Jallib starters guide

An introduction to JAL V2 & Jallib

Version: 1.01

Author: Joep Suijs, with contributions of Toon Peters and Wayne Topa.

Contents

Introduction

JAL is a high-level language for PIC microcontrollers[1] and originally developed by Wouter van Ooijen. JAL V2 (at this time of writing, the current version is V2.4n) is a re-write of the compiler, by Kyle York. JAL 2.0 shares the original JAL syntax and adds new features to it, like new types and arrays. The JAL distributions contain a Windows and a Linux executable. The JAL compiler is open source, so the source-code is available if you want to use it on another platform.

When you use JAL, you also need a set of libraries. You could of course create your own (and some people argue you should), but there is also a set of open source libraries called Jallib. It provides much functionality and supports over 300 different PIC microcontrollers. The Jallib team maintains and extends the libraries.

The aim of the Jallib team is to provide a clear, consistent set of libraries. Much of the behavior of the libraries is configurable in the way that you, as a user, understand the way the library is configured – there are no defaults in most cases (and if it has, you can always bypass them and specify your own values). This makes the libraries flexible, versatile and more powerful. As a consequence, since there are no defaults in Jallib, you'll have to actually think about what values you require. This makes the first learning step higher, it takes more time. But we at Jallib believe that good documentation with ready-to-use samples can help you getting started. It's not that hard...

This guide is a 'stroll through JAL and Jallib[2]', it provides you an introduction to the use of JAL and Jallib. Together with the samples and documentation that come with Jallib and with the tutorial book, it gets you up to speed quickly.

Links

The JAL starting point	http://justanotherlanguage.org/
Jal mailing list (jal user list)	http://tech.groups.yahoo.com/group/jallist/
Microchip site (with datasheets)	http://www.microchip.com/
Jallib	http://code.google.com/p/jallib/
JalEdit, a JAL IDE for Windows	http://jal.sunish.net/jaledit
PicShell , a IDE with simulator for windows and Linux	http://picshell.ovh.org/ps/
Jaluino, development of a platform-independent IDE and Arduino-alike hardware	http://jaluino.org
Jallib developers list	http://groups.google.com/group/jallib
The JAL compiler homepage[3]	http://www.casadeyork.com/jalv2/

[1] Microcontrollers are 'single chip computers': a CPU, ram memory, reprogrammable flash memory and peripherals like I/O ports, timers, usart and ADC, all on one chip. Microchip calls its broad range of these chips 'PIC microcontrollers'.
[2] The style is inspired by the book 'Learning Perl' by Randal L. Schwartz.

Prerequisite

The aim of this document is to introduce you to JAL and Jallib. It shows you a lot of the JAL language and the most common parts of Jallib, so you learn the Jallib philosophy and practical information at the same time.

If you have developed software for embedded systems before, this guide is a good introduction for you.

If you're new to embedded software development, this documents guides you. But since it is a bit brief, play with the examples shown and search on the referenced resources to get more detailed information.

This guide is not about hardware. It assumes have a working development environment: you have the compiler and Jallib up-and-running, can convert a sample file into a hex file, program your target system and execute it.

If you've learned a programming language before, you may know the "hello world" tradition: start with the simplest program that prints 'Hello world' on the screen, so you actually know everything is working. But printing on a screen is far from simple with microcontrollers and the "hello world" equivalent – the simplest program to check everything is working – for a microcontroller is "Blink-a-LED".

And how you do this? Basically:

- Setup your PIC hardware. Although this guide is not about hardware, we feel we can't leave it out all together. To blink a led, you need a led connected, like shown in figure 1. Appendix A gives you the full schematics diagram of our evaluation hardware. Please look at www.justanotherlanguage.org if you want more details on hardware or other blink-a-led related issues.

- Download the Jallib pack for your operating system from http://code.google.com/p/jallib/ and install it on your computer.

- Take 16f877a_blink.jal from the sample subdirectory.

