available on four Dwarf Bus (ML10) connectors and four wire-cup strips. The Dwarf Bus connectors can be used to connect Dwarf Board peripherals. The four Dwarf Busses are available on the ML10 connectors A...D. Note that A contains the PIC pins RA0 ... RA5 and RE0 ... RE2. Pin RA4 is not connected to any Dwarf Bus connector (but it is available on the Dallas 1-Wire connector). The tables below show the correspondence between Dwarf Bus pins (and wire cup connectors) and PICmicro controller pins. The green entries show the irregularities.

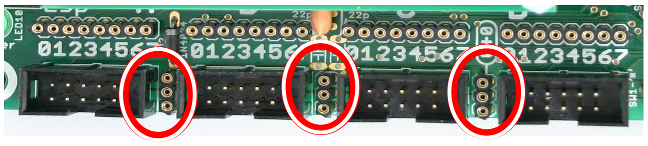
Dwarf Bus BUS_A	PICmicro port
pin_0	RA0
pin_1	RA1
pin_2	RA2
pin_3	RA3
pin_4	RA5
pin_5	RE0
pin_6	RE1
pin_7	RE2

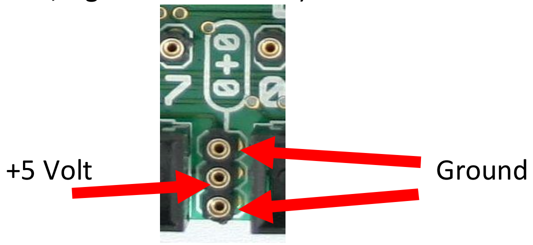
Dwarf Bus BUS_B	PICmicro port
pin_0	RB0
pin_1	RB1
pin_2	RB2
pin_3	RB3
pin_4	RB4
pin_5	RB5
pin_6	RB6
pin_7	RB7

Dwarf Bus BUS_C	PICmicro port
pin_0	RC0
pin_1	RC1
pin_2	RC2
pin_3	RC3
pin_4	RC4
pin_5	RC5
pin_6	RC6
pin_7	RC7

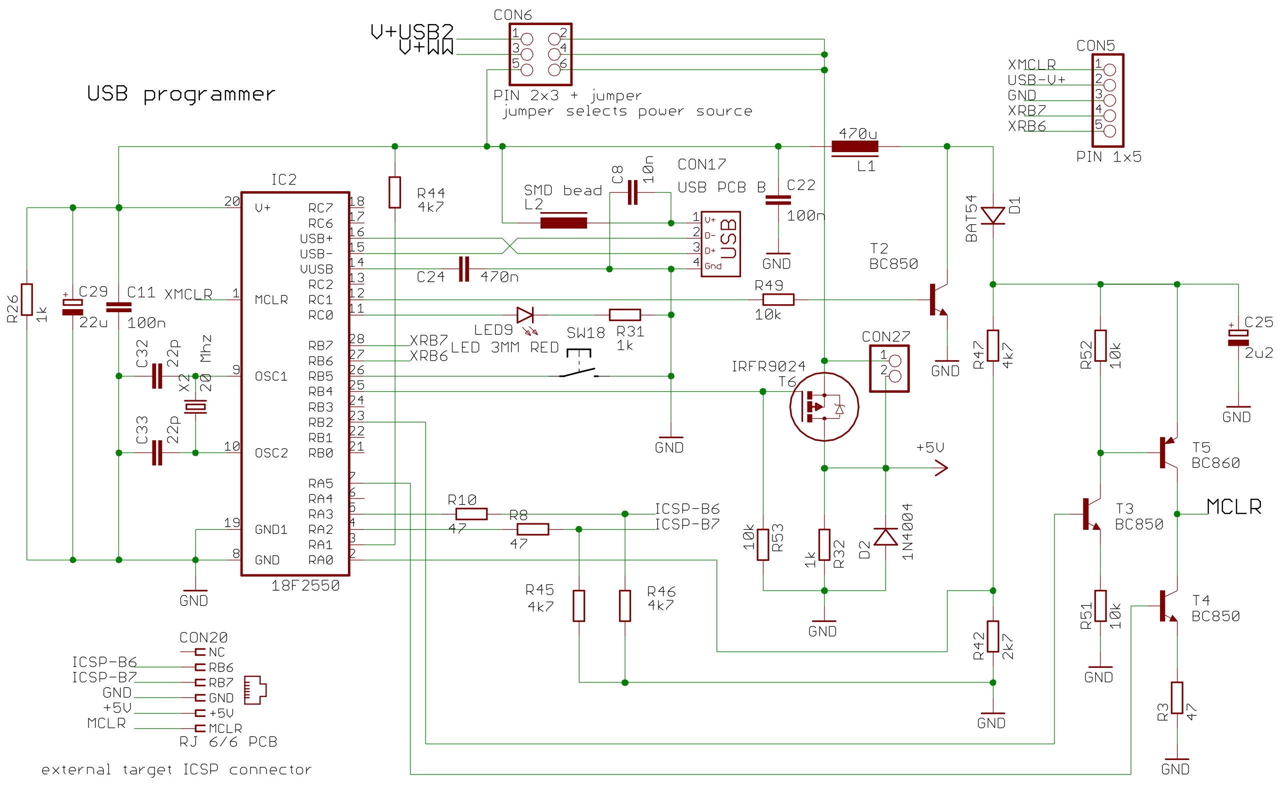
Dwarf Bus BUS_D	PICmicro port
pin_0	RD0
pin_1	RD1
pin_2	RD2
pin_3	RD3
pin_4	RD4
pin_5	RD5
pin_6	RD6
pin_7	RD7

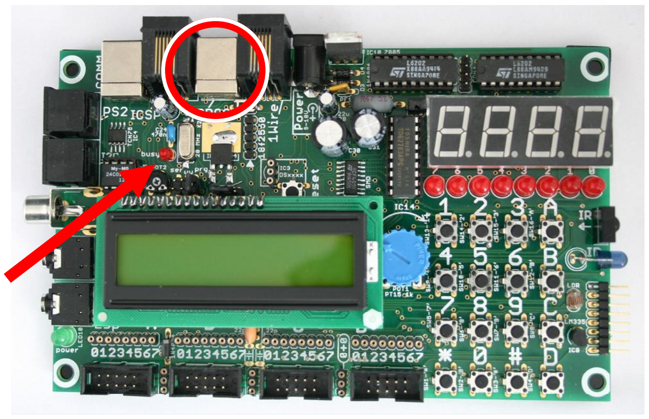
The ground and +5V lines are also available on three 3-pin wire-cup strips (middle cup is 5 Volt, top and bottom cups are ground, like on a coax cable: ground outside, signal in the centre).

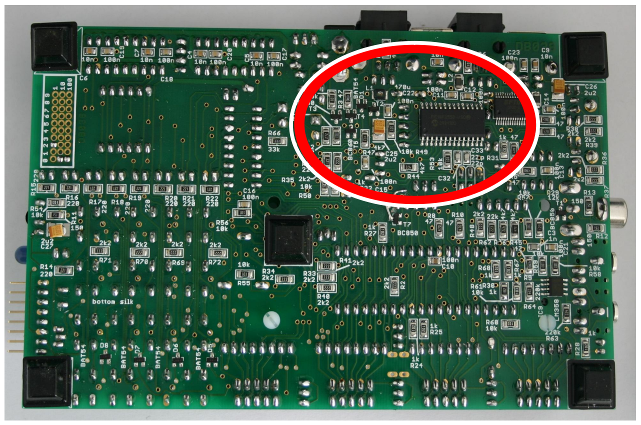




2.3 Programmer





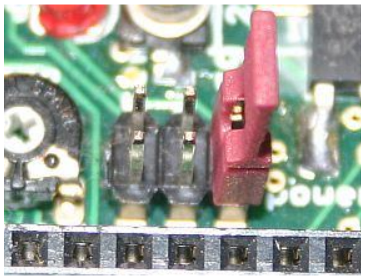


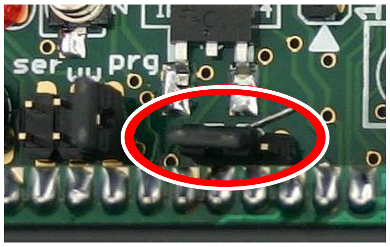
The on-board programmer uses the second USB connector (counted from the left). Most of this circuit is on the bottom (solder side) of the PCB. It is a simplified functional equivalent of the Microchip PICkit2 programmer. The main simplification is that it does 5V operation only, it does not attempt to vary the voltage to the target chip.

The PIC 18F2550 handles all aspects of the USB communication with the user PC and the programming of the target chip. The IRF9024 MOSFET switches the power to the target PIC and its peripherals. The yellow LED13 lights up when the programmer is active (is busy programming the target chip). When the small pushbutton switch below the USB connector is pressed while the USB connection is made the 18F2550 chip will enter bootloader mode, so the user can download a different firmware. The DB038 board works

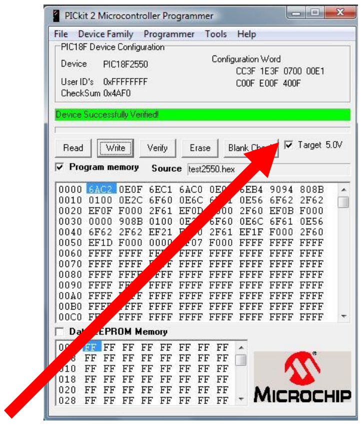
OK with pickit2 firmware version 1.20 and the pickit2 PC application version 1.20. Firmware and PC application version 1.21 do not work OK with this board. Other versions have not been tested.

The circuit around transistor T2 (L2, D1, C15) is a simple step-up voltage converter that generates the Vpp (programming enable voltage) for the 16F887 target PIC. The PIC 18F2550 is the controlling element in this converter. R15 and R16 allow the PIC to sense the generated voltage, so the chip can adjust its control of T2 accordingly. The voltage on C15 will be approximately 13 Volt. The 18F2550 drives the transistors T3, T4 and T5 to switch the target 16F887's MCLR pin to either the generated Vpp voltage on C15, or to ground, or to let the pin float.


Programmer USB interface is selected as power source.



The power enable jumper is in the active position (left two pins).


PICkit 2 Microcontroller Programmer
File Device Family Programmer Tools Help
Read Write Verify Erase Blank Check
MICROCHIP

PIC18F Device Configuration

Device PIC18F2550
User ID's 0xFFFFFFFF
CheckSum 0x4AF0

Configuration Word
CC3F 1E3F 0700 00E1
C00F E00F 400F

Device Successfully Verified!

☑ Target 5.0V

☑ Program memory Source test2550.hex

```
0000 6AC2 0E0F 6EC1 6AC0 0E[illegible] [illegible]EB4 9094 808B
0010 0100 0E2C 6F60 0E6C [illegible] 0E56 6F62 2F62
0020 EF0F F000 2F61 EF0D [illegible]00 2F60 EF0B F000
0030 0000 908B 0100 0E[illegible] 6F60 0E6C 6F61 0E56
0040 6F62 2F62 EF21 [illegible] 2F61 EF1F F000 2F60
0050 EF1D F000 0000 [illegible]07 F000 FFFF FFFF FFFF
0060 FFFF FFFF FF[illegible] FFFF FFFF FFFF FFFF FFFF
0070 FFFF FFFF [illegible]F FFFF FFFF FFFF FFFF FFFF
0080 FFFF FFF[illegible]FFF FFFF FFFF FFFF FFFF FFFF
0090 FFFF FF[illegible] FFFF FFFF FFFF FFFF FFFF FFFF
00A0 FFFF [illegible]F FFFF FFFF FFFF FFFF FFFF FFFF
00B0 FFF[illegible]FFF FFFF FFFF FFFF FFFF FFFF FFFF
00C0 FF[illegible] FFFF FFFF FFFF FFFF FFFF FFFF FFFF
```

☐ Da[illegible]EEPROM Memory

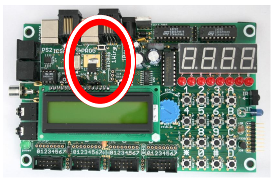
```
0[illegible] FF FF FF FF FF FF FF FF
[illegible]8 FF FF FF FF FF FF FF FF
[illegible]10 FF FF FF FF FF FF FF FF
018 FF FF FF FF FF FF FF FF
020 FF FF FF FF FF FF FF FF
028 FF FF FF FF FF FF FF FF
```

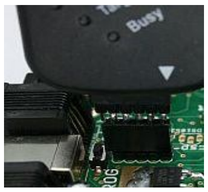
When the power enable jumper is not active this checkbox must be enabled to let the programmer hardware supply power to the rest of the board.

The pin header (directly below the small red LED) allows the user to select the main power source for the board. In normal use (program development) mode this will most likely be the USB programming connection to the PC (jumper in the rightmost position). When the USB programming connection is used to power the board the programmer will initially disable the power to the target circuit. The pickit2 PC application must be used to enable the target power. Other power options are the second (left) USB connector (USB virtual serial port), or an external Wall-Wart or battery.

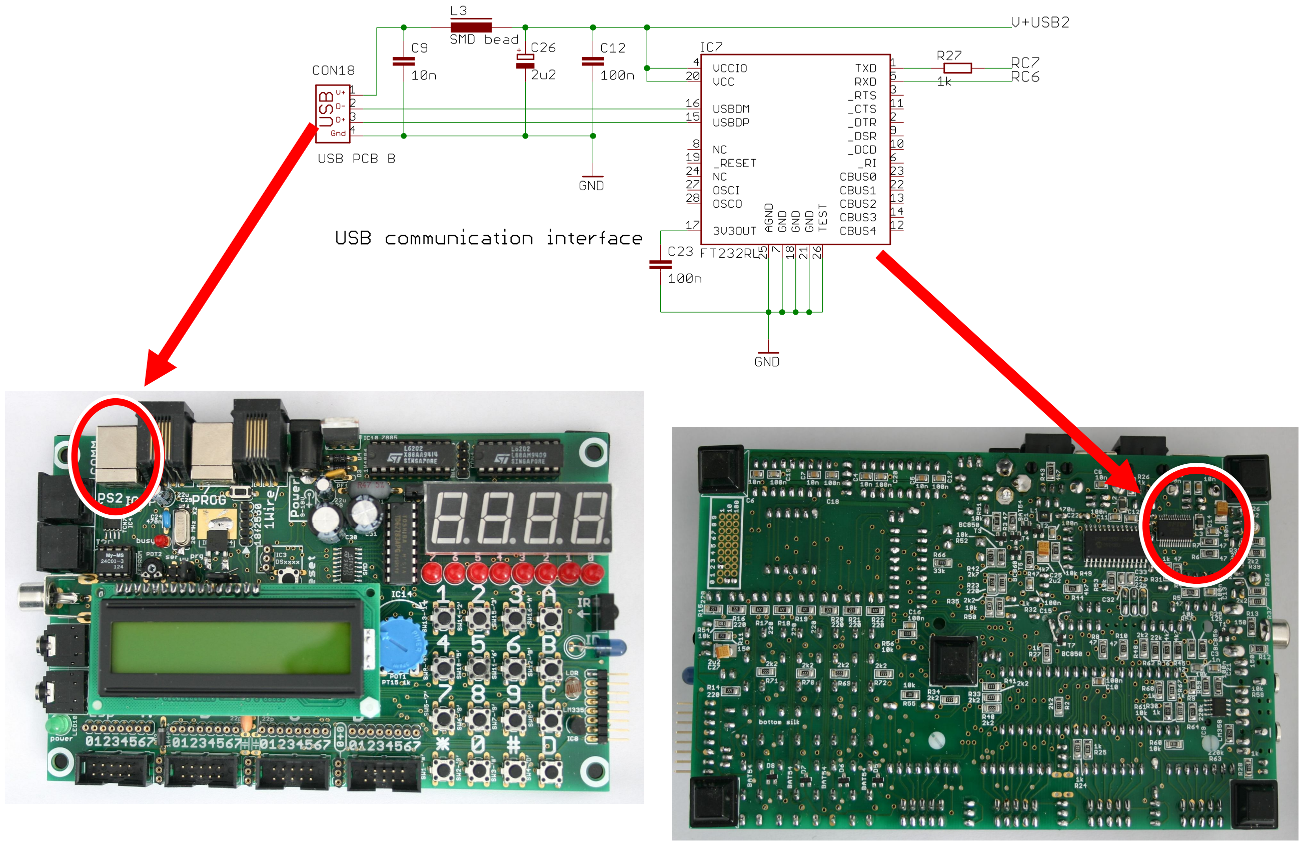
The programming signals are fed to the target 16F887 chip, but are also available on an RJ connector. This makes it possible to use the DB038 to program an external chip. This requires that the on-board target chip is removed. The pinout of the RJ connector is compatible with the Microchip ICD2.

The 18F2550 chip that is the heart of the programmer must itself be programmed. This is of course done before the board is shipped, but the chip can be re-programmed using an ICSP pin-header. This header is located between the second RJ connector and the LCD connector. The pinout is compatible with a Microchip PICkit2 programmer, but one pin (which has no function in the programming process) is omitted, and an 'extender' connector is required to make the connection. The pictures below show how this connection is to be made.





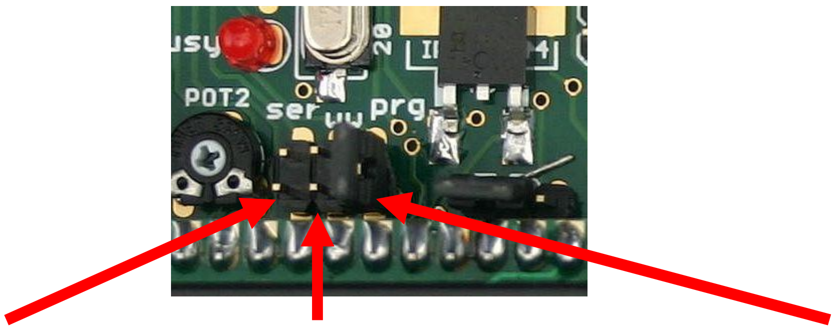
2.4 USB to asynchronous serial interface



A second USB port is provided that connects to an FT232RL chip. This chip converts the USB to asynchronous serial, for direct connection to the PIC16F887's UART pins. At the PC side the drivers provided by the manufacturer (http://www.ftdichip.com) can emulate a Windows serial port (convenient: the Windows application can use the port like any other serial port), or provide a DLL interface (which might be less convenient, but offers higher throughput). On the FTDI website the Windows serial port driver is called VCP driver (Virtual Communication Port).

If the board is to be used with only the second USB connected it should be selected as the power source. The 16F887's UART pins are connected only to the FTDI232RL chip. The other interface pins of the FT232RL chip are not connected. The 1kΩ series resistor in the TxD line makes it possible for an external signal to override the FT232RL’s output without doing any harm to the chip.

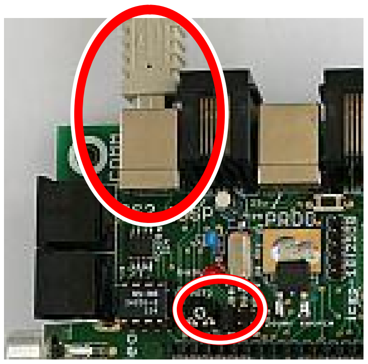
2.5 Power source selection jumper

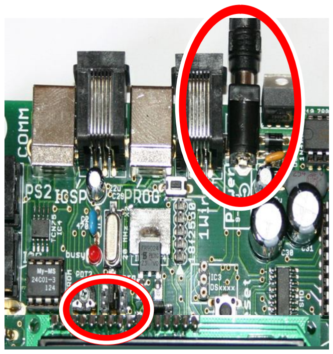


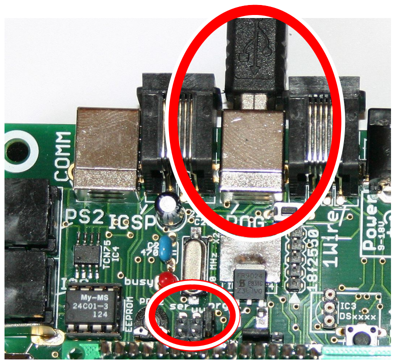
Powered from serial USB interface (jumper in left position)

Powered from wall-wart / battery connector (jumper in middle position)

Powered from USB programming interface (jumper in right position)



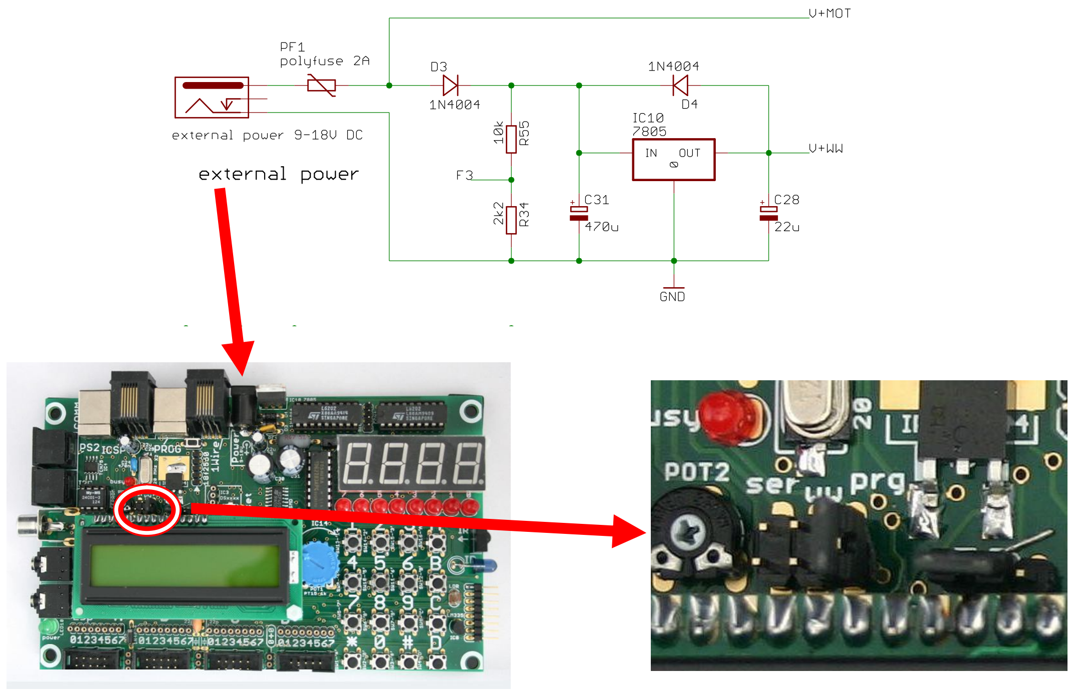




The USB interface chips are always powered from their own USB connector. A jumper selects the power source for the rest of the board (this connector is part of the programmer circuit, p 11). One exception: the motor interface can only be powered by the external battery / wall-wart connector, because its current and voltage requirements exceed what is commonly available from a USB port. Main power can be provided by the first USB port (programming interface), the second USB port (serial communication interface), or from a wall-wart or a batter. A battery or wall-wart must be 9...18V DC, 2.5mm plug; centre is positive, 200mA for the board itself, plus current for any peripherals.