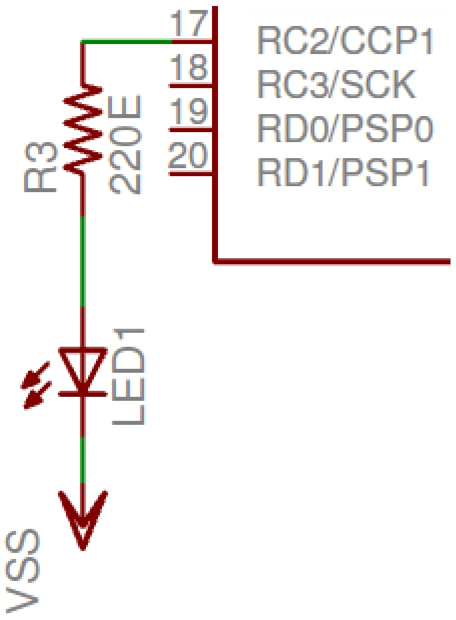


Figure 1

- Assuming you have the hardware setup of appendix A, change the pin from:

```
-- You may want to change the selected pin:
alias   led       is pin_A0
pin_A0_direction =  output
```

[3] Jallib distributions include the compiler and related documents. But we can't give you a list of jal-related links without this one!

To:

```
-- You may want to change the selected pin:
alias   led      is pin_C1
pin_C1_direction =  output
```

- Compile the sample.

- Program your target system and verify if the LED is blinking.

If you need more help, take a look at http://justanotherlanguage.org/content/tutorial_blink_a_led.

Getting started – microcontroller configuration.

The Jallib pack contains, amongst others, a directory with ready to compile samples. These samples are generally small programs that show the use of a specific library on a specific PIC.

For all PICs supported by Jallib, there is a blink example. And as described, we assume you used this to get your led blinking. Let's take a closer look to the source code of the blink-example.

The first part of 16f877a_blink.jal, the blink example for 16f877a:

```
01  -- -----------------------------------------------------
02  -- Title: Blink-an-LED of the Microchip PIC16f877a
03  --
04  -- Author: Rob Hamerling, Copyright (c) 2008..2009,
                                all rights reserved.
05  --
06  -- Adapted-by:
07  --
08  -- Compiler: >=2.4m
09  --
10  -- This file is part of jallib  (http://jallib.googlecode.com)
11  -- Released under the BSD license
              (http://www.opensource.org/licenses/bsd-license.php)
12  --
13  -- Description:
14  -- Sample blink-an-LED program for Microchip PIC16f877a.
15  --
16  -- Sources:
17  --
18  -- Notes:
19  --  - File creation date/time: 29 Jan 2009 08:41:14.
20  --
21  -- -----------------------------------------------------
22  --
23  include 16f877a                    -- target PICmicro
24  --
25  -- This program assumes a 20 MHz resonator or crystal
26  -- is connected to pins OSC1 and OSC2.
27  pragma target OSC HS               -- HS crystal or resonator
28  pragma target clock 20_000_000     -- oscillator frequency
29  --
30  pragma target WDT  disabled
31  pragma target LVP  disabled
32  --
33  enable_digital_io()                -- disable analog I/O (if any)
34  --
```

Note1: Line numbers are not included in program but used just for explanations.

Note2: all file name characters with Jallib are in lower case. So not 16f877A.JAL but 16f877a.jal.

The first 21 lines of the example are comments in JAL terms (all text on a line beyond '-- ' or ';' is ignored by the compiler) . Each Jallib file has a header with the same structure. It contains author information and information used to generate the JalApi documentation[4].

For the compiler, the real work starts at line 23:

```
23  include 16f877a                    -- target PICmicro
```

At this line, the device file is included. Jallib provides device files for over 300 PIC microcontrollers, all created and maintained by Rob Hamerling. The device file specifies the chip characteristics like the name and location of registers, bits within the registers, start and end of various memory blocks etc.

In addition to this, the device file provides:

- Definition of some constants (like true, false, high, low etc)
- Access to the digital I/O pins (we get to this shortly)
- A procedure called enable_digital_io().

The procedure enable_digital_io() is typical for the Jallib philosophy. Most PICs with an ADC (Analog to Digital Converter) have this ADC enabled by default. Since Jallib doesn't change things unexpectedly[5], it leaves this ADC enabled if this is the default.

However, in many cases people want to use digital I/O on all pins, which requires the ADC to be disabled. But there are different types of ADC, which need different code to disable. And there are PICs that don't have an ADC. In other words: it is PIC-dependant what action is required to assure digital I/O is enabled on all pins.