When the USB programmer is not connected, power from the serial USB interface or from the wall-wart / battery connector (as selected by the jumper) will be available for the board. However, when the USB programmer connector is connected, the programmer will at startup disable the power to the board, until power is re-enabled using the PC application.

2.6 *External power*



The Wall-Wart / battery power circuit provides power to the board when the power jumper is set for this source. The connector is a 2.5mm plug, centre pin positive. The external power source should provide 9…18V DC, at 200mA for the board itself, plus current for any peripherals. When motors are used these will likely require a specific voltage, and dominate the current requirement. The external power is provided directly to the motor driver circuit. The external power source must match the voltage required by the motors, and it must be able to provide the current required by the motors.

Polyfuse PF1 protects against potentially dangerous situations when for instance a high-current battery is used and the motors draw excessive amounts of current. Regulator IC10 provides a stable 5V power for most of the board. Diode D3 protects the regulator when power is applied in reverse. Diode D4 protects the regulator against reverse-current when C28 is charged but no external power is present.

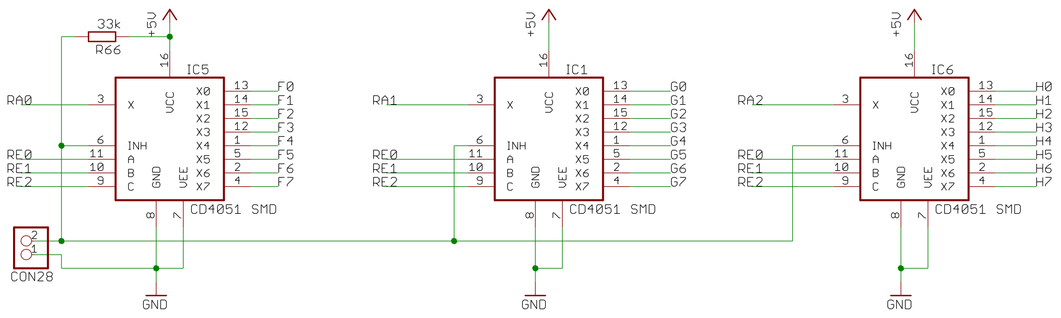
The resistor divider R55 / R34 provides (via a CD4051 multiplexer) a fixed part of the external voltage to a 16F887 A/D input. Hence the chip can (with some calculation) determine the external voltage, which might be convenient when the board is used as free-roaming robot, powered by a battery.

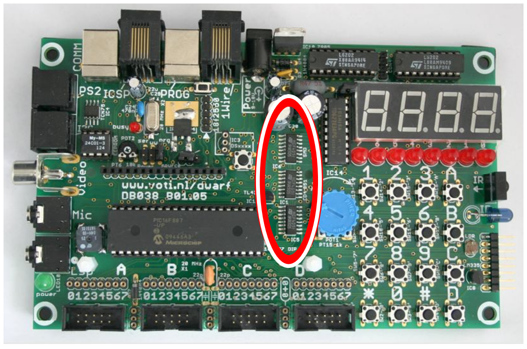
2.6.1 Software interface – external power voltage

The voltage of the battery or wall-wart (depending on what is connected) can be sensed by the 16F887. To use this function pin RA0 must be configured as an input and as an analog input, and pins RE0 … RE2 must be outputs and the value on these pins must be 0b011. The voltage on RA0 is provided by the voltage divider R33 / R23, so it will be 2.2 / 12.2 = 18% of the actual battery or wall-wart voltage. Note that the A/D measurement will use the PICs power as reference, which can be a rather inaccurate +5V. Using the TL431 voltage reference to correct the measurement will yield a much more accurate result.

16F887 pins	Direction	Value
RA0	Input, Analog	Battery or Wall-Wart voltage
RE0 … RE2	Outputs	0b011

2.7 Analog multiplexers

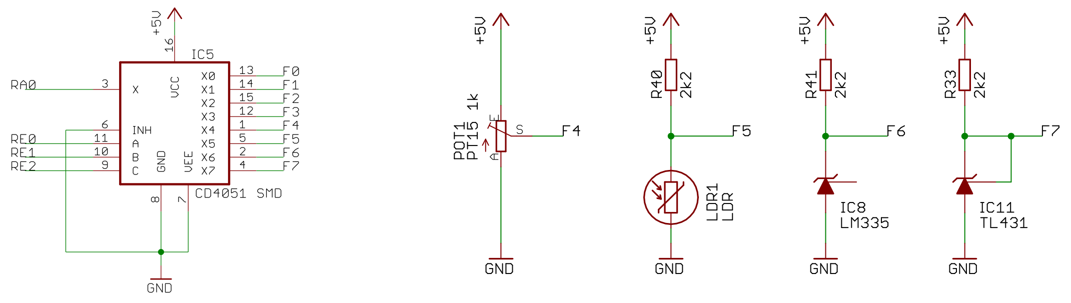




Three CD4051 multiplexer chips connect the three 16F887 pins RA0, RA1 and RA2 each to one of three sets of function lines, F0…F7, G0…G7 and H0...H7. The selection, which is the same for all three multiplexers, is determined by the 3-bit value set by the 16F887 on its pins RE0, RE1, and RE2. This allows the PIC to interface to 24 functions using only 6 of its I/O pins. On pin RA1 a weak pull-down is provided by resistor R36. This makes it possible to read the keypad (which can pull a line high, but not low). The table below summarizes the functions that are accessed using the multiplexors. Note that changing the value on the RE0..RE2 pins can produce unintended intermediate values. Hence it is advised to have the RA0..RA2 pins configured as inputs whenever the value on RE0..RE1 is changed.

Selector value			Function of 18F887 pin		
RE2	RE1	RE0	RA0	RA1	RA2
0	0	0	PS/2 connector - SCL	PS/2 connector - SDA	High to enable digit 1
0	0	1	I2C connector - SCL	I2C connector – SDA	High to enable digit 2
0	1	0	Microphone input	Loudspeaker output	High to enable digit 3
0	1	1	Battery voltage	Not used	High to enable digit 4
1	0	0	Potentiometer	Read keypad column 1	High to enable LEDs
1	0	1	LDR	Read keypad column 2	High to enable IR LED
1	1	0	LM335 temperature sensor	Read keypad column 3	TSOP IR receiver
1	1	1	TL431 2.50V reference	Read keypad column 4	Not used

2.8 *Simple analog inputs: Potentiometer, LDR, LM335 and TL431*



Multiplexer outputs F4 … F7 are connected to four analog inputs:

- a potentiometer,
- an LDR,
- an LM335 linear temperature sensor,
- a TL431 voltage reference.

The potentiometer has a blue thumbwheel. Turning the wheel clockwise increases the voltage. The potentiometer also functions as a load to the 5V power, guaranteeing a quick discharge when the power is removed (this is important when an USB cable is removed and quickly reinserted).

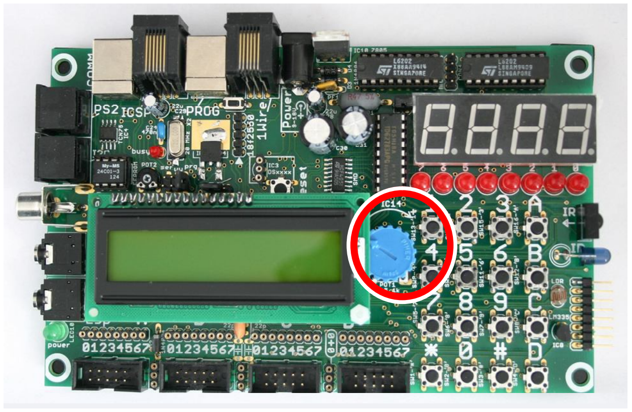
The LDR (light dependent resistor) is located at the right edge of the board. It is connected as voltage divider: low light level = high voltage, high light level = low voltage.

The LM335 temperature sensor is located at the right edge of the board, immediately above the LDR. It provides a voltage that is linear with the absolute temperature. The voltage is 10 mV * the temperature (in degrees Kelvin), or approximately 3 Volt at 25 degrees Celsius. The resolution of the 10-bit A/D converter with the 5V supply as reference is ~ 5 mV, so the temperature can be read with a resolution of ~ 0.5 degree. Note: resolution, not accuracy. The inaccuracy of the 5V supply will give a large error. Note that the board has another temperature sensor (TCN75 or TCN75A) that has a digital (I2C) rather than an analog interface.

The TL431 voltage reference provides a stable and accurate (1% maximum error) 2.5 Volt reference. It can be used to correct the inaccuracies of A/D conversions caused by using the (approximately) 5V power as reference. This chip is located in the centre of the board, to the left of the potentiometer (the blue thumbwheel).

2.8.1 Software interface - potentiometer

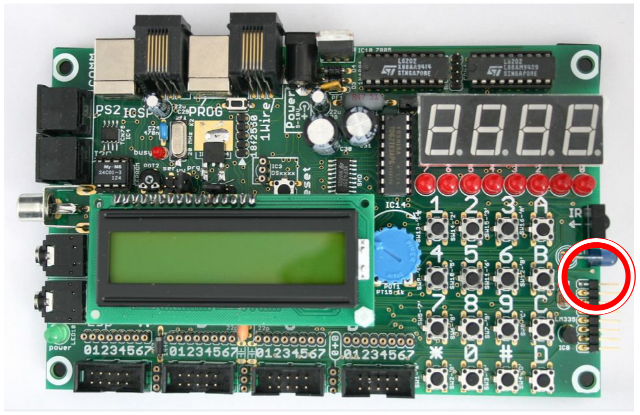
To read the on-board 1kΩ trim potentiometer pin RA0 must be configured as an analog input, and pins RE0 … RE2 must be outputs and the value on these pins must be 0b100. The voltage provided by the potentiometer is radiometric: it is a percentage of the 5V power supply, so when it is measured by the 16F887 using the power supply as reference the actual value of the power supply cancels out.



16F887 pins	Direction	Value
RA0	Input, Analog	Analog value 0 … 5 Volt (nominal), proportional to the setting of the trim potentiometer and the 5V power (radiometric)
RE0 … RE2	Outputs	0b100

2.8.2 Software interface - LM335 analog temperature sensor

The on-board LM335 analog temperature sensor can be used to provide an analog value to the 16F887 pin RA0 that is proportional to the absolute value (10 mV per degree Kelvin). To use this function pin RA0 must be configured as an input and as an analog input, and pins RE0 … RE2 must be outputs and the value on these pins must be 0b110. The analog voltage is absolute: it does not depend on the 5V power supply, so when it is measured by the 16F887 using the power supply as reference the accuracy of this 5V power supply (~5%) has a large influence on the measured value. To compensate for this the LM431 voltage reference can be read and a compensated temperature can be calculated in software.

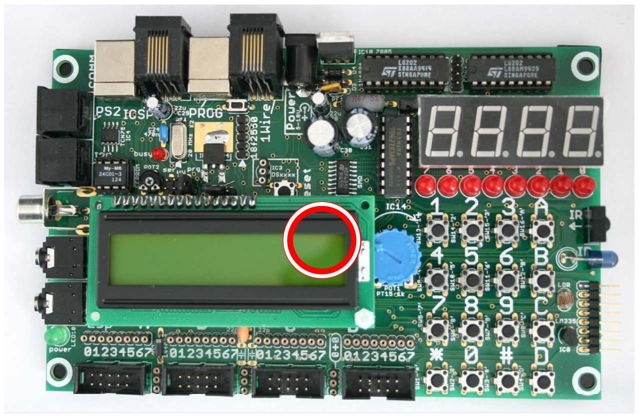


16F887 pins	Direction	Value
RA0	Input, Analog	Analog value of 10 mV per degree Kelvin (absolute)
RE0 … RE2	Outputs	0b110

2.8.3 Software interface - TL431 voltage reference

The on-board TL431 voltage reference chip can be used to provide a stable and accurate (1%) 2.500 Volt to the 16F887 pin RA0. To use this function pin RA0 must be configured as an input and as an analog input, and pins RE0 … RE2 must be outputs and the value on these pins must be 0b101.

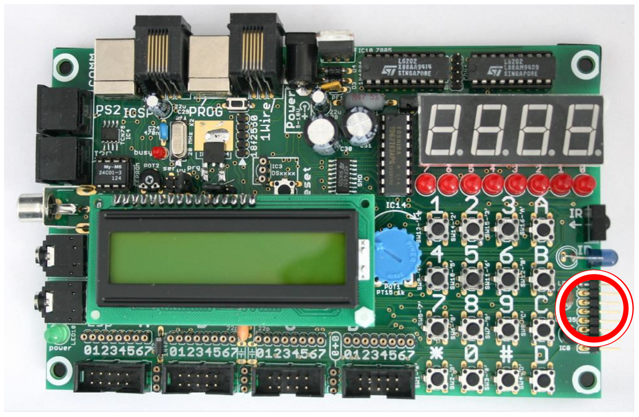
The A/D converter of the PIC uses its +5V power as reference for the conversion. It is my experience that the +5V as provided by an USB connector can vary quite a bit, even within let's say one second. Hence when the TL431 is used as a reference to correct an A/D measurement to an absolute value both measurements must be done in quick succession.



16F887 pins	Direction	Value
RA0	Input, Analog	Analog level of 2.5 Volt (absolute)
RE0 … RE2	Outputs	0b111

2.8.4 Software interface - LDR

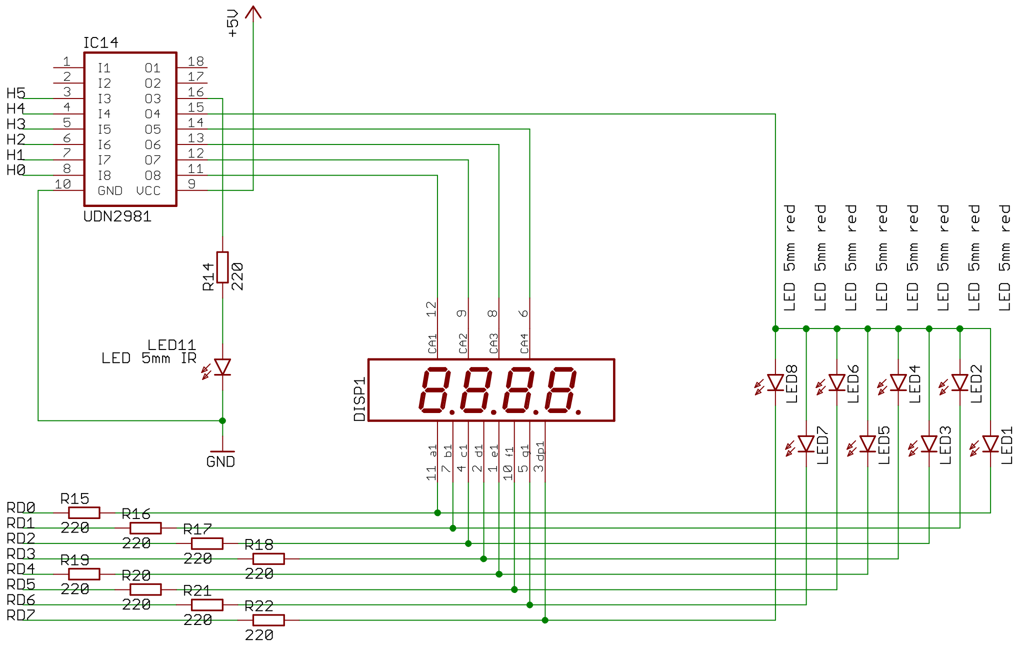
The on-board LDR (Light Dependent Resistor = visible light sensor) can be used to provide an analog value to the 16F887 pin RA0. This level will be higher in darkness and lower when the LDR is illuminated. To use this function pin RA0 must be configured as an input and as an analog input, and pins RE0 … RE2 must be outputs and the value on these pins must be 0b101. This analog voltage is radiometric: it is a percentage of the 5V power supply, so when it is measured by the 16F887 using the power supply as reference the actual value of the power supply cancels out.

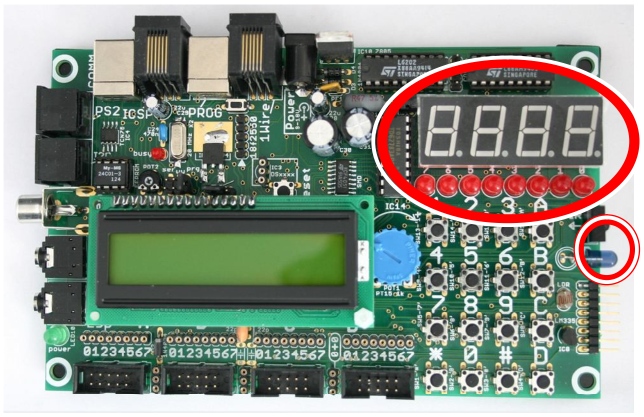


16F887 pins	Direction	Value
RA0	Input, Analog	Analog value, lower with more ambient light
RE0 … RE2	Outputs	0b101

2.9 *Seven-segment displays and LEDs*

2.9.1 Circuit





The multiplexer outputs H0..H5 are connected to an ULN2981 high-side driver. When an input of this driver is made high the corresponding output will be high too. (The reverse is not true: if the input is low the output will not be low, it will effectively be disconnected. An ULN2981 can only source current, not sink.) One output of the ULN2981 drives an IR LED. The other outputs drive the common sides of four seven-segment displays and a set of 8 LEDs. The low side of these displays and LEDs are (via 220 Ω series resistors) driven directly by the 16887 output pins RD0..RD7. The contrast of the seven segment displays can be improved by putting a yellow sheet of semi-transparent plastic on top of them.

2.9.2 Software interface – IR LED

The IR LED is activated by making pin RA2 an output and high, pins RE0..RE2 outputs and the value on these pins 0b011. Once this value has been set on pins RE0…RE2 the IR LED can be turned off and on by making pin RA2 low (LED off) and high (LED on).

16F887 pins	Direction	Value
RA2	Output	High ➔ LED on; Low ➔ LED off
RE0 … RE2	Outputs	0b101

2.9.3 Software interface – Eight LEDs

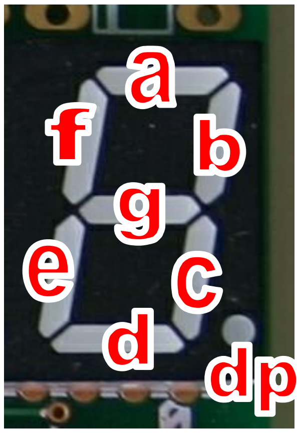
The eight LEDs are activated by making pin RA2 an output and high, pins RE0..RE2 outputs and the value on these pins 0b100, and making port D an output and each pin low for a corresponding LED on and high for a corresponding LED out. The rightmost LED corresponds to the lowest (least significant) bit.

16F887 pins	Direction	Value
RA2	Output	0b1
RD0 … RD7	Outputs	Each pin corresponds to one LED, low for LED on, high for LED off. RD0 corresponds to the rightmost LED.
RE0 … RE2	Outputs	0b100

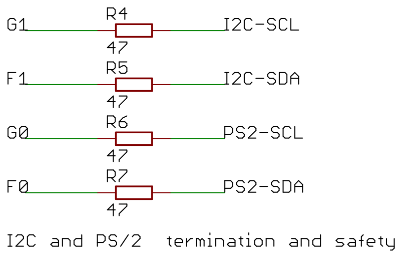
2.9.4 Software interface - Seven segment displays

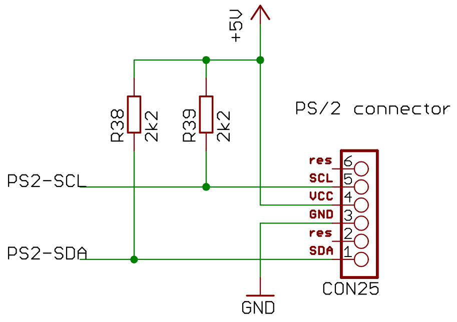
One of the four seven segment displays is activated by making pin RA2 an output and high, pins RE0..RE2 outputs and the value on these pins 0b000 … 0b011, and making port D an output and each pin low for a corresponding segment on and high for a corresponding segment off. Segment A corresponds to the lowest bit of port D (bit 0), segment G corresponds to bit 6, the decimal point to bit 8.

16F887 pins	Direction	Value	
RA2	Output	0b1	
RD0 … RD7	Outputs	Bit 0	Low enables segment A
		Bit 1	Low enables segment B
		Bit 2	Low enables segment C
		Bit 3	Low enables segment D
		Bit 4	Low enables segment E
		Bit 5	Low enables segment F
		Bit 6	Low enables segment G
		Bit 7	Low enables decimal point
RE0 … RE2	Outputs	0b000	Enables digit 3 (leftmost)
		0b001	Enables digit 2
		0b010	Enables digit 1
		0b011	Enables digit 0 (rightmost)

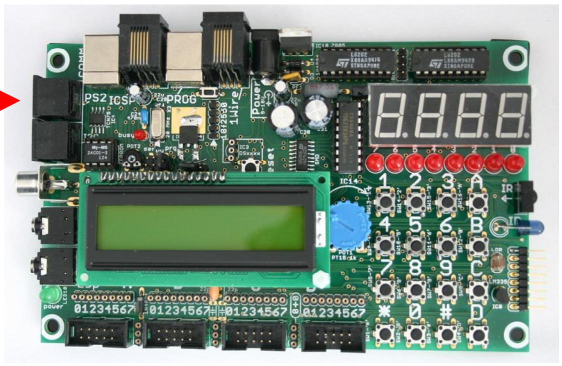


2.10 PS/2 connector



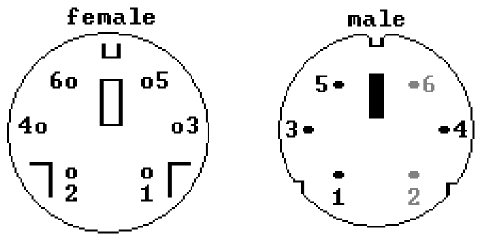


PS/2 connector



Outputs of two multiplexers are used as PS/2 lines. 47 Ω series resistors provide a (minimal) protection against hardware and software errors, and help to damp ringing when longer lines are used. 2.2 kΩ pull-up resistors are provided, as required by a PS/2 interface.

pin	function
1	data
2	(reserved)
3	gnd
4	+5v
5	clock
6	(reserved)



The PS/2 connector accepts a standard miniature DIN PS/2 keyboard or mouse. The connector has pins for ground, +5V, serial data, and serial clock. The PS/2 connector is physically and logically equivalent to the I2C connector, so it could be used as (second) I2C connector (or vice versa).

Note that PS/2 as used here means the mini-din plug as first used by the IBM PS/2 personal computers for keyboard and mouse. It has nothing to do with the PlayStation 2 (often abbreviated as PS2).

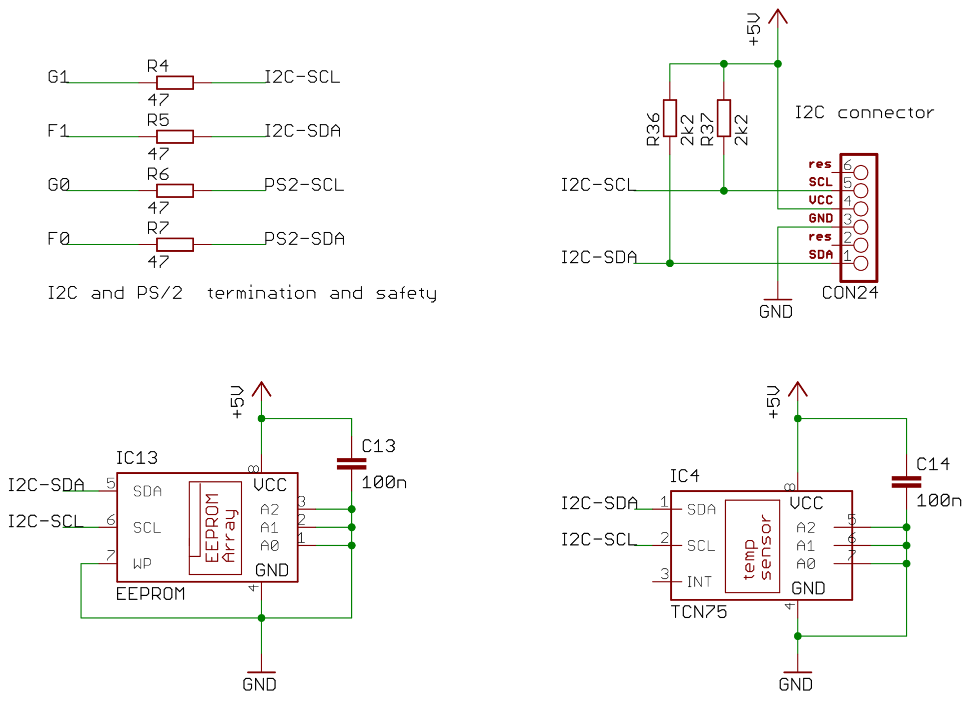
2.10.1 Software interface – PS/2

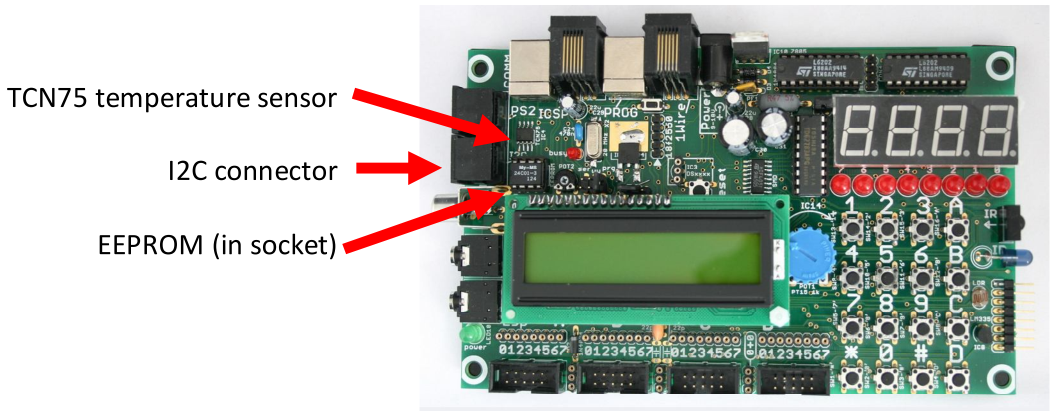
The PS/2 interface is activated by making pins RA0 and RA1 inputs, and pins RE0..RE2 outputs and the value on these pins 0b000. The PS/2 protocol must be implemented in software.

16F887 pins	Direction	Value
RA0	Make these pins digital inputs before the selector (RE0 … RE2) is set to 0b000. Set the	

RA1	output value of RA0 and RA1 to 0 (but the pins are still inputs). Now the PS/2 protocol can be implemented in software by switching between input (and reading the value) and output.	
RE0 … RE2	Outputs	0b000

2.11 I2C: EEPROM (optional), TCN75A, I2C connector

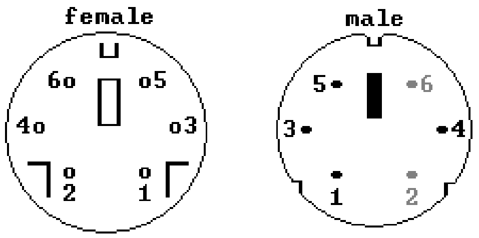




Outputs of two multiplexors are uses as I2C lines. 47 Ω series resistors provide a (minimal) protection against hardware and software errors, and help to damp ringing when longer lines are used. 2.2 kΩ pull-up resistors are provided, as required by an I2C interface.

Note that both the EEPROM and the temperature sensor chip have their address pins tied to ground, so the user-selectable address bits are 0.

Pin	function
1	SDA
2	(reserved)
3	gnd
4	+5v
5	SCL
6	(reserved)



The I2C connector is a PS/2-style miniature DIN connector. Two chips can be fitted on the board that connect to the I2C lines: an EEPROM and a TCN75 temperature sensor. The I2C connector is physically and logically equivalent to the PS/2 connector. If the on-board I2C peripherals are left out (or removed) the I2C connector could be used as (second) PS/2 connector.

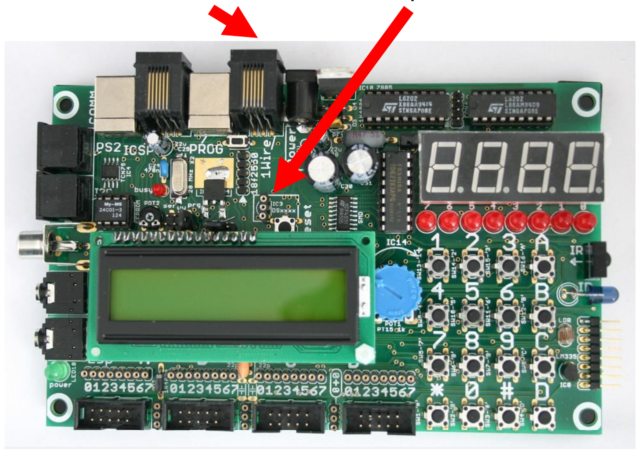
2.11.1 Software interface – I2C

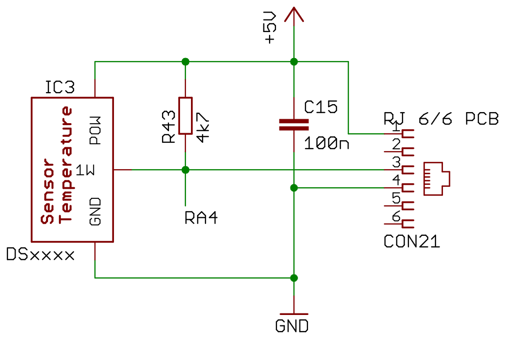
The I2C interface is activated by making pins RA0 and RA1 inputs, and pins RE0..RE2 outputs and the value on these pins 0b001. The 16F887 chip has I2C hardware, but the I2C bus is not connected to the pins that connect to this hardware. Hence the I2C protocol must be implemented in software.

16F887 pins	Direction	Value
RA0 RA1	Make these pins digital inputs before the selector (RE0 … RE2) is set to 0b000. Set the output value of RA0 and RA1 to 0 (but the pins are still inputs). Now the PS/2 protocol can be implemented in software by switching between input (and reading the value) and output.	
RE0 … RE2	Outputs	0b001

2.12 *Dallas one-wire: DS18S20 (optional) and connector*

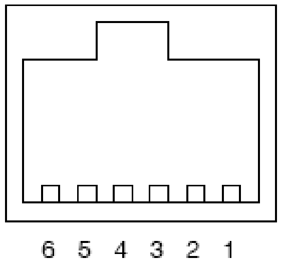
Dallas one-wire connector Optional DS18S20 temperature sensor can be fitted here





A Dallas One Wire interface is provided on an RJ11/RJ12 6/6 connector. The connector has pins for ground, +5V and one-wire data (the pinout of the connector conforms to the de-facto standard for one-wire). One 16F887 pin (RA4) is used (exclusively) for the one-wire bus (this pin is not even present on the ML10 and wire cups). A 4.7 kΩ pull-up resistor is provided. Optionally a DS18B20 temperature sensor can be fitted on the board.

Pin number	function
1	+5V
2	Not connected
3	One-wire data
4	Ground
5	Not connected
6	Not connected



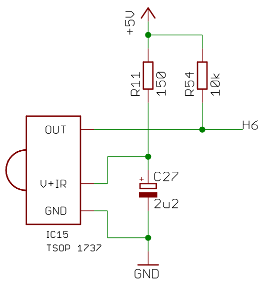
Pin numbers, looking into a female RJ11 / RJ12 connector (as present on the PCB)

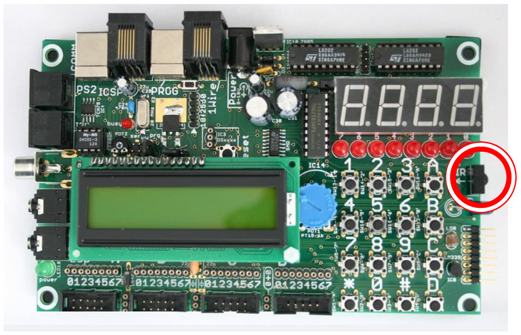
A note on the name RJ11 / RJ12: these names refer to two particular uses of the pins of a 6 position / 6 connection modular jack. Hence both names refer to the same physical plug, but both are strictly speaking used erroneously here, because the one-wire use of this plug is not what either RJ11 or RJ12 describes. But the names RJ11 / RJ12 are what everyone seems to calls these connectors and plugs, so that is how they are called here.

2.12.1 Software interface – Dallas 1-wire

The Dallas 1-wire bus and the optional on-board DS18B20 are connected to the PIC pin RA4. This pin is dedicated to the 1-Wire bus: it has no other function on the DB)37 board. On all PIcs I know this pin is an open-drain output, which is exactly what you need for a 1-Wire bus.

2.13 *IR receiver*





A TSOP1736 IR receiver(the actual chip that is used can vary) is located at the right edge of the board. The output of the receiver is connected to the H6 multiplexer output.

This type of IR receiver is sensitive to an IR signal that is both modulated at a specific frequency and has silent periods (when a continuous modulated signal is present the receiver will consider this background noise and attempt to filter it out). It this case the peak sensitivity is at either 36, which provides a good reception of Philips 36 kHz (RC5 etc), and reasonable reception of the 38 kHz signals used by most other remote controls (Sony etc). The output is high when no signal is received, low when an IR burst is received.

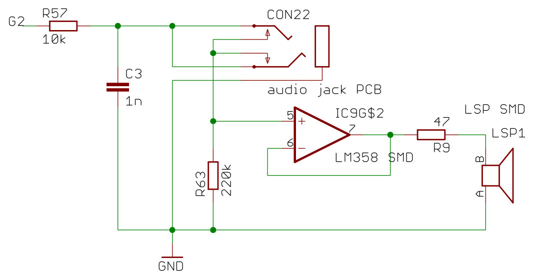
A matching IR LED is mounted just below the IR receiver. This LED is discussed in the section Seven-segment displays and LEDs (p 20).

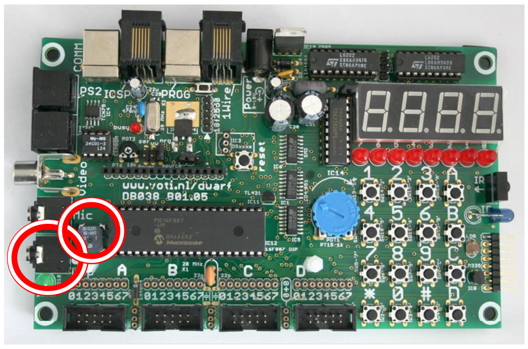
2.13.1 Software interface – IR Receiver

The IR receiver is addressed by making pins RA2 input, and pins RE0..RE2 outputs and the value on these pins 0b110.

16F887 pins	Direction	Value
RA2	Digital input	Low when an IR signal is received, otherwise high.
RE0 … RE2	Outputs	0b110

2.14 *Audio output: LSP and connector*





A small loudspeaker is located to the left of the PIC 16F887. The lowest connector at the left side is an audio output jack connector. When a jack is inserted into the audio output connector the on-board loudspeaker will be disabled. (note: the silkscreen on boards < 1.05 erroneously identified this connector as 'Mic'.)

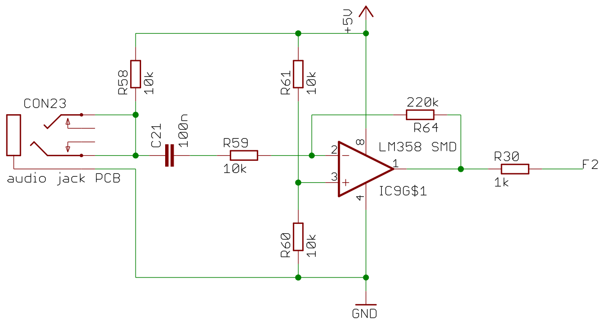
The multiplexer output G2 is filtered by a simple RC filter and fed to the audio out connector. When a sufficiently high PWM frequency is used the RC will filter it out (the -3dB frequency is 16 kHz). Alternatively the PIC's output can be toggled within the audio frequency range to produce a square wave directly. When the audio output connector is not used the output of the RC filter is connected to the OpAmp (half of an LM358 dual OpAmp), its output fed to a small on-board speaker.

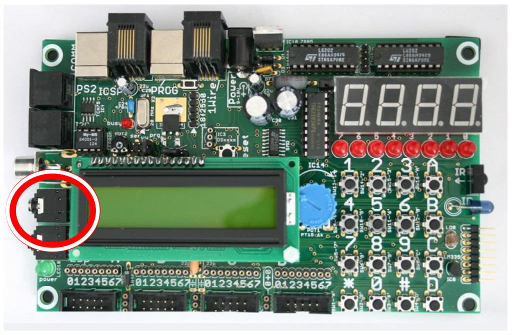
2.14.1 Software interface – LSP

The Loudspeaker receiver is addressed by making pins RE0..RE2 outputs and the value on these pins 0b010, and pin RA1 a digital output.

16F887 pins	Direction	Value
RA1	Digital output	The LSP signal follows this pins, after being filtered with a 16 kHz cutoff frequency.
RE0 … RE2	Outputs	0b010

2.15 *Audio input connector*





The second connector counting up from the bottom left side is a microphone input jack. (note: the silkscreen on boards < 1.05 erroneously identified this connector as ‘Lsp’.)

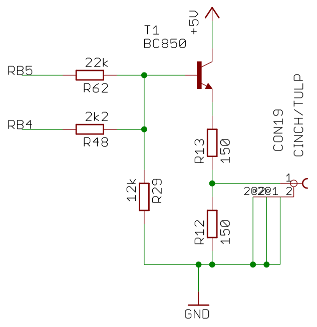
The microphone signal is amplified 22 times by the OpAmp (second half of the LM358). The OpAmp output is connected to the multiplexer pin F2. A 10kΩ resistor provides power to the audio input, as required by an electret microphone.

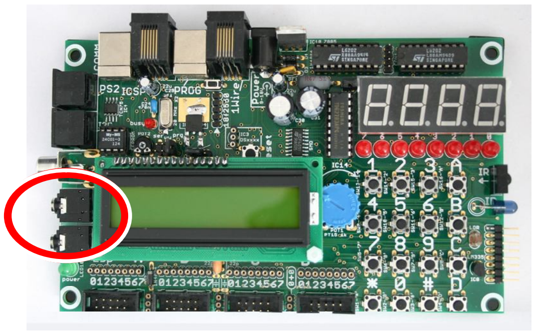
2.15.1 Software interface – audio input

The Loudspeaker receiver is addressed by making pins RE0..RE2 outputs and the value on these pins 0b010, and pin RA0 an analog input.

16F887 pins	Direction	Value
RA0	Analog input	This pin follows the (amplified) analog input.
RE0 … RE2	Outputs	0b010

2.16 *Composite video output connector*





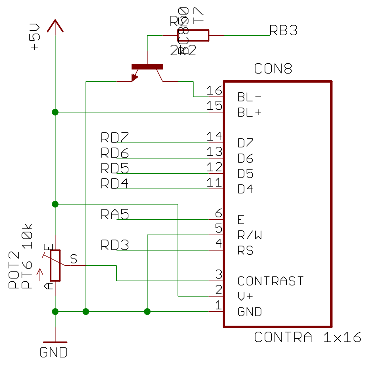
The middle connector at the left side of the board is a composite video output connector. The PIC16F887 pins RB4 and RB5 are connected to a crude two-resistor D/A converter. The resulting signal is buffered by the transistor and provided on the video connector. The pins RB4 and RB5 are used only for this video output circuit. The resistor network also serves to hold the RB4 and RB5 pins low during programming, which prevents inadvertent activation of the LVP programming mode (this is not needed on the 16F887, which does not supported this mode, but it is needed on some other PIC chips).

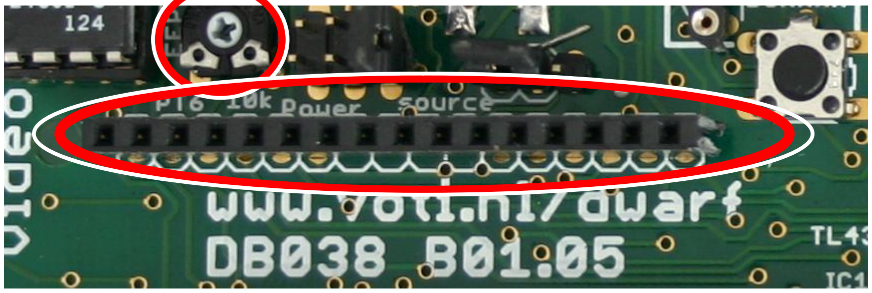
2.16.1 Software interface – video output

The RB4 and RB5 pins are dedicated to the video output.

16F887 pins	Direction	Value
RB4, RB5	Outputs	0b00 : sync (0.0V) 0b10 : black (0.3V) 0b11 : white (1.0V)

2.17 *HD44780 character LCD connector*





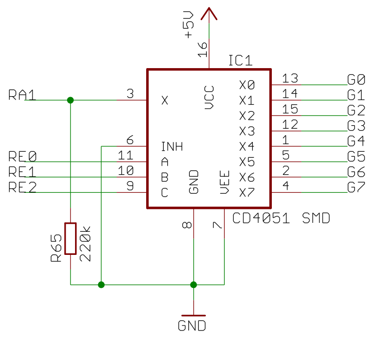
Optionally an HD44780-compatible LCD can be fitted on the board. It will cover the 16F887 chip, so if you want to access the PIC chip you will have to (temporarily) remove the LCD. The RD3 ... RD7 pins connect to the LCD's RS and D4 ... D8 lines. The LCD's R/W line is permanently tied to ground, so the PIC can write to the LCD, but not read. A high value on pin RB3 enabled the LCD background light. The RD3 ... RD7 lines serve many other functions on the board beside the LCD interface, but they can be programmed to do nothing on to those other functions while accessing the LCD. The LCD's E line is connected to the PICs RA5 pin, which is dedicated to this function. Keeping this line low prevents messing with the LCD while the RD3 ... RD7 pins are used for other purposes. The potentiometer that sets the contrast voltage for the LCD is the small potentiometer located just above the LCD connector, at the left side. Its normal setting (0V) is fully counter-clockwise (as indicated on the picture, the arrow is at 8 hours, which is 0 Volt).