So each device file supplies the procedure enable_digital_io() and implements it in the way it is required for that specific PIC.

Hence no automagic code, but an universal API to achieve what you want.

And for the record: line 33 contains the call to ensure digital I/O works on all pins[6]:

```
33  enable_digital_io()                -- disable analog I/O (if any)
```

[4] JalApi is the application programming interface specification of Jallib, specified per library. It describes how each library can be configured and how it is used. JalApi is part of the Jallib distribution and is also online available at http://jallib.googlecode.com/svn/trunk/doc/html/index.html

[5] Using Jallib device files means you get what is described in the Microchip datasheets, So by reading the datasheet the defaults, e.g. for the fused. Also names (of registers, bit fields etc) are as close as possible to the name in the datasheet.

[6] Digital I/O works on most pins of your PIC without calling enable_digital_io(). The procedure is used to configure pins that can be used as analog inputs to work properly as digital inputs.

Pragma

Pragmas are compiler directives. It does not generate code, but (in most cases) provides information to the compiler how to generate code. If you had a look at the device file mentioned before, you have seen that pragmas are used to tell the compiler what type of PIC you use and how its memory is organized. Complicated stuff from the datasheet, which is handled for you in the device file.

But for a given PIC there are still configuration options left, like the 'configuration bits' or 'fuses'. These are documented in the PIC datasheet and vary from device to device.

Lines 27 to 31 contain pragma statements to configure our 16f877a:

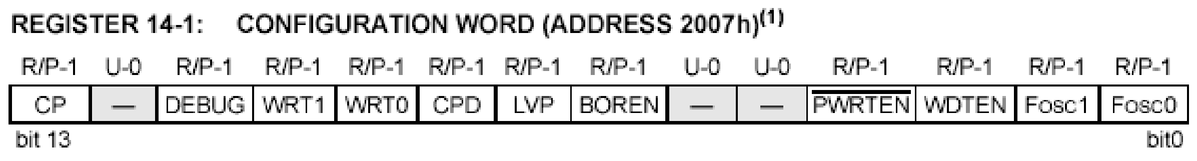
```
27  pragma target OSC HS                -- HS crystal or resonator
28  pragma target clock 20_000_000      -- oscillator frequency
29  --
30  pragma target WDT  disabled
31  pragma target LVP  disabled
```

First, look into line 28. On this line, the frequency of the oscillator is specified. This value is not used to configure the PIC but used by the compiler and its libraries. For instance to calculate parameters for the requested baudrate of the serial port or to generate the proper code for a requested delay. And if you ever want to use it: the clock parameter value is available as TARGET_CLOCK.

Note the underscore '_' within the speed definition. This can be used in any numeric constant you specify to make your code easier to read. The underscores within the numeric constant are ignored by the compiler.

The short version of line 27, 30 and 31 is it sets the oscillator type to high-speed (HS) and disables watchdog (WDT) and low voltage programming (LVP). If you want to learn more about these and other chip configuration options, read on. If not, you can skip the next section and go to the chapter on digital I/O ports.

The fuses are documented on page 143 of the PIC 16F877A datasheet:



REGISTER 14-1: CONFIGURATION WORD (ADDRESS 2007h)[1]

R/P-1	U-0	R/P-1	R/P-1	R/P-1	R/P-1	R/P-1	R/P-1	U-0	U-0	R/P-1	R/P-1	R/P-1	R/P-1
CP	—	DEBUG	WRT1	WRT0	CPD	LVP	BOREN	—	—	$\overline{\text{PWRTEN}}$	WDTEN	Fosc1	Fosc0
bit 13													bit0

And selection of the details:

bit 7 LVP: Low-Voltage (Single-Supply) In-Circuit Serial Programming Enable bit
1 = RB3/PGM pin has PGM function; low-voltage programming enabled
0 = RB3 is digital I/O, HV on MCLR must be used for programming
bit 6 BOREN: Brown-out Reset Enable bit
1 = BOR enabled
0 = BOR disabled
bit 5-4 Unimplemented: Read as '1'
bit 3 PWRTEN: Power-up Timer Enable bit
1 = PWRT disabled
0 = PWRT enabled
bit 2 WDTEN: Watchdog Timer Enable bit

1 = WDT enabled
0 = WDT disabled
bit 1-0 FOSC1:FOSC0: Oscillator Selection bits
11 = RC oscillator
10 = HS oscillator
01 = XT oscillator
00 = LP oscillator

On line 27 of the blink sample, we see that OSC is set to HS. From the datasheet we learn this sets the lower two bits to '10'[7]. On line 30 we disable the watchdog (clear bit 2) and on line 31 we disable 'low voltage programming' (clear bit 7).