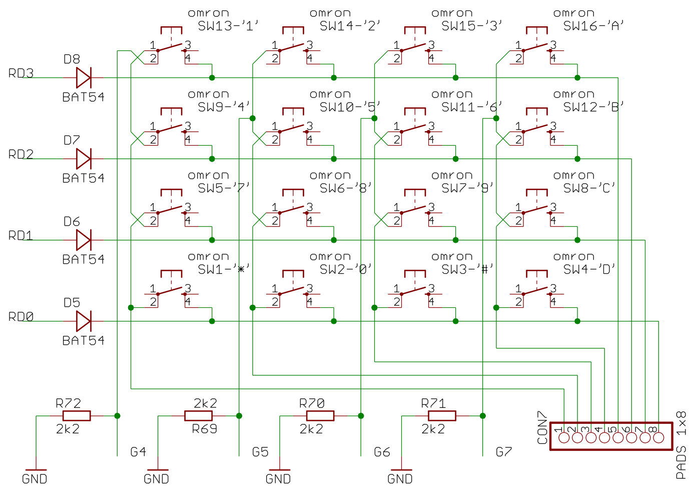
2.17.1 Software interface – character LCD

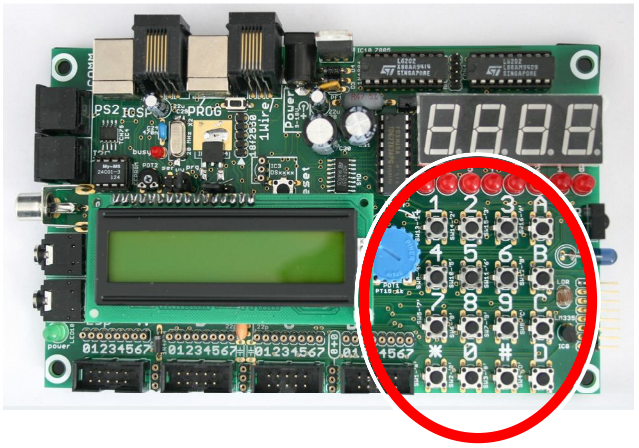
The RA5 pin is dedicated to the LCD. The other pins (RD3 ... RD7) are shared with other functions.

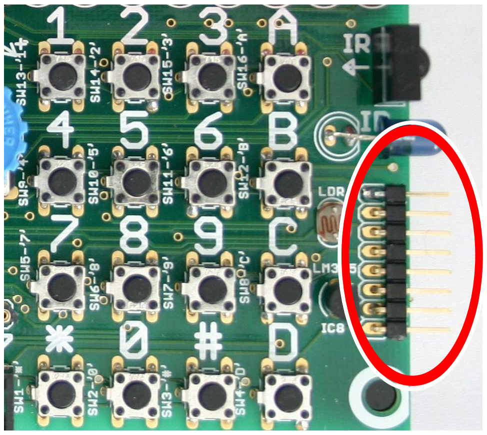
16F887 pins	Direction	HD44780 line
RA5	Output	E
RB3	Output	High enables backlight
RD3	Output	RS
RD4 ... RD7	Outputs	D4 ... D7

2.18 *On-board keypad and external keypad pads*









The 16F887 pins RD0..RD3 can, via the diodes D5..D8, drive one row of the left side of the keypad matrix high. The multiplexer outputs G4..G7 can be read to determine whether a switch on that row is pressed. The common side of the G multiplexer has a weak pull-down to make sure that a 0 will be read when no key is pressed. Connector CON8 can be used to connect an external 4 x 4 keypad parallel to the on-board keypad.

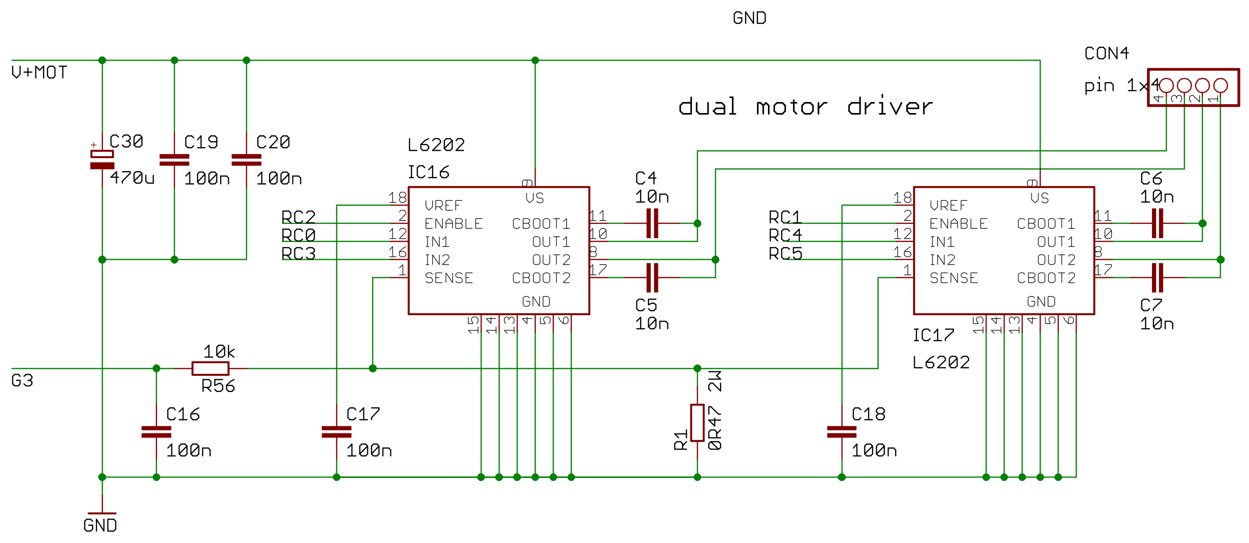
2.18.1 Software interface - keypad

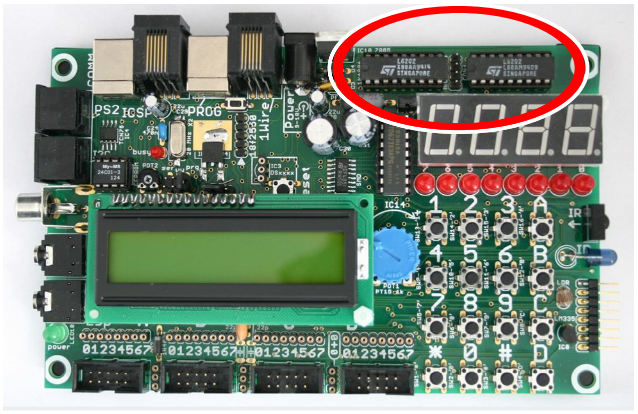
A single row of the 4 x 4 keypad is activated by making one of the 16F887 pins RD0 … RD3 output and high (and the other three low, or input). Pin RD0 corresponds to the lowest row (*, 0, #, D), etc. A column is selected by making pin RA1 a (digital) input, and pins RE0 … RE2 outputs and the value on these pins 0b100 for the leftmost column, 0b101 for the next column, etc. Pin RA1 will read 0b1 when the switch at the selected row and column is pressed, 0b0 when it is not.

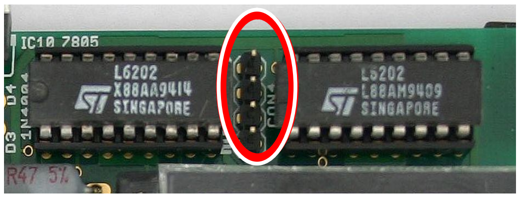
The pull-down provided by R65 (220kΩ) is weak. Hence a small delay must be observed between setting the RE0..RE2 and RD0..RD3 outputs, and reading the RA1 input. A delay of 10 µs should be sufficient.

16F887 pins	**Direction**	**Value**	
RA1	Digital input	0b0	Selected button is not pressed
		0b1	Selected button is pressed
RD0 … RD3	Outputs	0b0001	Selects row 0 (bottom row)
		0b0010	Selects row 1
		0b0100	Selects row 2
		0b1000	Selects row 3 (top row)
RE0 … RE2	Outputs	0b100	Selects column 0 (leftmost column)
		0b101	Selects column 1
		0b110	Selects column 2
		0b111	Selects column 3 (rightmost column)

2.19 Motor driver







The motor driver circuit uses two L6202 H-bridge chips. Resistor R1 is placed in series with the motor current. The voltage over this resistor is filtered by R56 / C16 and provided to multiplexer pin G3, to allow the 16F887 chip to determine the motor current. This can for instance be used to detect a stall.

Two motors can be connected to the 4-position header.

The L6202 chips can theoretically handle a continuous current of 1 A at 12 … 48 V (per chip). The 7805 that regulates the +5V for the rest of the board has a maximum input voltage of 35V. I think a maximum of 500 mA at 12 ... 24V is more realistic. Note that the L6202 is not designed to work below 12 V.

2.19.1 Software interface – motor current

The motor current causes a voltage across resistor R1. This voltage can be read by the 16F887. To use this function pin RA2 must be configured as an input and as an analog input, and pins RE0 … RE2 must be outputs and the value on these pins must be 0b011. The resistor will cause a voltage drop of 0.5mV per mA motor current.

16F887 pins	Direction	Value
RA2	Input, Analog	Motor current, 0.5mV per mA
RE0 … RE2	Outputs	0b011

2.19.2 Software interface – H bridges

The pins RC0 … RC5 are dedicated to driving the H-bridges. Pins RC1 and RC2 connect to the PWM modules of the 16F887, so those can be used to control the motor speeds.

16F887 pins	Direction	Value
RC0	Output	Motor 1 A
RC1	Output	Motor 2 enable
RC2	Output	Motor 1 enable
RC3	Output	Motor 1 B
RC4	Output	Motor 2 A
RC5	Output	Motor 2 B

A B value	Motor effect (if enable is high)
0 0	Brake
0 1	Forward
1 0	Backward
1 1	Brake

3 PIC 14-bit architecture and instruction set summary

3.1 *Introduction*

The 16F887 is a PIC with a 14-bit core (each instruction occupies 14 bits). Other PICs have 12, 16, 24 or 32 bit cores. Unless another core size is mentioned this summary is about 14-bit cores only.

When you program your PICs using a High Level language the compiler will isolate you from most details of the PIC architecture, but some details will still shine through. So even when you use a compiler it is still a good idea to have some knowledge of the PIC architecture, if only to know what kind of constructs can be translated compactly to PIC instructions and which can't.

The PIC architecture is a bit peculiar, even (or especially) for people who are familiar with more mainstream processor architectures. The PIC uses the Harvard architecture: the code and data spaces are completely separate. Most other CPU's use the VonNeuman architecture, where code and data share a common address range. On a PIC code address 0 and data address 0 have nothing to do with each other, and can even address a different number of bits. The addressable element in the PIC data space is a byte (8 bits), the addressable element in the PIC code space is the instruction, which is 14 bits on the 14-bit cores.

The basic PIC instruction set is very simple. The processor is byte (8 bit at a time) oriented. There is one special register, called the W register. (The more common name for such a register is ‘accumulator’.) This register is part of the CPU, it does not appear in any memory space. Most instructions specify a data memory location, and work on the content of that data memory location and the content of the W register. The result is either left in the W register, or left in the data memory location. The processor maintains a status register that reflects the zero, carry and digit-carry status of the last calculation. The digit-carry flag is useful only for calculations on BCD-coded data, and is not discussed in this summary.

The 14-bit PIC instruction set was designed for a maximum of 2048 instructions and 128 data addresses. This was sufficient for the first 14-bit core chips, but recent chips contain more code and data. The 16F887 for instance has 8192 instructions and 512 data locations. Code pages and memory banks are used to increase the addressing range of the basic instruction set, which complicates the use of the PIC instruction set considerably.

The datasheet of each PIC chip contains a short description of the instructions set. A more in-depth description can be found in the Midrange Reference Manual.

In the instruction tables **a** represents an address, **[a]** the (byte) content of that address, **n** a (byte) literal, **b** a 3-bit literal (a bit number, 0..7), and **X . b** represents bit b of X.

As stated, most instructions that calculate a result can leave this result either the data location, or in the W register. This is indicated by ',F' or ',W'. But actually F and W are just pre-defined constants (1 and 0), so the assembler accepts strange things like 'INCF W,W' which is interpreted as 'INCF 0,W' (which is probably not what the programmer intended). When the ',F' or ',W' is omitted the assembler uses a default. I always forget what it is (if you don’t consider that a reader of your software probably will), so it is advised always to specify the target explicitly.

MPASM, the Microchip assembler, can be used in two modes: absolute or relocatable. Absolute mode translates a single assembly source file, which can use textual include (#include) to include other ‘library’ files. Relocatable mode can use multiple source files, and uses a linker script to determine to direct the placement of code and data in the PIC address ranges. The presence of a linker script in the list of project files selects relocatable mode, the absence selects absolute mode.

3.2 Data load and store

There MOVLW instruction loads the W register with a literal. The MOVWF instruction saves the W register to a memory location. These instructions do not affect the flags. The MOVFW instruction that loads the W register from a memory location is technically an arithmetic instruction: it affects the zero flag, and its destination can be either the W registers or (!) the memory location. The CLRW instructions clears the W register, the CLRF instruction clears a memory location. Both set the zero flag.

Operation	Opcode	Notes
W := n	MOVLW n	-
[a] := W	MOVWF a	-
W := [a]	MOVF a, W MOVFW a	affects Z
W := 0	CLRW	sets Z
[a] := 0	CLRF a	sets Z

Note that the assembler accepts “MOVFW a” as synonym for “MOVF a,W”.

3.3 Arithmetic and logic instructions

The monadic (one-operand) instructions operate on the content of a memory location specified in the instruction. The result can be stored either in the memory location or in W. These instructions include increment, decrement, rotate left, rotate right, swap nibbles, and (!) move.

Operation	Opcode for		notes
	W := f([a])	[a] = f([a])	
Increment	INCF a, W	INCF a, F	affects Z
Decrement	DECF a, W	DECF a, F	affects Z
Rotate right	RRF a, W	RRF a, F	involves C
Rotate left	RLF a, W	RLF a, F	involves C
Swap nibbles	SWAPF a, W	SWAPF a, F	affects no flags
Move	MOVF a, W	MOVF a, F	affects Z

Note the rather peculiar instruction “MOVF a, F”. This instruction copies the content of a memory location to … that same memory location. Its only useable effect is that it sets or clears the Z flag depending on whether the content is zero or not.

The dyadic (two-operand) arithmetic instructions have two forms. The first form is like the monadic instructions. It operates on a memory location specified in the instruction, uses the W register as second operand, and can store the result either in the memory location or in W. The second form operates on an 8-bit constant, uses the W register as second operand, and stores the result in the W register. The arithmetic operations that can be done in both forms are add, subtract, and, or and xor. Note that W is the second operand, not the first. This makes a difference for subtraction only, but has confused many people.

Add and subtract affect all three flags. Rotate involves the carry bit. The others affect only the zero flag, except swap nibbles which affects no flag at all. For subtracting the carry flag is set when no carry occurs and cleared when a carry occurs, so it is often referred to as the carry - /borrow flag: carry refers to the carry for addition, /borrow refers to the negated (the / is used to indicate negation) borrow for subtraction.

Operation	Opcode for			notes
	W := [a] op W	[a] := [a] op W	W = n op W	
Add	ADDWF a, W	ADDWF a, F	ADDLW n	affects Z, C, DC
Subtract	SUBWF a, W	SUBWF a, F	SUBLW n	affects Z, C, DC
And	ANDWF a, W	ANDWF a, F	ANDLW n	affects Z
Or	IORWF a, W	IORWF a, F	IORLW n	affects Z
Xor	XORWF a, W	XORWF a, F	XORLW n	affects Z

3.4 Bit manipulation

The are bit set and bit clear instructions set or clear a single bit in a memory location. Note that both the memory location and the affected bit within the memory location are encoded in the instruction. For the memory location indirect addressing can be used to affect a calculated address, but there is no direct way to affect a calculated bit within a byte.

Operation	Opcode
[a] . b = 1	BSF a, b
[a] . b = 0	BCF a, b

The assembler accepts the synonyms show in the table below for setting and clearing the flag bits in the status register.

Synonym	Effect	Same as
SETZ	Set zero flag	BSF 3, 0
CLRZ	Clear zero flag	BCF 3, 0
SETC	Set carry flag	BSF 3, 2
CLRC	Clear carry flag	BCF 3, 2

3.5 Control: jump and subroutines

The processor has instructions for

- transferring control to a different location (GOTO), and for
- first storing the location of the next instruction on the stack, and then transferring control to a different location (CALL).

The RETURN instruction retrieves (and deletes) the last location stored by a CALL and returns control back to it. CALL and RETURN are used to implement subroutines. A special form of the RETURN instruction (RETLW) stores the specified literal value in the W register before control is transferred to the retrieved location.

Operation	Opcode
Jump to x	GOTO x
Store return address, then jump to X	CALL x
Retrieve stored return address and jump to it.	RETURN
W := n Retrieve stored return address and jump to it.	RETLW n

The chip has a primitive stack that is used for one purpose only: saving and storing (code) return addresses. This stack has 8 entries. For pushing the stack behaves like an 8-entry book shelve: when you push a new address onto the stack (let's say at the left end), the oldest address (at the right end) falls off and is lost forever. For popping there appears to be a copying machine at the far end: when you take (pop) the leftmost book from the shelve the rightmost book is duplicated. Hence there are always exactly 8 addresses in the stack.

When more than 8 addresses are pushed onto the stack the oldest address will be lost. This in itself is no problem, but when subsequently more than 8 addresses are popped the 9th pop will return the same address as the 8th pop, which will likely cause the program to crash. There is no way for the processor to detect that a stack overflow or underflow has occurred. When the MPLAB simulator is used to simulate a program a warning can be issued. Most high-level language compilers will analyze a program and give a warning or error when it could overflow the stack. But assembler programmers are on their own.

Note that there are no conditional jump instructions as found in most other processors. Instead the conditional execution instructions can be combined with a GOTO to create a conditional jump. But because all instructions can be executed conditionally GOTO's can often be avoided.

The RETLW instruction is mainly used to create tables of constant data on PICs that cannot read their code memory (or to be compatible with such PICs). This technique is not discussed here.

3.6 *Control: conditional execution*

The bit-test-and-skip instructions skip the next instruction when a single bit in a memory location is either clear or set. A strange omission is that the W register is not memory-mapped, so testing a bit in the W register is more complicated than testing a bit somewhere in memory. The status register is memory-mapped (and mapped into all banks), so the bit-test-and-skip instructions can use the status flags. Like for the bit set and clear instructions, both the address and the bit are constants encoded in the instruction.

Operation	Opcode
skip next instruction when [a] . b is set	BTFSS a, b
skip next instruction when [a] . b is clear	BTFSC a, b

Skip instructions are often used to skip an instruction depending on the current value of one of the flag bits in the status register. To make this use of BTFSS and BTFSC more readable the assembler accepts the synonyms show in the table below.

Synonym	Effect	Same as
SKPZ	Skip next instruction if and only if the zero flag is set	BTFSS 3, 0
SKPNZ	Skip next instruction if and only if the zero flag is not set	BTFSC 3, 0
SKPC	Skip next instruction if and only if the carry flag is set	BTFSS 3, 2
SKPNC	Skip next instruction if and only if the carry flag is not set	BTFSC 3, 2

Two special increment and decrement instructions exit that look a lot like the increment and decrement already described, but these forms

- do not affect the zero flag;
- skip the next instruction when the result is zero.

Operation	Opcode
[a] := [a] - 1 or W := [a] - 1 skip next instruction when result is 0	DECFSZ a, F DECFSZ a, W
[a] := [a] + 1 or	INCFSZ a, F

W := [a] + 1 skip next instruction when result is 0	INCFSZ a, W

The normal decrement and increment instructions (DECF, INCF) affect the zero flag. When you need a decrement or increment instruction that does not affect any flags you can use DECFSZ or INCFSZ followed by a NOP.

3.7 SLEEP, the watchdog, interrupts, NOP

The sleep instruction puts the chip in a very low power state. This sleep state can be terminated by a reset, or by an interrupt.

The chip has a watchdog timer. When this timer is enabled (in the configuration fuses) it will reset the chip, unless the CLRWDT instruction (CLeaR WatchDog Timer) is executed periodically. The watchdog is enabled by a bit in the Configuration Fuses, check the 16F887 datasheet, section'special features of the CPU'. The standard use is that the application has one main loop, which contains a single CLRWDT instruction. When the program crashes the main loop will not be executed any more, hence the CLRWDT instruction is not executed. But the watchdog timer keeps running, and when it rolls over the chip is reset.

The processor has a simple interrupt mechanism. Various interrupt sources can trigger an interrupt. When an interrupt is accepted by the processor it executes what is effectively a CALL to address 0x04, and it clears the GIE (Global Interrupt Enable) flag to prevent further interrupts. The interrupt service routine should end with a RETFIE instruction, which is effectively a RETURN instruction that also sets the GIE flag, thus enabling further interrupts. Only one interrupt can be active at any time. It is up to the interrupt service routine to determine the cause of the interrupt and to take appropriate action, including clearing the interrupt source. When the processor was in the sleep condition when the interrupt occurred the RETFIE at the end of the interrupt service routine will return to the instruction immediately after the SLEEP instruction.

Operation	Opcode
Put the chip in sleep (low power) mode	SLEEP
Clear the watchdog timer	CLRWDT
Return from interrupt	RETFIE
Do nothing	NOP

Upon an interrupt the processor saves the current program counter, but nothing else. It is up to the interrupt service routine to save and restore the W register and any file registers that it uses. It will be very difficult to avoid using the W register or affecting the flags, so the interrupt entry and exit code will have to save and restore at least W and the flags. Note that this entry and exit code cannot make any assumptions about the currently selected register bank. Hence it needs two file registers (SAVE_W and SAVE_STATUS) that are available in all banks and one file register (SAVE_PCLATH) that can be in any bank. Note the use of SWAPF instructions to avoid affecting the flags. When the application uses more than just code page 0, and the interrupt code does any PC manipulation (GOTO, CALL, write to PCL) the PCLATH register must be also be saved at entry, set to page 0 *(because that is where the interrupt code is), and restored at exit.

```
interrupt entry

   ; save W and flags
   MOVWF   SAVE_W
   SWAPF   STATUS, W
   MOVWF   SAVE_STATUS

   ; save PCLATH, clear PCLATH (page 0)
```

```
    MOVFW   PCLATH
    BANKSEL SAVE_PCLATH
    MOVWF   SAVE_PCLATH
    CLRF    PCLATH
```

```
                interrupt exit

    ; restore PCLATH
    BANKSEL SAVE_PCLATH
    MOVFW   SAVE_PCLATH
    MOVWF   PCLATH

    ; restore flags and W
    SWAPF   SAVE_STATUS, W
    MOVWF   STATUS
    SWAPF   SAVE_W, F
    SWAPF   SAVE_W, W
```

A surprisingly common mistake made by PIC beginners is to enable the watchdog without knowing. (The bit that switches the watchdog on or off is in the configuration flags.) This will let the program run for a short time, then the watchdog will trigger and the chip resets. When the program is 'blink-a-LED' it might even seem to work correctly, although the blinking will probably be irregular and/or at an unexpected frequency.