The brownout bit is untouched in our sample program, so with the Jallib tradition, you can assume it keeps the default (which is 1, as described in the datasheet).

Note: As you probably have read in the datasheet, fuses cannot be changed by the PIC itself (at least not for this PIC), so this configuration has no effect when you use a bootloader.

[7] You might wonder how 'OSC' relates to these bits defined. Is there magic in the compiler or Jallib? Of course not, just look in the device file and search for fuse_def. There you will see the names of the target pragmas and it's possible values with related bits. Just keep this in mind – it is useful information when you try to set a less-common configuration.

Operating with digital I/O pins

PIC chips – like PIC16F877A – have several digital I/O pins you can handle in your code and these pins are defined in the device files. Take a look at the remainder of the blink sample:

```
33 enable_digital_io()                 -- disable analog I/O (if any)
34  --
35  -- You may want to change the selected pin:
36  var bit led            is pin_A0    -- alias
37  pin_A0_direction = output
38  --
39  forever loop
40     led = on
41     _usec_delay(250_000)
42     led = off
43     _usec_delay(250_000)
44  end loop
45  --
```

On line 36, we create an alias, called 'led' for pin_A0. So if we write 'led', we actually use 'pin_A0'. And if we want to use a led on a different pin, we only have to make changes here, rather than throughout the whole program. And we need to change line 37, where we set the direction of the led pin A0 to output.

Next is a simple loop which lasts forever (line 39 to 44). In this loop the led is switched on, we wait 0.25 seconds (250_000 microseconds), the led is switched off, another 0.25 seconds delay and then back to line 39, where we started.

Now we have seen the use of a digital output, let's take a look at the next sample, 16f877a_in_and_out.jal. This is also a blink-sample, but the blink rate can be changed by pushing a button that is connected to a digital input. The schematics shown of figure 2 shows how the button S1 is connected to the PIC16f877A, in addition to the LED we used before.

In the code below, we have the pin definition like above. Next, at line 38, there is a statement that starts with ';@jallib'. Since it starts with ';'. It is handled as comment by the compiler and you can ignore this too: these statements are part of the sample generation process of Jallib.

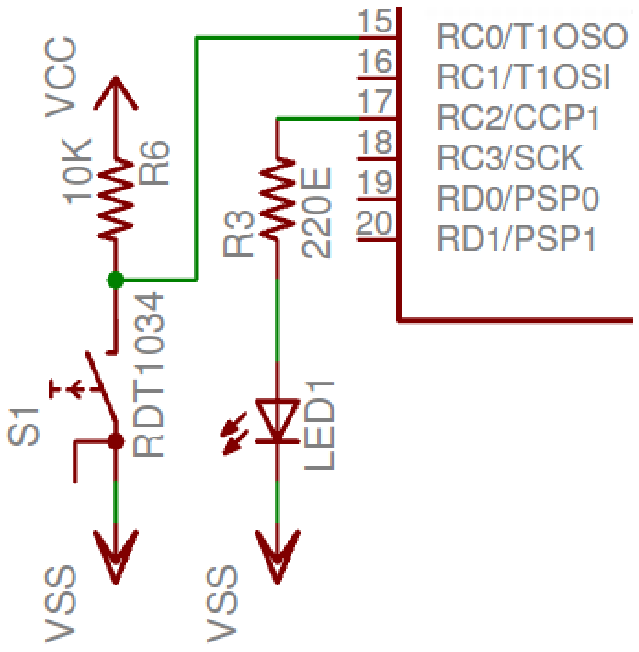


Figure 2

```
36  alias led            is pin_c2
37  alias led_direction  is pin_c2_direction
38  ;@jallib section button
39  -- button IO definition
40  alias button            is pin_c0
```

```
41  alias button_direction   is pin_c0_direction
42
43  enable_digital_io()
44
45  include delay
46
47  led_direction = output
48
49  button_direction = input
50
51  forever loop
52     led = ! led
53
54     if (button) then
55       delay_1ms(250)
56     else
57       delay_1ms(100)
58     end if
59
60  end loop
```

On line 40 and 41, we define the button pin aliases, just like we did with the LED, and on line 49 we set the pin as input. This is just to show you how this is done and in this particular case we could leave the statement out since I/O pins are input at startup.

As you can see, the loop has changed a bit. First of all, the 'led = on' and 'led = off' statements are replaced by only one statement at line 52, which contains two references to 'led', the I/O pin to which the LED is connected. The right hand reference 'led' reads the value of led (0 or 1), the exclamation point inverts the value (so 0 becomes 1 and 1 becomes 0). The result is the new value which is assigned to the I/O pin by the left hand reference to 'led'.

Next, we see an 'if/then/else' construct. The if-statement on line 54 evaluates the expression between parentheses, which in this case means it reads the value of 'button'. If the result is true, it continues execution after 'then' up to 'else' (line 55, in this case) and next after 'end if' (line 59). If 'button' is false, execute resumes after 'else' (line 57).

So why say 'evaluate the expression' when we just read the value of button? Consider the next cases:

```
1     if (button == true) then
2     if (button == false) then
3     if ( ! button ) then
4     if (alpha > 12) then
5     if (alpha > (bravo + 3)) then
```

The first case is functionally the same as the one on line 54. It takes the value of button, compares[8] it to 'true' and both values are the same, the expression as a whole is true.

[8] We use '==' to check if two values are equal, which is different from assigning a value to a variable. Like you have seen before, we use a single '=' for assignment.

In the second case, the value of button is compared with 'false'. If the value of button and 'false' are the same, the expression evaluates to true.

In the third case, evaluate means we take the value of button and invert its value with the exclamation mark. So this is functionally the same as the previous case.

Case 4 checks if the value of variable 'alpha' is larger than 12. If so, the expression evaluates to true, otherwise to false.

And the last case checks if the value of variable alpha is larger than (bravo + 3). I guess this is complex enough for now, but you can imagine more complex conditions can be evaluated to the two cases – true or false – that are used by the 'if' statement.

On your test board, you have a resistor from the button pin to +5V and a button to ground. So if you don't push the button, the resistor will pull the pin high, executing line 55 which gives a blink time of 250ms on and 250ms off. If you push your button, the pin gets low, executing line 57, so the LED blinks faster – 5 flashes per second at 100ms on and 100ms off.

This is how you can use a single digital I/O pin. And along the way, we introduced different bit constants. Below is an overview of all bit constants that are used by JAL and JALLIB:

```
const bit   TRUE              = 1
const bit   FALSE             = 0
const bit   HIGH              = TRUE
const bit   LOW               = FALSE
const bit   ON                = TRUE
const bit   OFF               = FALSE
const bit   ENABLED           = TRUE
const bit   DISABLED          = FALSE
const bit   INPUT             = TRUE
const bit   OUTPUT            = FALSE
```

We've seen pins A0, C1 and C2. If you look at the datasheet, you see the corresponding pins are named RA0, RC1 and RC2. And in the same datasheet, you see that pins are grouped in ports of (up to) 8 pins. You can access the port – that is: all pins that belong to the same port – through , registers where the right bit (lsb[9]) is pin 0 and the left bit (msb) is pin 7. Each I/O port has these two 8-bit registers:

```
PORTA                = 0b_0101_0101
PORTA_direction      = 0b_0011_0011   -- 0 = output
```

These are the 8-bit equivalent of the two bit variables of A0. In the code above, the even pins are set high and the odd pins low. The next line set the direction register, so pins 2, 3, 6 and 7 are set to output (since output = false = '0') and the other pins remain input. Now the pins set to output become high or low, according to the value assigned to PORTA.

[9] lsb = least significant bit, msb = most significant bit.