3.8 Code memory and code addressing

Control instructions that specify a new code location (GOTO and CALL) contain only a limited number of bits for this new location. On the PICs that have a code space that is larger than this number of bits can specify (2048 instructions for the 14-bit cores) code paging is used. This is a trick to specify the remaining (higher) bits: those bits are taken from a fixed location (the PCLATH register). Before a GOTO or CALL you must make sure that these bits are set appropriately, otherwise your instruction will lead your program to an unexpected location (often referred to as never-never land). The table below shows how the target address is constructed. The colons (:) mean that the various bits are concatenated. The RETURN (and RETLW) instructions are not mentioned here, because the stack stores all bits of the return address.

new location after a GOTO or CALL	[2 bits from PCLATH] : [11 bits from instruction]
new location after modification of PCL	[5 bits from PCLATH] : [8 (modified) PCL bits]

The value of the PCLATH register after a reset is 0x00, so if you

- do not write to the PCL, and
- your code is smaller than 2048 instructions (11 bit address)

you don’t have to worry about code paging. Turning this logic around: when you have just added a small amount of code to a perfectly working program and it suddenly starts acting weird it might be a good idea to check whether the size of your code has crossed a 'magic border' and you must start paying attention to code paging.

When the size of your code increases above the no-worries limit a simple way to keep your program working for jumps and calls is to use the **LGOTO** and **LCALL** pseudo-instructions. The assembler translates these pseudo-instructions to a setting of the PCLATH register, followed by the GOTO or CALL instruction. Another option is to use the **PAGESEL** pseudo-instruction, followed by a GOTO or CALL. But note that conditionally executing an LGOTO or LCALL will not produce the expected result, because only the first instruction will be skipped! For a conditional jump or call the PAGESEL must be put before the skip:

```
                   Conditional long jump

   PAGESEL destination      ; set page bits for 'destination'
   SKPNZ                    ; skip if not zero
      GOTO destination      ; jump
```

The code produced by the LCALL, LGOTO, PAGESEL pseudo-instructions depends on the chip for which the code is generated. On chips with less than 2048 instructions (one code page) no code will be produced beside the GOTO and CALL. (A consequence is that a conditional LGOTO or LCALL will work on such chips, but will fail on chips with more code pages!) On chips with more than one code page a sequence of one or two BCF and BSF instructions will be generated, followed (for LGOTO and LCALL) by the GOTO or CALL. An unfortunate consequence is that you cannot be sure how many instructions will be generated (hence you do not know how long executing these instructions will take).

The older PICs could not read their own code, but newer PICs implement a special procedure to read the content of the code memory. This procedure is described in the datasheet chapter “DATA EEPROM and Flash program Memory Control”. On most recent PICs (including the 16F887) the processor can also write to the code memory, but such a write halts the processor for a considerable time so it is not very useful for a normal application. This self-writing feature is what makes a boot loader possible: a program that communicates, often with a PC, to receive an application program that is stored in the (remaining) code memory, so no programmer is needed to load new code. This is often used to allow a field service to update the application code.

3.9 Data memory and data addressing

The PIC documentation calls the data space 'file registers'. You could interpret this as stating that the data RAM (file) and special purpose hardware (registers) are mapped in the same address space.

Just like the code space the data space of recent PIC chips is larger than can be specified in an instruction. The same trick as for the code space is used: some bits of the effective address are taken from a fixed location, and the programmer must make sure that those bits are set appropriately, but in this case the mechanism is called register file banking. The table below shows how the effective data address is constructed.

effective **direct** (instruction-specified) address	[3 bits from STATUS] : [7 bits from instruction]
effective **indirect** (FSR) address	[1 bits from STATUS] : [8 bits from FSR]

Indirect addressing is used when an address must be calculated at run time. The PIC instruction set itself does not support indirection (the use of a calculated address). Instead two memory mapped registers are used: the FSR register acts as pointer, the INDF 'register' acts as if it were the registers who's address is currently in FSR. This is roughly equivalent to **indirection** as it is available on other processor architectures, but with only a single pointer register.

The data addressing scheme shown so far has a big problem: to move data from one bank to another bank a LOT of bank selection instructions must be used. To reduce this problem some addresses 'map to' the corresponding address in bank 0 (or sometimes to another bank). Another way to say the same thing is that some registers appear at more than one address. The registers that map to all banks are called **shared**. The shared data RAM is very convenient for the assembler programmer, but there are only 16 shared addresses on most 14-bit cores (but some PICs have no shared RAM at all).

When your program does not use too much data the most convenient data banking strategy is to keep the bank selection bits pointing to bank 0, only changing these bits when you must access a (probably special function) register in another bank, and then set the bank bits back immediately afterwards. Use the RAM in the higher data banks (mostly or only) using indirect addressing.

You can of course set the bank selection bits yourself, using for instance BCF and BSF instructions. MPASM provides the **BANKSEL** macro to set the bank bits for accessing a specified register address using an instruction. Likewise the **BANKISEL** macro sets the (single) bank bit for indirect addressing. When you use this macro to access a data array you must make sure that the whole data array that you are going to access is within one bank.

The PICs data memory banking, with fragmented pieces of RAM, makes accessing an array that is larger than the amount of contiguous RAM very tedious. For each access you will have to translate the array index to the correct bank bits and the address within the bank. The irregular layout of the register banks makes it impossible to do this with just a few instructions. Even high-level languages often restrict the size of an array to what will fit in a single memory bank.

3.10 Instruction timing

The 14-bit core PICs typically run at a maximum clock frequency of 20 MHz. A PIC requires 4 clocks per instructions, so the chip runs at 5 MIPS (5 million instructions per second). One instruction takes 0.2 µs. This is true for all instructions (even skipped instructions!), except those that write to the program counter. Such instructions count as 2, taking 8 clocks, or 0.4 µs at 20 MHz.

An assembler programmer will often find the need to insert some delay between instructions. The NOP instruction provides a one-instruction delay. A two-instruction delay can be created with one instruction by a jump (GOTO) to the next instruction. Even though the jump does not modify the program counter it still writes to it, so the instruction still counts as two normal instructions for the instruction timing. A four-instruction delay can be created by a CALL to a subroutine that contains only a RETURN. There will often be a RETURN instruction somewhere, so in effect only the CALL needs to be added. But note that this construct uses a stack entry. If two stack entries are available an 8-instruction delay can be realized by the curious construction of calling a subroutine, which consists of a CALL to its own RETURN, immediately followed by that return instruction. The same RETURN instruction serves both the initial subroutine call and the CALL inside the subroutine. Note that this construct uses 2 stack entries. This can easily be extended to provide 16, 24, 32, etc. instructions delay.

1-instruction delay	2-instruction delay	4-instruction delay	8-instruction delay
`   NOP`	`   GOTO next`	`   CALL delay4`	`   CALL delay8`
	`next:`		
		`delay4:`	`delay8:`
		`   RETURN`	`   CALL delay4`
			`delay4:`
			`   RETURN`

3.11 Using I/O pins

On 14-bit core chips that have analog capabilities the pins that can have an analog function default to that analog function. This is not a coincidence: when a digital IO pin is tied to a voltage somewhere between high and low this can cause the internal circuits to drawn much more current than when IO pin levels are well-defined. A digital signal on an analog pin is no problem, but the reverse is, so the pins that have analog capabilities default to analog. But this means that for digital use such pins must first be configured for digital use! On the 18F887 this applies to some of the pins of PORTA, PORTB and PORTE. Note that for digital use both the A/D converter and the Comparator might need to be disabled.

Contrary to the analog functions, which are generally enabled by default, most digital special IO functions (UART, I2C, PWM etc) must be enabled before they can be used, and often the pin must also be configured for the appropriate direction. When using a special digital function it is advised to read both the section on those functions, and the section on the IO port and pin.

PICs use a funny IO architecture where a reading of a pin always returns the current external level of the pin, which can be different from the last value written to a pin, even when the pin is set as output, for two reasons:

- PIC instructions execute in a 2-stage pipeline, and for IO pins the read part of the next instruction takes place before the writing of the previous instruction.
- The load on the pin can be too high for it to reach its 'desired' level. This can easily happen (for a short time) when the load is capacitive, but it can also happen with a static load that is well within the normal operating specifications.

All PIC instructions that modify some bits in a byte are read-modify-write instructions, so when for instance two consecutive BCF (bit clear) instructions on an IO port are executed the second instruction can ruin the effect of the first because of the first of the two reasons. Actually a BCF instruction (and a lot of other instructions, like INCF) on a port can ruin any previous setting of that port because of the second reason! It should be noted that the first reason occurs far more often, and can be avoided by placing a NOP or another instruction between any two read-modify-write instructions on the same IO port. But to really avoid all problems it is advised to allocate a separate register, do manipulations on that register, and copy it to the port register after each change. This technique is called a “using a shadow register”.

For some purposes (instance I2C, PS/2, Dallas 1-wire) open-collector (or open-drain) IO pins are needed. Most PICs have one such pin (RA4), but other pins can be used in this mode by writing a 0 to the pin itself and then writing the desired IO state to the TRIS (direction) register. A 0 in the TRIS register happens to set the bit to output, so writing to the TRIS register mimics writing to an open-collector output pin. (But note that the TRIS registers are not in bank 0!)

3.12 Configuration fuses

Some configuration aspects of a PIC chip cannot be set from a running program, for instance the type of oscillator: the hardware must know this before the program can run at all! These aspects are configured in the configuration fuses. The proper place to specify the configuration fuses values is in the source of your program, often near the top of your file. The configuration fuses can also be specified in the Configure > Configuration Bits… tab in MPLAB, or even in the application used to program the PIC. Don’t do that, without the configuration fuses value your program is not complete.

The configuration bits that are available differ for various PICs. Check the chapter “Special features of the CPU” in the chip’s datasheet for the details. For the 16F887 the configuration bits are split in two words, CONFIG1 and CONFIG2. The tables below show explains the features that can be configured and the gives the suggested value for the DB038 board.

CONFIG1		
Name	Explanation	Suggested value
DEBUG	Selects whether the internal debugger hardware is enabled. The debugger hardware claims the pins RB6 and RB7 for communication with the external debugger hardware. The DB038 programmer is not meant to be used as debugger.	_DEBUG_OFF
LVP	Selects whether Low Voltage Programming is enabled. LVP claims the RB3 pin. LVP is not needed on a DB038 (pin RB3 is used for other purposes), so keep this feature disabled.	_LVP_OFF
FCMEM	Selects whether the fail-safe clock monitor is enabled. Keep this disabled unless you need and understand this feature.	_FCMEN_OFF
IESO	Selects whether the Internal-External Switchover feature is enabled. Keep this disabled unless you need and understand this feature.	_IESO_OFF
BOREN	Selects whether and when brown-out reset is enabled. Keep this disabled unless you need and understand this feature.	_BOR_OFF
CPD	Selects whether Data Memory read protection is enabled.	_CPD_OFF
CP	Selects whether Code Memory read protection is enabled.	_CP_OFF
MCLRE	Selects whether the RE3/MCLR pin is used as MCLR or as RE3. The DB038 board assumes the pin is MCLR.	_MCLRE_OFF
PWRTE	Selects whether the power-up timer is enabled. Keep this enabled unless you understand this feature and need to run without it.	_PWRTE_ON
WDTE	Selects whether the watchdog timer is enabled. Keep this disabled unless you need and understand this feature.	_WDT_OFF
FOSC	Selects the type of oscillator and the functions of the RA7 and RA7 pins. The DB038 board uses a 20 MHz resonator, so this feature must be configured accordingly.	_HS_OSC

CONFIG2		
Name	Explanation	Suggested value
WRT	Selects which parts of the code memory (if any) are write protected. Writing to code memory is not for the beginner, so it is advised to make as much code memory write-protected as possible, until you need and fully understand this feature.	_WRT_HALF
BOR4V	Selects the brown-out trip level. If the brown-out feature is used (BOREN setting in CONFIG1) it is advised to set it to 4.0V (the alternative is 2.1 V).	_BOR40V

In an assembly program the configuration fuses can be specified using the __config directive. The code snippet below shows the suggested configuration for the 16F887. The #include must be before the __config lines, because this file contains e definitions of __CONFIG1, __DEBUG_OFF, etc.

```
   ; include target chip stuff
   #include <P16F887.INC>

   ;  configuration settings
   __config _CONFIG1, _DEBUG_OFF & _LVP_OFF & _FCMEN_OFF & _IESO_OFF & \
        _BOR_OFF & _CP_OFF & _MCLRE_OFF & _PWRTE_ON & _WDT_OFF & _HS_OSC
    __config _CONFIG2, _WRT_HALF & _BOR40V
```

Like everything in the real world code protection on PICs is not absolute. The ancient 16c84 was used widely in pay-TV decoders, so it was a favorite target for hacking and receipts for defeating its code protection were circulating on the web. The newer PICs are reported to be invulnerable to these (rather simple) attacks, but undoubtedly other attacks exist. There are even companies that offer to retrieve the code from any protected PIC, but such services are not cheap and often require more than one PIC, and a considerable amount of money. The best (and probably only) way to protect your code from piracy is to make sure that it is not worth the effort by keeping the price of your product low compared to the effort of breaking the code protection. Note that this vulnerability of code-protected chips is not limited to PICs.

3.13 The 12-bit core instruction set

The 12-bit core PICs (like the 12F509 and 10F200) are the small brothers of the 14-bit core chips. The 12-bit instruction set lacks a few instructions, and there are fewer bits to specify a code or data address. This makes programming a 12-bit core trickier. Instructions that are missing compared to a 14-bit core are:

- There is no RETURN instruction, only RETLW.
- There are no interrupts, so there is no RETFIE.
- There are no literal forms of add and subtract.

There is no RETURN instruction, but unfortunately the assembler still accepts the RETURN mnemonic, and translates it to a 'RETLW 0', which clears the W register. This can be a very unpleasant surprise to a programmer who is used to a 14-bit core, where the W register can be used to return a calculated value.

Code addressing on 12-bit cores is summarized in the next table. Note one very strange detail: one bit of the new code address after a call or modification of PCL (but not after a GOTO) is always 0. As a consequence only half the code space is accessible for these actions, and the accessible and inaccessible regions alternate. This complicates the use of jump tables and subroutine calls. A solution to this problem it to put entry points for all subroutines at the start of your code. An entry point uses a GOTO to jump to the subroutine proper.

new location after a GOTO	[2 bits from PCLATH] : [9 bits from instruction]
new location after a CALL	[2 bits from PCLATH] : [0] : [8 bits from instruction]
new location after modification of PCL	[3 bits from PCLATH] : [0] : [8 (modified) PCL bits]

The stack on 12-bit core chips has only two levels. This often requires some convoluted programming.

3.14 Reflections on the PIC architecture

The PIC instruction set can both be effective and powerful, resulting in very compact and fast code, or cumbersome and deficient, resulting in bloated and slow code. It all depends. The PIC instruction set is very good at the bit-manipulations and byte-sized arithmetic that are typical for small controller applications. Problems arise when either multi-byte arithmetic is needed or the code and/or data is so large that the paging and banking gets in the way. This typically happens when more complex calculations or larger data sets (both comparatively speaking) are required. The use of a compiler will often shield you from the ugly details, but it cannot hide the large amounts of code required to do things that can be done with much less code on more mainstream architectures. On the other hand good compilers will take advantage of the power of the PIC instruction set for simple operations.

To illustrate the weak side of the PIC architecture imagine an application that requires random access to a 120-byte array. There are PICs that have 120 and more bytes of RAM data, but this is spread over four memory banks, none of which contains 120 bytes of data. So the data access routine has to figure out, for each byte, in which bank the byte is, set the banking bits appropriately, and to adjust the address according to the starting address of the data fragment in that bank. Store the adjusted address in FSR and the byte can be accessed in INDF. But to access the next random byte this whole procedure has to be done again. Contrast this with the very simple way in which a small array that fits in one bank can be accessed: set the bank bits, add the start address of the array within the bank and the index, move the result to FSR and use the byte in INDF. To access the next byte, just change the FSR accordingly.

When you argue that you will use C (or another high-level language) and be shielded from these details you might be in for an unpleasant surprise: most compilers do not support arrays spread over multiple banks (some don't even support banking at all), and those who do will still produce the same large code that you would produce by hand, increasing the size of your application and reducing its execution speed.

My conclusion about the PIC architecture is that for PICs the relation between 'application complexity' and 'engineering effort' is much less smooth than for more regular architectures (especially due to paging and banking). This means that PICs will often do well (read: as good as or better than other architecture) for smaller applications, but do not so well (worse than other architectures) for larger applications. But a lot of experience is needed to rate an application's complexity before applying this rule; it is certainly not just the amount of code, data or MIPSs.

One might ask: why use PIC microcontrollers at all? Alternatives are available with an instruction set that is much easier for the assembler programmer (for instance the Atmel AVR chips). Apart from inertia (PICs are popular, so the next generation finds the internet full of PIC examples, so they will use PICs too, which will lead to even more internet coverage for these chips) there are three reasons to choose Microchip’s PIC chips:

1. There is a very wide range of PIC chips too choose from, from tiny and primitive 6-pin chips that are no larger than an SMD transistor, to 100-pin chips with all sorts of peripherals, like Ethernet, CAN and USB.
2. Microchip has a very good reputation for availability of its products. For a manufacturer who designs a product now and hopes to sell it for the next 10 years this can be more important than any other argument (including ease of use and even price).
3. For an assembler programming course one might argue that after mastering the horribly complex 14-bit PIC architecture everything else will be easy.

Note that these arguments do not apply to all situations. For someone who wants to build a one-off project now none will matter, and he will probably be better off with for instance an AVR or ARM microcontroller, or even a PC.

4 PIC hardware notes and tips

The information about a PIC chip is spread over a number of documents:

- The data sheet of the chip (or a related set of chips) itself.
- The errata, documenting the differences between the datasheet and the actual silicon as currently available.
- The midrange reference manual (for the 14-bit core chips only).
- The programming document, describing how a programmer must handle the chip (not interesting unless you want to design your own programmer).
- For chips that are a direct successor of another chip there will often be a migration document that describes in detail the differences between the old chip and the new chip.
- Optionally, check for any application notes that might be relevant to your application.

The electrical characteristics chapter of the data sheet contains a section called absolute maximum ratings. These figures indicate the circumstances which a chip will survive for a limited (but unspecified) time. Don't interpret these ratings as normal operating conditions!

The electrical characteristics section in the datasheet describes the circumstances for which the chip is guaranteed to work. This does not mean that it necessarily won't work when one or more of these characteristics are violated, just that the manufacturer does not guarantee it. For a one-off hobby project you could exceed some characteristics (power supply voltage, ambient temperature, clock frequency), test your circuit, find it working, and use it. For a very large production run you could do the same, but with much more testing (and probably extensive re-testing whenever Microchip changes its production process). In between these extremes I would advise you to keep well within all characteristics.

Note that Microchip (in application notes, or in summarizing a chip’s features) often does not stick to this rule. You might interpret this as a permit to do likewise, but remember that the consequences are for you, not for them.

PIC IO pins have a primitive over-voltage / under-voltage protection, consisting of diodes to Vcc and Gnd. These diodes are certainly not meant to carry significant current during normal operation, and it is a continuing debate whether they are designed to carry any current at all during normal operation of the chip. It is reported that even a minimal current through these diodes can cause the A/D converters to behave very strangely. A funny consequence of these diodes is that when the power to a PIC is removed but one of its inputs is still high, it will be powered via the protection diode! This is not a guaranteed mode of operation, but it can ruin an attempt to reset the PIC by removing the power.

PICs are fairly robust against mistreatment, but I killed a few by applying the power in reverse. A power supply voltage well above the allowed 5.5V also been reported as an effective PIC-killer. To protect myself against this mistake I often put a 1N400x 'fools diode' between the power lines (to short the power if I ever accidentally apply it with reverse polarity), and I always use a 7805-based power supply (to limit the current in such a situation).

I put all my chips in a 'turned' chip socket to avoid breaking the chips flimsy legs. Some people have reported that when such a combo is put in a breadboard or (straight-pin) socket that board or socket can no longer reliably accept normal chips (because the springs have been forced to open too wide).

The radix (base) of a literal without an explicit radix specification is determined by an assembler option! Hence the only way to make sure that someone else can assemble your program (and get the same result) is to specify an explicit radix for each and every literal. To make sure that you have done so you could assemble with both (hex and decimal) defaults and check that the produced .hex file are the same.

Some things that appear to be PIC assembler instructions are actually macro's that can generate a number of instructions, not necessarily one. Funny things will happen when you try to skip such a macro-instruction: when it has generated more than one instruction you will skip only the first, when it has generated zero instructions you will skip the next instruction after the macro!

5 Using Hi-Tech Lite (= free) C compiler

5.1 Introduction

The MPLAB installation (I used MPLAB 8.40) contains a free C compiler. This is the Lite version of the Hi-Tech C compiler. It has code optimization disabled, which results in code that is roughly twice the size of the optimized version. The price for the fully licensed version of this compiler is some $1200, so I guess for now you will prefer the larger code size of the free version. (There are cheaper C compilers on the market, but not with an easy to use free version. There is even the GNU SDCC compiler, but as far as I know the PIC version is still under development, as it has been for years.) You can find a quick start guide (19 pages) and a manual (519 pages) in the installation directory (default is C:\Program Files\HI-TECH Software\PICC\PRO\9.65\doc). This chapter serves as a short introduction to using this C compiler on the DB038 board, and documents the C library provided for the DB038. Like all compilers and assemblers, the C compiler generates a .hex file, which can be downloaded to the DB038 board using the pickit2 application.

5.2 Project creation

You can create a new project using these steps:

1. Start MPLAB
2. Start **Project → Project Wizard**
3. Select the device (PIC16F887)
4. **Create New project File** : specify the project file, create it in a new (empty) directory (don't make the path name too long, and avoid spaces and weird characters in the path name)
5. You are prompted to add files to the project. Add the files that you already have, you can add other files lateron.
6. Finish the wizard.
7. Select **Project → Select Language Toolsuite..** Set the Active Toolsuite to HI-TECH Universal Toolsuite. The location should automatically be set to the path where the Hi-Tech suite is installed, if not you must set it manually.

You can add existing files by right-clicking on the appropriate entry (**Source Files** for C files) in the project window. A new file can be created by choosing **File → New**, and saving the file in the project directory. Note that after saving the file you must still add it to the project.

Select **Project → Build** (F10) or better: **Project → Rebuild** (ctrl-F10) to build your project: When you change header files, even those listed as header files in the files list, **Project → Build** does not seem to recompile the C files.

Once created you can start MPLAB and open the project by double-clicking the .mcp file.

5.3 Details for PIC programming

The integer sizes of the compiler are shown in the table below. The **bit** type is a non-standard integer type that can hold only 0 and 1. It is restricted to global variables, local static variables, and function return values; it cannot be the type of a parameter or local variable. The short long type is also a non-standard type (not found in normal C compilers). Remember that when the signed or unsigned designation is omitted, all integers default to unsigned, except for char, for which the default is compiler-dependent. So, except when you don't care (when you use ASCII values for instance), always explicitly specify **signed char** or **unsigned char**. The size of a **double** variable is by default 24 bits, but a compiler command line option can switch to 32 bits. I would not feel comfortable if my code depended on this setting, because it is not visible in the source.

Types	Size	Remarks
Bit	1 bit	Non-standard; see text.
(signed or unsigned) char	8 bits = 1 byte	
(signed or unsigned) short int	16 bits = 2 bytes	
(signed or unsigned) int	16 bits = 2 bytes	
(signed or unsigned) short long int	24 bits = 3 bytes	non-standard
(signed or unsigned) long int	32 bits = 4 bytes	
Float	24 bits = 3 bytes	
Double	24 bits = 3 bytes, or 32 bits = 4 bytes	see text

The PIC special function registers are directly available as global C variables. You can check the header file at C:\Program Files\HI-TECH Software\PICC\PRO\9.65\include\ pic16f887.h for the names. Bank switching and code pages are handled by the compiler. Variables with a **const** designation are stored in code memory (flash). There is much more flash in most PICs than RAM, so use **const** whenever you can. Variables are automatically placed in all RAM banks, but a variable cannot be 'spread' over more than one bank.

Bit manipulation can be done using the normal C method of or-ing (|) or and-ing (&) with a constant that has only one bit set or cleared, as shown (for TRISA and PORTA_SHADOW) in the example below. This is an example of bit manipulation, for real code using the macros in the DB038 C library is to be preferred. Note that the bits in the TRIS register must be 0 for outputs, hence the ^ 0xFF at the end.

Use the LEDs

```
// RA3 must be output, and must be high
TRISA &= ( 1 << 3 ) ^ 0xFF;
PORTA_SHADOW |= 1 << 3;
PORTA_FLUSH();

// RE0 .. RE2 must be outputs, value 4
TRISE &= 0x07 ^ 0xFF;
PORTE_SHADOW = 4;
PORTE_FLUSH();

// PORTD must be outputs
TRISD = 0x00;

// write a pattern to the LEDs
PORTD_SHADOW = 0x99 ^ 0xFF;
PORTD_FLUSH();
```

A PIC does not have a stack for data, only one for return values. This stack is limited, for a 16F887 it can hold 8 return addresses. The compiler generates a warning when it detects an attempt to exceed this limit, but I doubt that it will always be able to detect this problem, especially when function pointers are used. Recursion is also a problem on a PIC. The compiler seems to report this by the simple method of crashing (probably because it tries to find the number of stack entries used, which is of course infinite). The compiler seems to crash or enter an infinite loop in some other error situations too.

The compiler generates an .lst file. It shows a lot of information, have a look in it to see the (sometimes horrible) assembly code it generates from your C code.

5.4 C library

The C library for the DB038 can be found at the DB038 homepage at www.voti.nl/DB038. Its library files are summarized in the table below. (The example files are documented in the next section.)

The library is provided as source files. I suggest that you copy everything to your project directory, and include the files in your project as needed.

Files	Contain
db038.h db038.c	Configuration fuses, busy waiting, I/O port shadowing, nibble-to-char.
db038-adc.h db038-adc.c	Read the A/D converter.
db038-dallas.h db038-dallas.c	Access to Dallas one-wire peripherals.
db038-i2c.h db038-i2c.c	I_2C library.
db038-keypad.h db038-keypad.c	4 x 4 keypad reading.
db038-lcd.h db038-lcd.c	HD44780 character LCD initialization, clear, cursor position control, char print, string print, custom character definition.
db038-leds.h db038-leds.c	Show values on the seven segment displays and the LEDs.
db038-string.h db038-string.c	Character and string functions.
db038-uart.h db038-uart.c	UART initialization, character send and receive, string send.

5.4.1 Basic stuff

Header file	`db038.h`
Code file	`db038.c`

The header file includes the Hi-Tech PIC-specific include file. The C file sets the configuration fuses. The C file must always be present in your application. The header file is included by the other header files, so you can probably don't need to include it explicitly.

```
void wait_ms( unsigned int x );
```
This function waits x milliseconds. Note that the parameter is an unsigned int (16 bits) , so the maximum delay is 65 seconds.

```
WAIT_US( x );
```
This macro (it is not a function!) waits x microseconds. x must be a constant.

```
unsigned char PORTA_SHADOW;
void PORTA_FLUSH( void );
```

To avoid read-modify-write problems all changes to PORTA must be made by first making the change to PORTA_SHADOW, after which the macro PORTA_FLUSH must be called to copy the change to the port itself. Similar macros are provided for PORTB, PORTC, PORTD, and PORTE.

```
PIN_SET( port, bit );
PIN_CLEAR( port, bit );
```

These macro's set (or clear) the specified bit (LSB = 0) in the specified port's shadow register, and copy the shadow register to the port register.
Example: PIN_SET(PORTA, 0);

```
PIN_COPY( port, bit, bit_value );
```

This macro copies the bit_value to the specified bit of the shadow register of the specified port, and copies the shadow register to the port.
Example: PIN_COPY(PORTA, 1, RA0);

```
PORT_SET( port, byte_value );
```

This macro sets the specified port's shadow register to the specified byte_value, and copies the shadow register to the port.
Example: PORT_COPY(PORTE, 0x03);

```
void wait_ms( unsigned int x );
```

This function waits x milliseconds. Note that the parameter is an unsigned int (16 bits) , so the maximum delay is 65 seconds.

5.4.2 A/D converter

Header file	`db038-adc.h`
Code file	`db038-adc.c`

```
unsigned int adc_channel_read( unsigned char channel );
```

This function configures the specified A/D pin, performs an A/D conversion and returns the 10-bit (right-adjusted) result (value 0 .. 1203). The table below shows the relation between channel numbers and I/O pins.

Channel	Analog pin name	I/O pin name
0	AN0	RA0
1	AN1	RA1
2	AN2	RA2
3	AN3	RA3
4	AN4	RA5
5	AN5	RE0
6	AN6	RE1
7	AN7	RE2

Channel	Analog pin name	I/O pin name
8	AN8	RB2
9	AN9	RB3
10	AN10	RB1
11	AN11	RB4
12	AN12	RB0
13	AN13	RB5
14	CVref	-
15	Fixed Ref	-

```
unsigned int adc_ldr( void );
unsigned int adc_potmeter( void );
unsigned int adc_lm335( void );
unsigned int adc_tl431( void );
unsigned int adc_microphone( void );
unsigned int adc_battery( void );
unsigned int adc_current( void );
```

These functions configure the analog multiplexers, perform an A/D conversion, and return the 10-bit (right-adjusted) result (value 0 .. 1203) of the on-board peripherals.

5.4.3 Dallas one-wire

Header file	`db038-dallas.h`
Code file	`db038-dallas.c`

This library provides access to Dallas 1-wire peripherals.

```
bit d1w_present( void );
```

Check whether a one-wire peripheral is present. Calling this function can also be used to reset the one-wire devices.

```
void d1w_byte_send( unsigned char x );
```

Send one byte over the one-wire bus.

```
unsigned char d1w_byte_receive( void );
```

Receive one byte over the one-wire bus.

```
unsigned char d1w_family_code_read( void );
```

Read the family code of the connected one-wire device. This function can be used only when only one one-wire device is present on the bus.

```
void ds18s20_conversion_start( void );
```

Start conversion on any DS18S20 devices on the one-wire bus.

```
unsigned int ds18s20_temperature_read( void );
```

Read the raw temperature data (16 bits) from a connected DS18S20 device. This function can be used only when only one one-wire device is present on the bus. This function requires that a conversion has been started and completed.

5.4.4 I_2C

Header file	`db038-i2c.h`
Code file	`db038-i2c.c`

This library provides access to the I_2C bus. On the DB036 board this bus connects to the TCN75(A) temperature sensor and the EEPROM (24C01). The I_2C bus is also available on a connector, so chips can also be connected externally.

```
#define I2C_TCN75_ADDRESS 0x48
#define I2C_24C01_ADDRESS 0x50
```

These defines provide the I_2C addresses of the indicated chips. These addresses are without the R/W bit.

```
void i2c_read(
   unsigned char address,
   unsigned char data[],
   unsigned char n
);
```

This function performs a read operation on the I_2C bus. The address is the I_2C device address (without the R/W bit). Data is a pointer to the bytes that will be read. N is the number of bytes that will be read.

```
void i2c_write(
   unsigned char address,
   const unsigned char data[],
   unsigned char n
);
```

This function performs a write operation on the I_2C bus. The address is the I_2C device address (without the R/W bit). Data is a pointer to the bytes that will be written. N is the number of bytes that will be written.

5.4.5 Keypad

Header file	`db038-keypad.h`
Code file	`db038-keypad.c`

```
#define KEYPAD_0         0
#define KEYPAD_1         1
#define KEYPAD_2         2
#define KEYPAD_3         3
#define KEYPAD_4         4
#define KEYPAD_5         5
#define KEYPAD_6         6
#define KEYPAD_7         7
#define KEYPAD_8         8
#define KEYPAD_9         9
#define KEYPAD_A      0x0A
#define KEYPAD_B      0x0B
#define KEYPAD_C      0x0C
#define KEYPAD_D      0x0D
#define KEYPAD_S      0x0E
#define KEYPAD_H      0x0F
#define KEYPAD_NONE   0xFF
```

These macros identify the keypad keys.

```
unsigned char keypad_read( void );
```

This function reads the keypad and returns the first key it finds pressed or KEYPAD_NONE if no key is pressed.

```
bit key_is_pressed( unsigned char key );
```

This function returns true when the specified key is pressed, otherwise it returns false.

`char ascii_from_key( unsigned char x );`

This function returns the ASCII character corresponding to the key x.

5.4.6 LCD functions

Header file	`db038-lcd.h`
Code file	`db038-lcd.c`

`void lcd_init( unsigned char x, unsigned char y );`

This function initializes the I/O pins and the LCD itself. x and y must be the number of characters in X and Y direction. It is easy to swap x and y, so better use one of the LCD_INIT_ macro's instead.

`LCD_INIT_1_16`

This macro calls lcd_init to initialize a 1 line by 16 characters LCD.

`void lcd_command( unsigned char x );`

This function sends the command byte x to the LCD.

`void lcd_clear( void );`

This function clears the LCD and places the (invisible) cursor at the top left position.

`void lcd_line( unsigned char n );`

This function places the (invisible) cursor at the start of line n. The top line is line 0.

`void lcd_data( unsigned char x );`

This function sends the data byte x to the LCD.

`void lcd_char_print( char c );`

This function prints a single character to the LCD. The cursor is advanced one position. When the end of a line is reached, further characters are ignored. The table below shows the characters that have a special effect.

Character	ASCII value	effect
	0 .. 7	User-defined characters
'\n'		Newline: the cursor is placed on the first position of the next line. If the cursor was already on the last line further characters will eb ignored.
'\r'		Carriage return: the cursor is placed on the first postion of the current line. The current line is not cleared.
'\v'		Clear: the LCD is cleared and the cursor is placed at the top-left position.

`void lcd_string_print( const char *p );`

This function prints a string of characters, using lcd_char_print.

```
void lcd_custom(
   unsigned char n,
   unsigned char d1, unsigned char d2,
   unsigned char d3, unsigned char d4,
   unsigned char d5, unsigned char d6,
   unsigned char d7, unsigned char d8
);
```

This function defines user-defined character n (n = 0..7).

5.4.7 LED and seven-segment display functions

Header file	`db038-leds.h`
Code file	`db038-leds.c`

`SEGMENT_A  .. SEGMENT_G, SEGMENT_DP`

These macro's define the bit patterns for the individual segments

`SEVEN_0 .. SEVEN_F`

These macro's define the bit patterns for the numbers '0' .. 'F'.

```
void leds_show(
   unsigned char leds,
   unsigned char milliseconds
);
```

This function displays the specified pattern on the LEDs, for the specified time. Each bit in the leds parameter corresponds to one LED, the LSB corresponds to the rightmost LEDS. A LED is on when the corresponding bit is high.

```
void display_show(
   unsigned char n,
   unsigned char d,
   unsigned char milliseconds
);
```

This function shows pattern d on the seven-segment display n (0..3, 0 is the rightmost display), for the specified time. The pattern d can be composed by OR-ing or ADD-ing values from SEGMENT_A .. SEGMENT_DP, or by calling the function seven (see below).

```
void displays_leds_show(
   unsigned char d0,
   unsigned char d1,
   unsigned char d2,
   unsigned char d3,
   unsigned char leds,
   unsigned char milliseconds
);
```

This function shows the specified patterns on the four displays and the LEDs, for the specified time.

```
unsigned char seven_from_nibble( unsigned char x );
```

This function returns the seven-segment pattern corresponding to the value of the lower 4 bits of the parameter x.

```
void display_hexadecimal_show(
   unsigned int value,
   unsigned char points,
   unsigned char leds,
   unsigned char milliseconds
);
```

This function shows the specified integer value on the seven-segment displays in hexadecimal format, and the pattern leds on the LEDs, for the specified time. The lower 4 bits of the parameter points specifies which point segments will light up, the LSB corresponds to the rightmost point.

```
void display_decimal_show(
   unsigned int value,
   unsigned char points,
   unsigned char leds,
   unsigned char milliseconds
);
```

This function shows the specified integer value on the seven-segment displays in decimal format, and the pattern leds on the LEDs, for the specified time. The lower 4 bits of the parameter points specifies which point segments will light up, the LSB corresponds to the rightmost point. Note that the specified value can be too large, in which case only the lower four digits of the value will be shown.

5.4.8 Character and string functions

Header file	`db038-string.h`
Code file	`db038-string.c`

This file provides character and string functions.

```
char char_from_nibble( unsigned char x );
```

This function returns the ASCII character (0..9, A..F) corresponding to the value of the lower 4 bits of parameter x.

```
unsigned char str_len( const char *p );
```

This function returns the length of the string p.

```
bit str_equal( const char *p, const char *q );
```

This function returns whether the strings p and q are equal.

```
bit str_prefix( const char *p, const char *q );
```

This string returns whether the string p is a prefix of string q.

```
void str_copy(
   char *p, unsigned char s,
   char *q, unsigned char n
);
```

This function copies up to n characters from string q to string p. S is the size of string n.

```
bit char_in( char c, const char * s );
```
This functions returns whether character c appears in string s.

```
bit string_in( const char * p, const char * q );
```
This function returns whether the string p appears somewhere in string q.

5.4.9 UART functions

Header file	**db038-uart.h**
Code file	**db038-uart.c**

```
void uart_init_1200( void );
void uart_init_2400( void );
void uart_init_4800( void );
void uart_init_9600( void );
void uart_init_19200( void );
void uart_init_57600( void );
void uart_init_115200( void );
void uart_init_250000( void );
```
These functions initialize the UART at different baudrates.

```
bit uart_char_available( void );
```
This function returns true when a character is available, false otherwise.

```
char uart_char_receive( void );
```
This function waits for a character to become available, reads it and returns it. Note that this function will not return until a character is available. If you want to avoid such waiting, use the function uart_char_available to check whether a character is available, and call uart_char_receive only when one is available.

```
bit uart_send_idle( void );
```
This returns true when the transmitter is idle.

```
void uart_char_send( char c );
```
This function waits for the transmitter to become idle, and then starts transmission of the character. It does not wait for the transmission to complete.

```
void uart_string_send( const char *p );
```
This function uses uart_char_send to send a 0-terminated string of characters.

```
void uart_string_receive(
   char *received,
   unsigned char received_size,
   unsigned int milliseconds,
   const char * terminate,
   const char * ignore );
```

This function receives a string of characters. The pointer *received* must point to a caller-allocated string. Characters that are in the string *ignore* will be skipped. Receiving will terminate as soon as one of the following conditions holds:

- *received_size* - 1 characters have been received (not counting skipped characters, - 1 to have room for the terminating '\0'), or
- *milliseconds* milliseconds have elapsed, or
- A character that is in the string *terminate* has been received.

5.4.10 Examples

The library .zip contains a number of example programs (for each example, a C file and an MPLAB .mcp file).

Example file	What it does
main-adc	Reads the potentiometer and shows its value (0..1023) in decimal format on the seven segment displays.
main-ds18s20	Shows (on the seven segment displays) "----" when no one-wire device is connected, the device code when a non-DS18S20 device is connected, or the temperature when a DS18S20 is connected.
main-keypad	Reads the keypad and shows a pressed key on the leftmost seven-segment display; also checks whether this key is pressed and if so shows it on the next display.
main-lcd-custom	Shows how user-defined characters can be used to make a simple animation on a 16 characters by 2 lines LCD.
main-leds	Counts in binary format on the 8 LEDs.
main-seven	Counts in hexadecimal on the four seven-segment displays, in binary on the LEDs, and in binary (but slower) on the decimal points of the seven segment displays.
main-tcn75	Reads the TCN75 temperature register and shows it on the seven-segment displays. When powered by the USB the reading can be instable due to voltage fluctuations.
main-uart	Receives characters from the UART (19k2) and from the keypad, received characters are sent over the UART and their ASCII code is shown (in hexadecimal) on the seven segment display.
main-wavecom	Demonstrates communication with a Wavecom GSM modem.

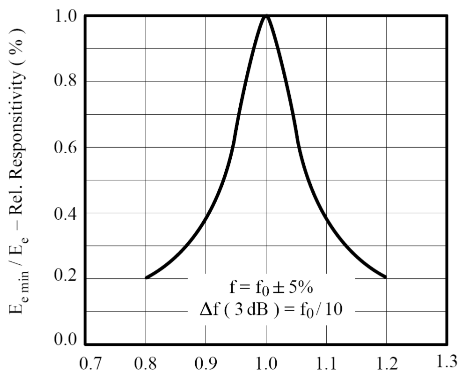
6 Background information

6.1 The RC5 infra-red remote control protocol

IR remote control units use modulated Infra-Red light to send information to for instance a television set. I guess they use light because using light is, unlike using radio waves, not subject to complex regulation. Infra-red light was used instead of visible light used because there are much less sources of infra-red light in the average living room than sources of visible light, and infra-red light will not blind a human eye.

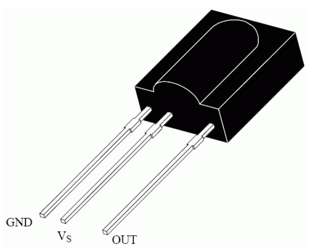
There are some domestic sources of IR light, most notable FL lamps, which produce a 100 Hz or 120 Hz (depending on the country) flicker. To minimize interference from these sources the remote control's IR light is modulated (switched on and off) at a much higher frequency. The most common IR protocols use 36 kHz (Philips) or 38 kHz (Sony). The matching receive module contains a band filter that rejects all signals that are not in the frequency band for which the received is designed. Hence there are different receivers for 36 kHz signals (for instance the TSOP1736) and 38 kHz signals (TSOP1738). The figure below shows the

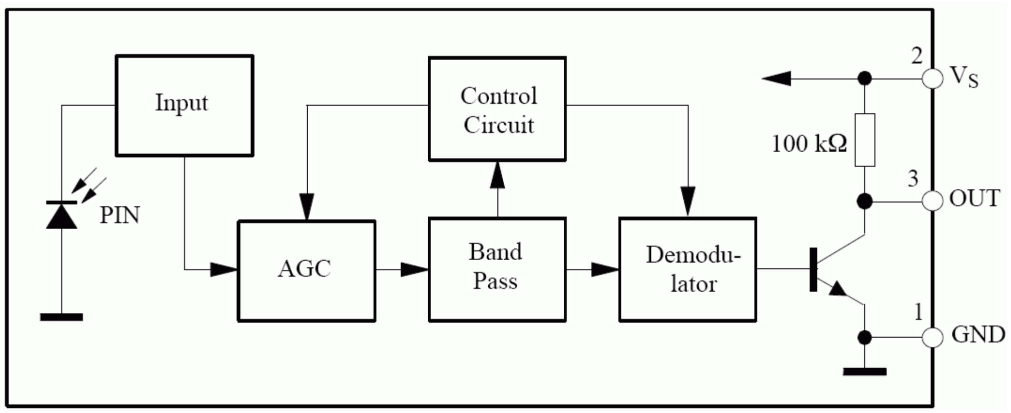
Attenuation as a function of the frequency deviation for a TSOP receiver.