In addition to this, we can also access a nibble – a set of 4 bits. The registers[10] available for this are:

```
PORTA_low_direction
PORTA_high_direction
PORTA_low
PORTA_high
```

Variables with '_low' map to the lower 4 bits of a register (bits 0…3) and the variables with '_high' map to the higher 4 bits (bits 4…7). In both cases, the lower 4 bits of these registers are used, so PORTA_high_direction = 0b_0000_1111 set pin A4 trough A7 to 1 (which is input).

Before we go to the next example, let's take a look at the way we created the delay.

In 16f877a_blink.jal, we used _usec_delay(250_000) to specify the delay in microseconds. The procedure name starts with an underscore, which indicates it is either from the reserved namespace of the compiler, or it is internal for Jallib and not intended for general use[11].

In 16f877a_in_and_out.jal, we use delay_1ms(250) where 250 (or 100, as used on line 57) specify the delay in milliseconds, which is more appropriate then microseconds when we want to blink a LED. This procedure is defined in delay.jal, a Jallib library that is available since we included it in line 45, just like we included the device file. Actually, a device file is a library like any other. What makes a device file special is that it specifies everything that is device-specific, while other libraries are as device-independent as possible.

About the parameter of include statement in general: this is the filename of the file to be included, but without the '.jal' extension. Use a forward slash to specify a directory name, e.g. include the file delay.jal from the directory 'old_version' with:

```
include old_version/delay
```

[10] You won't find these registers in the datasheet of your PIC. To provide easy access to the upper or lower half of a port, there are pseudo variables defined in the device file. And its parameter is not really a nibble either. It is a byte of which the upper half is ignored. We will go into pseudo variables later.

[11] Jallib internals might have internal dependencies and the interface may change in future versions of Jallib – we are reluctant to change external interfaces (the ones shown in the samples and tutorials). The internal interfaces – the ones with the underscores – are changed whenever it suits our needs. So you better not use them in your application.

Analog to Digital converter

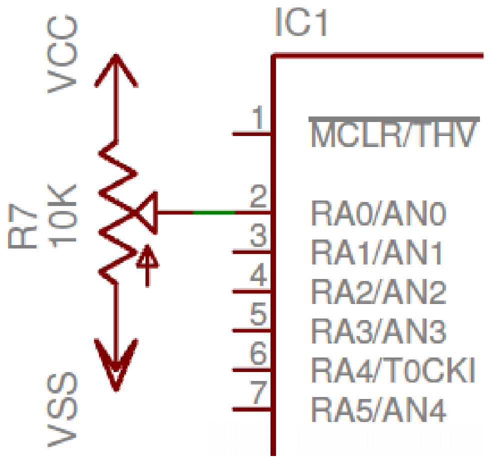


Figure 3

Many PICs have an Analog to digital converter (ADC). The number of ADC pins and bits varies. Also some have advanced features like reference pins. This is all well-documented within the datasheet of you PIC and beyond the scope of this document.

We take a look at sample 16f877a_adc.jal, which reads the analog value of pin AN0. The hardware used to test this is shown in figure 3.

From sample 16f877a_adc.jal:

```
01  -- ok, now let's configure ADC
02  -- we want to measure using high resolution
03  -- (that's our choice, we could use low resolution as well)
04  const bit ADC_HIGH_RESOLUTION = high
05  -- we said we want 1 analog channel...
06  const byte ADC_NCHANNEL = 1
07  -- and no voltage reference
08  const byte ADC_NVREF = 0
09  -- now we can include the library
10  include adc
11  -- and run the initialization step
12  adc_init()
```

As you can see, this sample is well-documented. Line 4 enables high-resolution ADC, which is 10 bits on PICs that support 10bit ADC. Low-resolution (8 bits) ADC is also supported while high resolution (10 bit) support is enabled.

Line 6 sets the number of ADC channels you want to use and line 8 disables references, so the ADC will convert relative to the power supply voltage[12].

Line 10 and 12 actually activate the ADC library. Make sure you don’t call enable_digital_io() after this adc_init(), since it disables use of ADC.