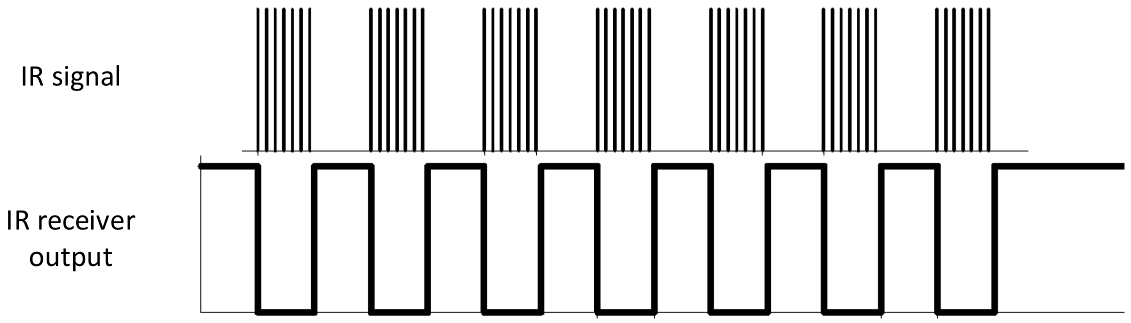
For optimal performance a receiver of the correct frequency must be used, but in practice the band filters in the receivers will accept a nearby frequency with some attenuation. The figure below show that a TSOP1736 receiver module will attenuate a 38 kHz signal (38/36 = 1.06) by 40% (the relative responsitivity is 60%), so the maximum distance is 77% (√0.6) compared to a 36 kHz signal: not much to worry about.

The pictures below show a typical IR receiver module and its internal block diagram.





The IR receiver module contains an AGC (Automatic Gain Control) unit that will amplify the incoming signal to a fixed level. This works best when the signal is present roughly 50% of the time. When the signal present more than 50% of the time (or even constant) the AGC will recognize is as 'too strong' and turn down the amplification until it seems to be present about 50% of the time. A typical recommendation from a receiver manufacturer is that a pulse lasts at least 400 µs (otherwise the receiver might not notice it), but on average (over a 10 ms interval) the signal must be present not more than 50% of the time. Note that in this context a signal is a 36 or 38 kHz 'tone'. The output of the IR receiver module is the signal without the modulation, and it is inverted. Hence the output is low when the modulated signal is presents, and it is high when no such signal is present, as shown in the figure below.

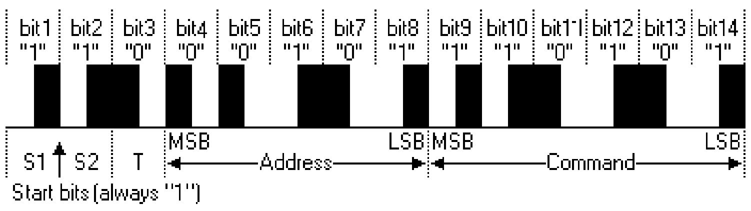


Manufacturers of television sets and the accompanying IR controls use various ways to encode information. The Philips RC5 code is the most common so it will be described here. Information about other codes can be found on the internet. The RC5 code uses bi-phase modulation to transmit a bit. A zero is transmitted as 889 µs signal followed by 889 µs silence, a one is transmitted as 889 µs silence followed by 889 µs signal. Note that this satisfies the requirements of a typical receiver: a pulse is more than 400 µs, and averaged over one bit the signal is present exactly 50% of the time. The signal is modulated at 36 kHz, so 889 µs of signal is 32 pulses. These pulses can have a duty-cycle of 50% (13.9 µs on, 13.9 µs off), but in practice the duty-cycle is often reduced to 1/3 (9.26 µs on, 18.5 µs off) or 1/4 (6.9 µs on, 20.8 µs off) to conserve the power consumption in the (battery powered) remote control.

RC5 message format

-

The black blobs are the modulated IR signal.



An RC5 message consists of a two start bits (always 0), a toggle bit, a five-bit address, and a six-bit command, both MSB (Most Significant Bit) first. The address identifies the type of device (TV set, CD player, etc). The command identifies the button (1, 2, volume up, etc). The message is sent (repeatedly) as long as the corresponding button on the remote is hold down. The toggle bit is inverted every time a button is pressed, so it allows the receiver to distinguish between a button being pressed rapidly or hold down continuously. The table shows the RC5 codes for some common devices and commands (buttons). An internet search will give you more.

address	Device
0	TV1
1	TV2
2	Videotext

address	Device
16	Audio 1 preamp
17	Tuner
18	Analog cassette

command	Interpretation
0	0
1	1
2	2

3	TV extension
4	Laser Vision player
5	VCR1
6	VCR2
7	reserved
8	SAT1
9	VCR extension
10	SAT2
11	reserved
12	CD-video
13	reserved
14	CD-photo
15	reserved

19	Adio 2 preamp
20	CD
21	Rack audio
22	DCC Magneto
23	Reserved
24	Reserved
25	CD writeable
26 .. 31	Reserved

3	3
4	4
5	5
6	6
7	7
8	8
9	9
16	Volume +
17	Volume -
18	Brightness +
19	Brightness -
20	Color saturation +
21	Color saturation -
22	Bass +
23	Bass -
24	Treble +
25	Treble -
26	Balance right
27	Balance left

The RC5 code has only 5 bits to identity the device. An extended version of the protocol uses the second stop bit as 6th bit of the address. This second start bit is a 1 in the basic RC5 code. The extended version replaces it by the inverse of the 6th address bit, so and extended-code receiver will conveniently interpret the 6th address bit of a basic RC5 message as 0.

Sending an RC5 signal is relatively simple, C pseudo-code is shown below. The rc5_bit_send function sends one bit according to the bi-phase encoding. The two for loops can each send 31 pulses of 14 µs light and 14 µs no light, but the first loop will switch the LED only when the bit that is to be transmitted is 0, the second loop will do so only when the bit is 1. Note that 14 µs is used as an approximation of 1000 / (2 * 36). The time used by the other instructions in the loop can probably not be neglected (1µs at 20 MHz is 5 PIC instructions, jumps count as 2), so in practice this 14 µs will probably have to be lowered a bit.

Send one bit using RC5 code

```
void rc5_bit_send( unsigned char x ){
   for( i = 0; i < 31; i++ ){
      if( ! x ) LED on
      wait 14 µs
      LED off
      wait 14 µs
   }
   for( i = 0; i < 31; i++ ){
      if( x ) LED on
      wait 14 µs
      LED off
      wait 14 µs
   }
}
```

The rc5_send function sends a full RC5 message.

Send a full RC5 message

```
void void rc5_message_send(
   unsigned char toggle,
   unsigned char address,
   unsigned char command
){
   rc5_bit_send( 1 );
   rc5_bit_send( 1 );
   rc5_bit_send( toggle );
   for( i = 0; i < 5; i++ ){
      rc5_bit_send( address & 0x10 );
      address = address << 1;
   }
   for( i = 0; i < 6; i++ ){
      rc5_bit_send( command & 0x20 );
      command = command << 1;
   }
}
```

Correct receiving an RC5 message is a bit more subtle, because the timing is controlled by the sender. On the bright side, 36 kHz modulation is removed by the receiver module, so that is one thing less to worry about. A naïve implementation of RC5 is shown below. It waits for the first edge (which is halfway the first start bit), then waits one-quarter of a bit cell (three quarters into the bit cell of the first start bit), samples the value, waits one bit cell, samples again, etc. Ideally all sampling moments are ¾ into each bit cell, so a high at that moment corresponds with a 1 and a low corresponds with a 0.

Receive one RC5 bit – simpleminded version

```
 unsigned char rc5_bit_receive( void ){
   // the receiver is active low!
   unsigned char x = ! receiver_output;
   wait 1.778 µs
   return x;
}
```

Once you have a single-bit receive function receiving a full RC5 message is easy:

Receive an RC5 message

```
unsigned char rc5_toggle, rc5_device, rc5_command;
void rc5_receive_message( void ){
   unsigned char i;
   rc5_device = rc5_command = 0;
   while( receiver ){}
   wait 444 µs
   rc5_toggle = rc5_bit_receive();
   for( i = 0;  i < 5; i++ ){
      if( rc5_bit_receive() ) rc5_device |= 0x10;
      rc5_device = rc5_device >> 1;
   }
   for( i = 0;  i < 6; i++ ){
      if( rc5_bit_receive() ) rc5_command |= 0x20;
      rc5_command = rc5_command >> 1;
   }
}
```

The problem with this simple approach to receiving a bit is that it is very sensitive to timing problems. Let's assume that the receiver's clock is 10% faster than the sender's. The RC5 message consists of 14 bits, so the sample point for the last bit (LSB of the command) is 13 (and some) bit cells after the rising edge of the first start bit. Assuming the sender and receiver are perfectly synchronized at that first edge, the receiver's sample moment will be 10% of 13 bit times earlier. The width of the IR pulse is ½ bit cell, so this will raise problems. You can calculate for yourself what the maximum clock difference is for successful communication. In practice IR remote controls often use a cheap ceramic resonator or even the build-in RC oscillator of a microcontroller. A good resonator has a maximum inaccuracy of 1%, the RC oscillator of a microcontroller can be much worse (5 or 10%).

This timing problem arises because the simple approach synchronizes at one point (the rising edge of the first start bit) and then tries to stay synchronized for all 13 bit cells. A better approach is to synchronize in each bit cell. Within the each bit cell there is only one point in time that can be used to synchronize on: the edge halfway the bit cell. Hence the rc5_bit_receive function below first samples the receiver and stores this info, just like the simple version. Next it waits half a bit cell, then it waits for an edge, and then it waits for a quarter of a bit cell.

Receive one RC5 bit - resynchronizing version

```
unsigned char rc5_bit_receive( void ){
   unsigned char x = ! receiver_output;
   wait 889 µs
   if( receiver_output ){
      while( receiver_output ){}
   } else {
      while( ! receiver_output ){}
   }
   return x;
}
```

The problem identified and solved here (maintaining synchronization) is central to all forms of serial communication. Asynchronous serial communication (what we – incorrectly – call RS232) solves this by using relatively accurate clocks, and resynchronization at the start of each character. Synchronous communication uses a separate clock line. Bi-phase or Manchester encoding as used in RC5 embeds clock information in the data stream, allowing the receiver to re-synchronize at each bit, at the cost of one extra edge per bit cell. Hard disks often use an encoding that a compromise between asynchronous communication and bi-phase.

8. Test application

The DB038-test-102.hex file contains an application that can be used to test most of the hardware of a DB038 board. This description applies to version 1.02 of this application. The source of this test application is not available.

The tests show information on the LCD (either a 1x8 or a 2x16 LCD can be used), but can be used without the LCD. Some tests use the UART, but only the UART test itself requires that a PC is connected with a suitable terminal program running. The serial settings are 4800 baud, 8 data bits, no parity. Note that the virtual COM port number that created by Windows can vary, depending for instance on the USB connector used. You can use the device manager to find out which port is created (and if you want you can change it).

The various tests are selected by pressing a switch of the on-board keypad. Except when noted otherwise, the reset button must be pressed to end the test.

1 : BEEP : Press key 1. Three beeps will be generated on the on-board small speaker, or on an external speaker, when one is connected. After the three beeps the test will end (and a new test can be selected).

2 : KITT : Press key 2. A Kitt-style pattern will be shown on the LEDs. After three back-and-forth’s the test will end (a new test can be selected).

3 : SEVEN : Press key 3. The seven-segments will be tested: first all segments of each digit will be activated, next each segment of all displays will be activated. The LCD backlight will also blink. After three cycles the test will end (a new test can be selected).

4 : POT : The 8-bit analog value read from the potentiometer will be shown on the two right-side seven-segment displays, and on the LEDs. The value shown on the right-side displays should be proportional to the setting of the potentiometer, from 00 (fully anti-clockwise, half past seven) to FF (fully clockwise, half past five). The two left-side seven-segment displays count up in hexadecimal mode.

5 : LDR : The 8-bit analog value read from the LDR is shown on the two right-side seven-segment displays, and on the LEDs. When the LDR is shielded from light the value should be high (0xF0 or more), when the LDR is illuminated the value should be lower. The actual values can vary, depend on the lighting conditions and the type of LDR fitted on the board. The two left-side seven-segment displays count up in hexadecimal mode.

6 : LM335 : The 8-bit analog value read from the LM335 temperature sensor is shown on the two right-side seven-segment displays, and on the LEDs. The value that is displayed depends both on the temperature and on the power supply voltage of the PIC. At room temperature I see values in the range 0x90 .. 0x9F. Heating the LM335 temperature sensor (tip of soldering iron) should produce higher values (for instance up to 0xA0). The two left-side seven-segment displays count up in hexadecimal mode.

7 : TL431 : The 8-bit analog value read from the TL431 voltage reference is shown on the two right-side seven-segment displays, and on the LEDs. The value that is displayed depends both on the power supply voltage of the PIC. A normal value would be 0x7D. The two left-side seven-segment displays count up in hexadecimal mode.

8 : V-EXT : The 8-bit analog value read from the external power supply (wall-wart) is shown on the two right-side seven-segment displays, and on the LEDs. The value that is displayed depends both on the

voltage provided by the external power supply. For a 9V wall-wart a normal value would be 0x51, for a 12V wall-wart 0x7E. The two left-side seven-segment displays count up in hexadecimal mode.

9 : MOTORS : The two motors are driven: first each motor separately in each direction, then both motors together, both in the same direction and in different directions. The LEDs show the digital testing of the motor driver pins. The seven-segment display shows the motor current on the two right-side displays (raw A/D value, in hexadecimal). After three cycles the test terminates.

A : IR-RECEIVE : The IR receiver is polled. When a signal is received this is shown on the LEDs. The two left-side seven-segment displays count up in hexadecimal mode.

B : KDB-UART : When a key is pressed, its value is shown on the LCD and on the rightmost seven-segment display, and its transmitted by the UART at 4800/N/1. Each character sent from the PC to the board is echoed to the PC, along with its hexadecimal value. This test mode can be left by typing a Q at the PC's terminal program, or by a reset of the PIC.

C : MIC : The microphone input is read and the value is shown on the LEDs in a more or less logarithmic form, and on the two right-side seven segment displays. The normal background noise should light on the rightmost four LEDs. Speaking directly into the microphone should light more LEDs. Note that on the B01.01 boards the silkscreen marks Mic and Lsp are wrong: the microphone connector is next to the cinch (video) connector.

D : IR-REFLEX : The IR transmit LED sends an IR signal, and the IR receiver is polled. When a signal is received, the LEDs are lighted. Without any IR shielding this will cause the LEDs to always light up. I use an 8 cm black shrink tube over the send LED to block direct IR light from the LED to the receiver. Now when I put my hand or another object at 20 cm distance it will reflect the IR light sufficiently to light up the LEDs.

* : VIDEO : A crude 3x3 checkerboard video signal is generated.

# : no test

0 : FACTORY TEST : This test performs the following sub-tests. It will halt with an error message on the seven-segment displays when a sub-test fails.

1. The speaker beeps twice.
2. The kitt pattern is shown once (back and forth).
3. The seven-segment test cycle is shown once.
4. The potentiometer A/D value is read and checked. It is assumed that the potentiometer is set at half past ten.
5. The LDR, LM335 and TL431 A/D values are read and checked.
6. The external voltage A/D value is read and checked. An external 12V supply must be connected. Without an external supply the error message "E4" will be shown.
7. The motor driver circuit is tested. This requires that a dummy load is connected, 50 Ω for the 'upper' motor, 100 Ω for the 'lower' motor.

When all sub-test are completed successfully the seven segments display will show the message "good" and the LEDs will show a mirrored 4-LED kitt display. Subsequently the B : KBD-UART, C : MIC, D: IR-REFLEX and * : VIDEO tests should be performed.

7 Suggested lessons, assignments and projects

7.1 *A short course in PIC assembler*

This is an outline of a PIC assembler course I teach at the Hoogeschool Utrecht:

Lesson 1 : General introduction, PIC byte instructions, bit set/clear, difference between literals and memory addresses.

Lesson 2 : Skip instructions, GOTO/CALL/RETURN, macro’s, use of MPLAB, simulation, timing.

Lesson 3 : Delay tricks, banking, read-modify-write issue, DB038 hardware overview, use of pickit2 application.

Lesson 4 : using one 7-segment display.

Lesson 5 : Multiplexing 7-segment displays (and LEDs), reading the keypad.

Lessons 6, 7 : code paging. Optional subjects: switch bounce, A/D conversion, serial communication, IR, music, UART.

The subjects for lessons 5, 6 and 7 can be dealt with quickly. In the remaining time (and some extra time in their project week) the students must complete a self-defined project.

7.2 *Simple assignments*

1. Blink a LED
2. Kitt-style LED display (use shadow registers for the ports!)
3. µs delay routine: delay W µs
4. ms delay routine: delay W ms
5. conversion macro: return R if W equals X (otherwise do not change W!)
6. BCD to seven-segment value
7. Multiply subroutine
8. Beep at 1 kHz (requires an accurate delay)
9. Read A/D and display the raw values (potmeter, LDR)
10. Display a value on 1 seven-segment display (requires BCD to seven-segment conversion)
11. Display a 4-digit value on 4 seven-segment displays (time multiplexing)
12. Read the keypad

7.3 *Intermediate projects*

1. Play a melody
2. Audio ‘beat’ detector (disco jewel)
3. Display a text on a HD44780 character LCD (be sure to get and use the correct 4-bit initialization sequence)
4. Show a processed A/D value (temperature, external supply voltage)
5. Use the UART to communicate with a PC
6. Beep and play a melody using a PWM melody
7. Simple calculator (+,-)
8. VU meter on the LEDs (spot-peak and bar-current)

9. Send IR-remote codes (RC5, Sony, etc)
10. Interface to one Dallas one-wire chip
11. Make an access-control system (enter pin code on keypad, can be changed also)

7.4 *Advanced projects*

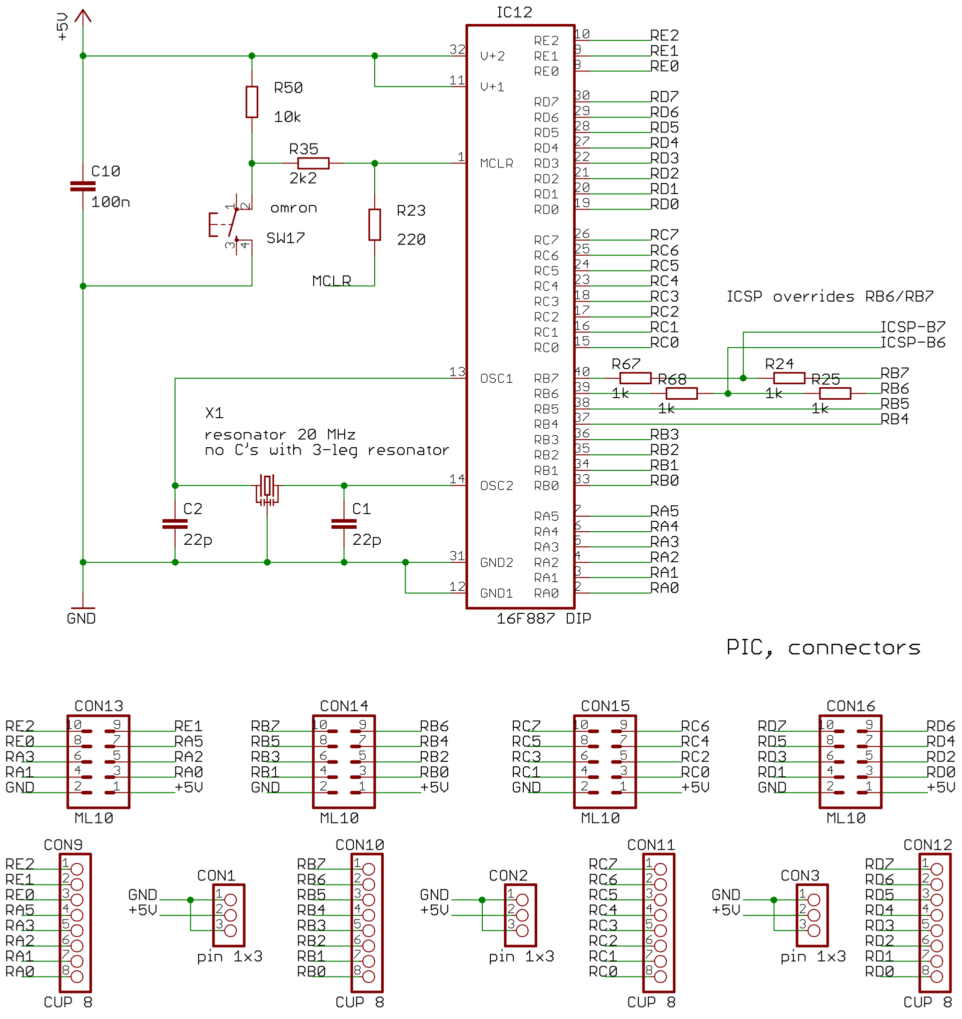
1. Use the I2C EEPROM
2. Use the I2C temperature sensor
3. Complex calculator (+,-,*,/, hex←→ dec, sine, cosine, etc)
4. Enter melody on keypad, store in internal or external EEPROM, play on request
5. Interact with PC: show waveform from A/D converter (simple oscilloscope)
6. Interface multiple Dallas 1-wire chips
7. Connect and interface to external SPI chips (HC595, MCP23S17)
8. Connect and interface to external I2C chips (PCF series, , MCP2317)
9. Receive IR remote control codes, show on displays, log to PC
10. IR cheater: whenever the TV is set to channel 5, immediately change it to channel 6
11. Interface to a PS/2 keyboard and/or mouse
12. “Dump” a .wav from the PC to the board, play it using PWM
13. Record sound from the microphone, pass it to the PC, store it there for later replay (via the board)
14. Cricket: make cricket-like sound in the dark, but be silent in the light
15. Scared-of-sound robot
16. Line-following robot (probably requires 2 external sensor, like CNY70)
17. Light-seeking robot
18. Object-avoiding robot (use IR-reflex to detect objects)
19. Data logger: record daylight (maybe you can even distinguish clouds from sunshine), temperature, maybe add other sensors like humidity, log everything to a PC
20. Create composite video
21. Video game: PIC pong
22. PC program parses a B/W picture and translates it to code that displays that picture as video
23. Advanced access control system: keypad, external keypad, master code, log to PC
24. Lap counter and/or speed display for model train or model racetrack, use IR reflex for detection, or add external CNY70’s
25. Magic wand message device – you will probably need to add a tilt switch or another device to synchronize the message with the movement
26. A PC application converts a ringtone file to a PIC (include) file that plays the ringtone. Extra: multiple melodies, played at random or on command from the keypad or an IR command.
27. Guitar tuner. Microphone is the input, string is selected by the keypad, deviation is shown on the LEDs. You will probably need to implement a DFT.\
28. MIDI player
29. Automatic light controller: button press or presence of people switch light on (IR reflex), but only at night (LDR), light goes off after a fixed time (use potentiometer) or button press, light can be simulated by a LED, or connect an external lamp suing an SCS, parameters can eb set from a PC, log to a PC.

7.5 *Other suggestions – not DB038-specific*

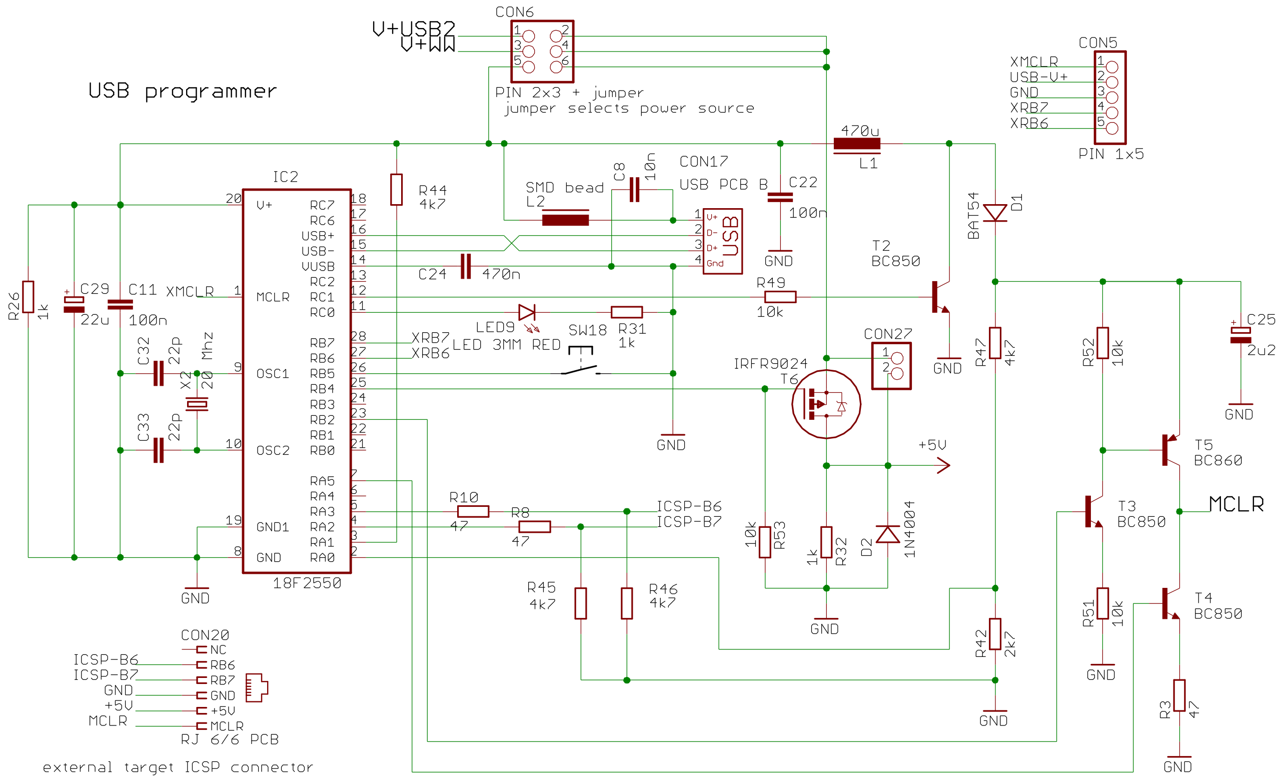
1. Interface to a graphics LCD
2. Interface to a matrix LED message display
3. Traffic light system: use external 74HC595 chips to interface to LEDs. Simple: two crossing roads. Extra: add pedestrians, multiple lanes, sensors for cars, buttons for pedestrians, emergency vehicles (IR?), timing parameters adjustable from a PC.
4. Video overlay. Add an LM1881 for detecting the sync signals.

8 Circuit diagrams

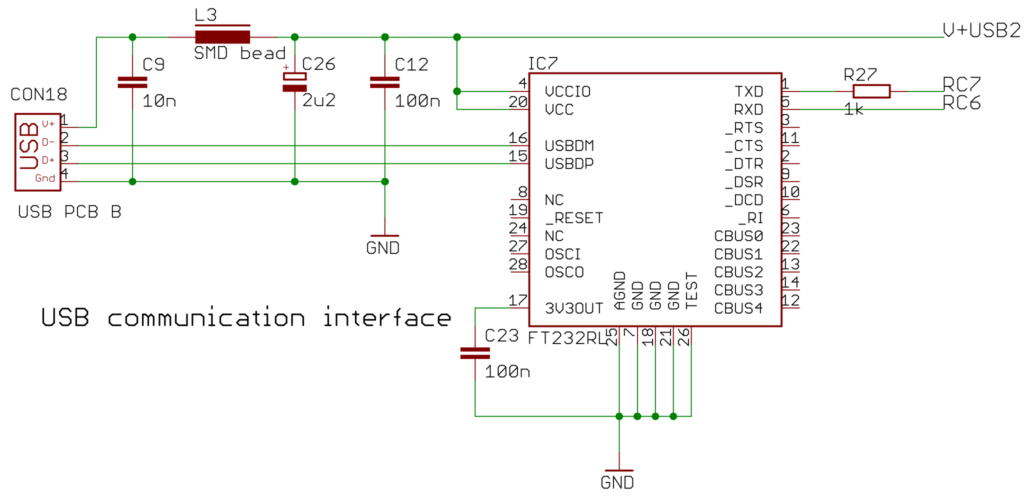
8.1 *PIC 16F887, ML10 connectors, wire cups*



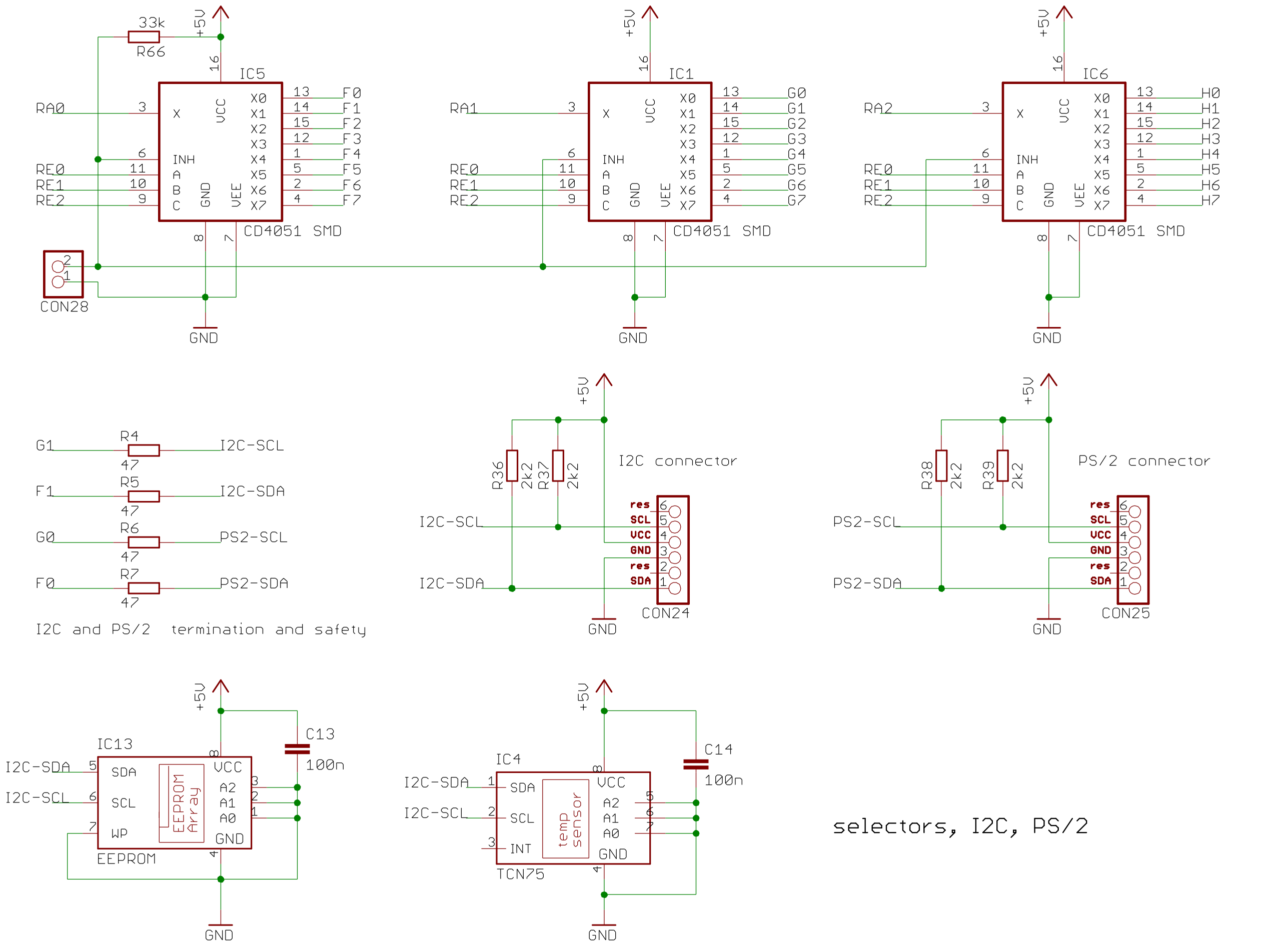
8.2 USB Programmer



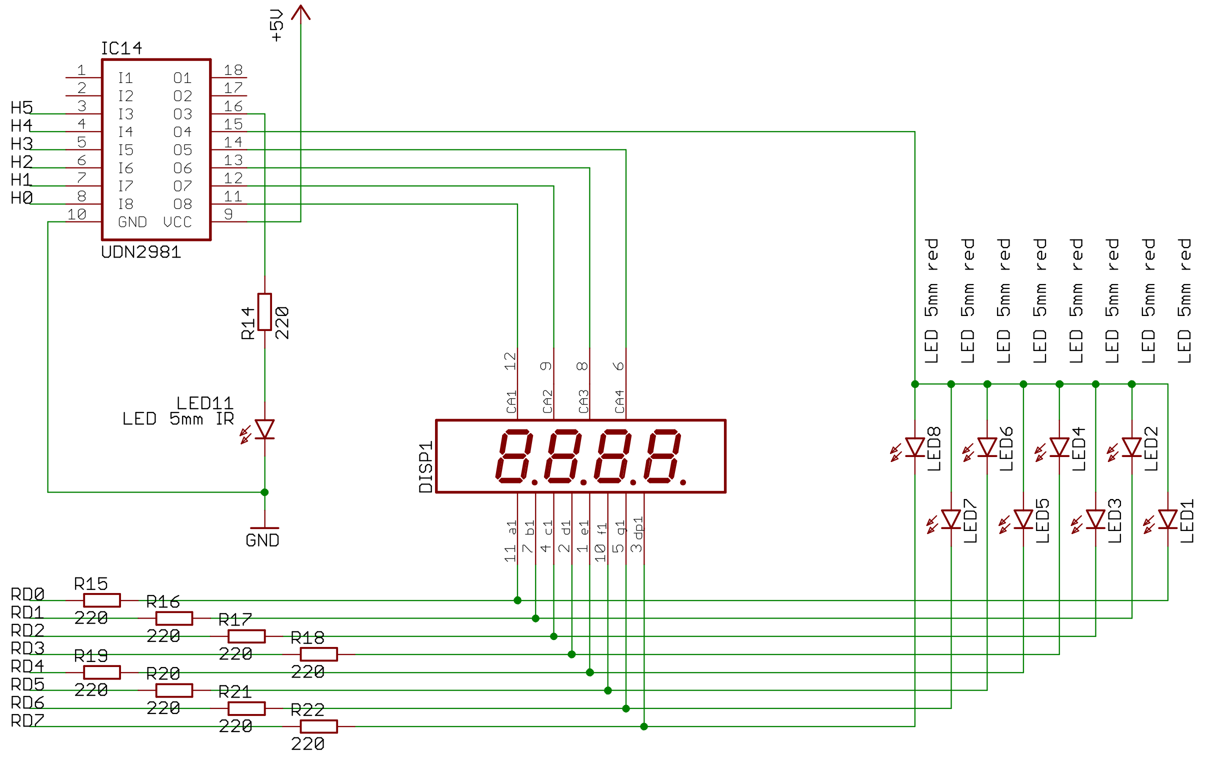
8.3 USB-serial interface



8.4 Multiplexors, PS/2, I2C

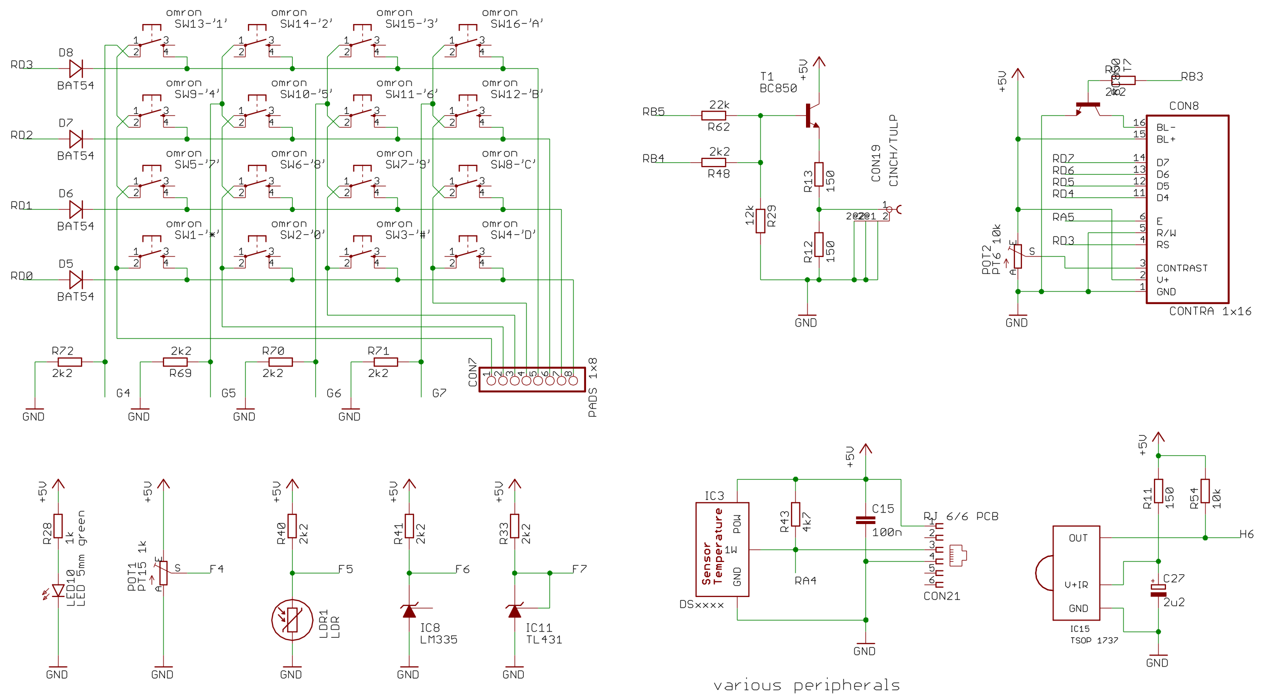


8.5 LEDs and seven-segment displays



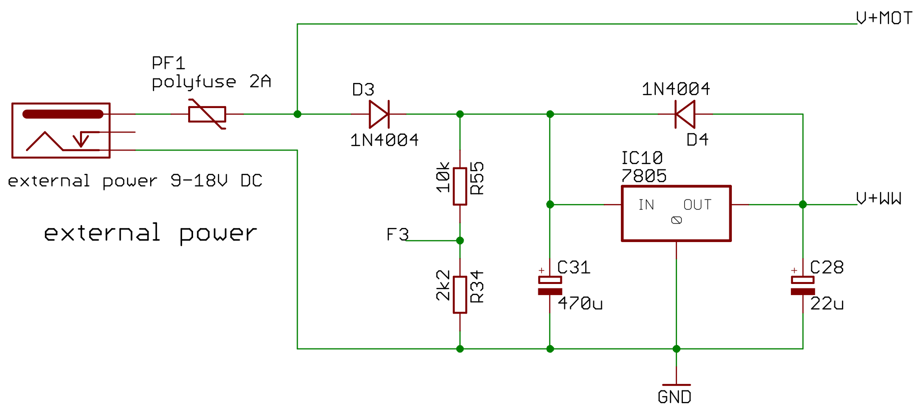
Diodes, IR-diode, 7-segment displays

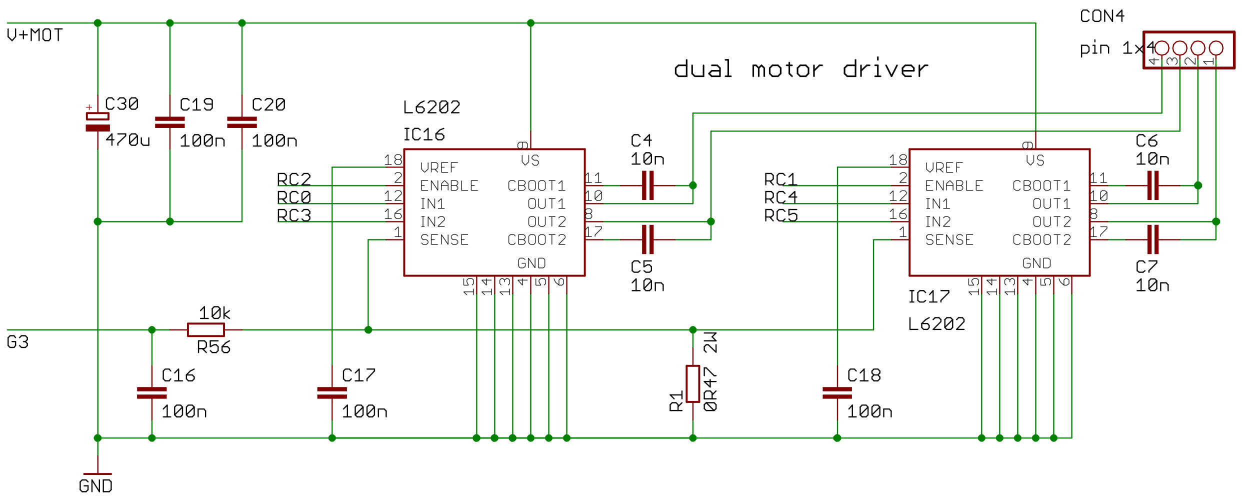
8.6 Miscellaneous interfaces



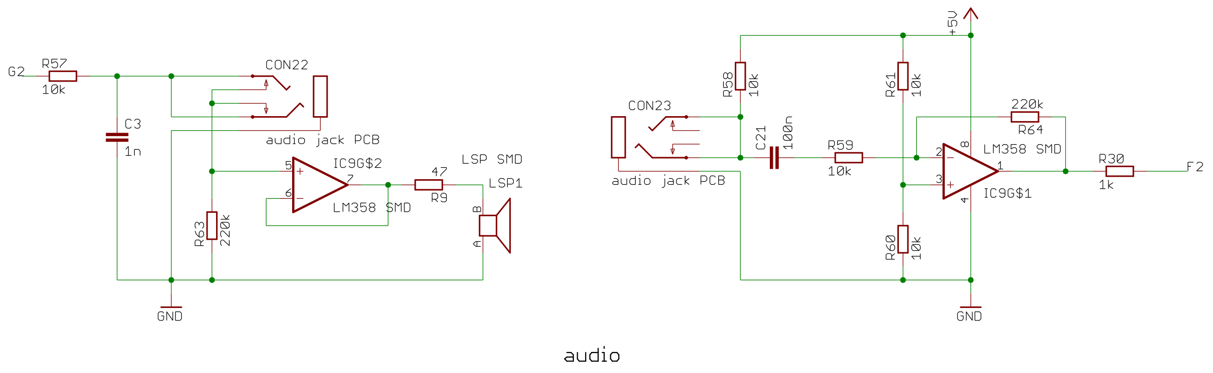
various peripherals

8.7 H-bridges and external power





8.8 Audio interfaces



9 External links

Microchip website	http://www.microchip.com/
PIC 16F887 datasheet	http://ww1.microchip.com/downloads/en/DeviceDoc/41291F.pdf
PIC 16F887 errata	http://ww1.microchip.com/downloads/en/DeviceDoc/80302E.pdf
Pickit2 1.20 firmware	http://ww1.microchip.com/downloads/en/DeviceDoc/PK2V012000.zip
Pickit2 1.20 application (also contains the user manual)	http://ww1.microchip.com/downloads/en/DeviceDoc/PICkit%202%20V1.20%20Setup.EXE
MPLAB	http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en019469&part=SW007002
FT232RL datasheet	http://www.ftdichip.com/Documents/DataSheets/DS_FT232R_V204.pdf
FT232RL VCP driver	http://www.ftdichip.com/Drivers/VCP.htm
CD4051 datasheet	http://www.fairchildsemi.com/ds/CD%2FCD4052BC.pdf
24C01 EEPROM datasheet	http://www.atmel.com/dyn/resources/prod_documents/doc0180.pdf
TCN75 datasheet	http://ww1.microchip.com/downloads/en/DeviceDoc/21490C.pdf
UDN2981 datasheet	http://www.allegromicro.com/en/Products/Part_Numbers/2981/2981.pdf
DS18S20 datasheet	http://datasheets.maxim-ic.com/en/ds/DS18S20.pdf
TSOP173x datasheet	http://www.voti.nl/docs/TSOP17.pdf
uA7805 datasheet	http://www.ti.com/lit/gpn/ua7805
L6202 datasheet	http://www.st.com/stonline/products/literature/ds/1373.pdf
LM358 datasheet	http://www.national.com/ds/LM/LM158.pdf

10 Document change notes

version	Date	Notes
2.00	2010-04-23	Adapted for board 1.05; IR explanation added
1.01	2009-11-13	Hitec C libraries chapter added
1.00	2009-08-28	General rewrite; EPS pictures
0.1	2008-09-15	First version, adapted from the DB037 document