Next the main loop of sample 16f877a_adc.jal:

```
01  var word wmeasure
02  var byte bmeasure
03  forever loop
04     -- access results in high resolution
05     wmeasure = adc_read(0)
06
07     -- though we are in high resolution mode,
08     -- we can still get a result as a byte, as though
09     -- it were in low resolution.
10     bmeasure = adc_read_low_res(0)
```

[12] ADC configuration can be quite different for other PICs. Check out your sample or http://justanotherlanguage.org/content/jallib/tutorials/tutorial_adc_intro for more info.

```
11
12  end loop
```

On the first two lines, a word variable (16 bits) and a byte variable (8 bits) are declared. Line 5 shows how to get a 10-bit value from ADC pin 0 (AN0), line 10 gets an 8-bit value from this pin.

If you want more info, you can get it from:

- 16f877a_adc.jal sample (or an ADC sample using your PIC) showing a working example of the ADC that prints the result to the serial port.
- The JalApi documentation.
- Microchip datasheets.

LCD & print

We've seen how Jallib supports peripherals like digital I/O and analog input. But there is more. Jallib also supports 'external' devices like LCD displays. Let's take a look at the most common LCD display: a character display with an hd44780-compatible controller.

Figure 4 shows how an LCD character display with two lines of 16 characters each and a 16x1 pin connector is connected to the PIC16F877A. The schematics diagram is part of the evaluation board diagram shown in appendix A.

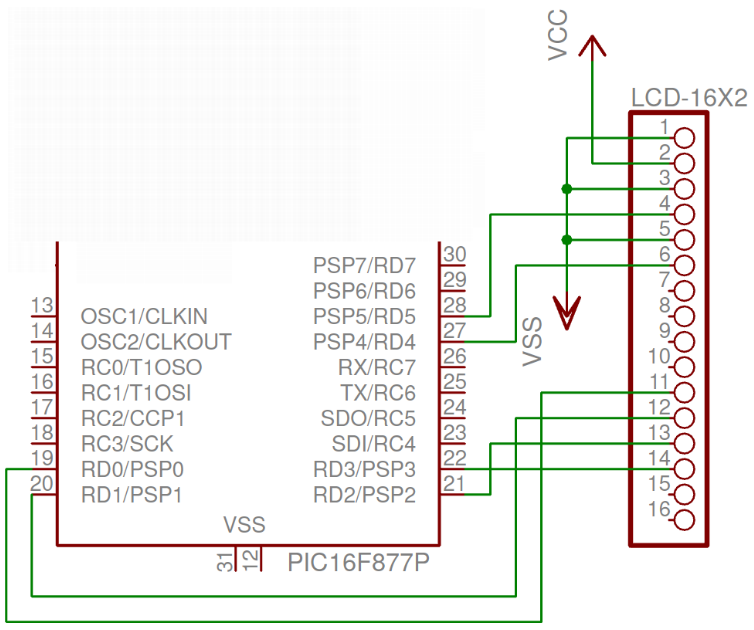


Figure 4

If you have a different configuration or need info to connect your display to the PIC, take a look at the tutorial at http://justanotherlanguage.org/content/tutorial_lcd. If the 4 data-lines of your display are not connected to the upper or lower half of a specific port, take a look at samples for *_lcd_hd44780_4_1.jal. For this tutorial sample 16f877a_lcd_hd44780_4bit.jal is used, which is similar to *_lcd_hd44780_4_4.jal samples.

As usual, writing a program with Jallib starts with configuring and declaring some parameters. So we first have to declare to which pins the LCD is connected (line 2 to 9). Next we declare the LCD geometry (line 11 and 12).

```
01  -- LCD IO definition
02  alias lcd_rs           is pin_d5      -- LCD command/data select.
03  alias lcd_rs_direction is pin_d5_direction
04  alias lcd_en           is pin_d4      -- LCD data trigger
05  alias lcd_en_direction is pin_d4_direction
06
07  -- dataport nibble:
08  alias lcd_dataport is portd_low       -- LCD data  port
09  alias lcd_dataport_direction is portd_low_direction
10
11  const byte LCD_ROWS     = 2           -- 2 lines
12  const byte LCD_CHARS    = 16          -- 16 chars per line

31  lcd_rs_direction        = output
32  lcd_en_direction        = output
33  lcd_dataport_direction  = output
34
35  include lcd_hd44780_4            -- LCD library with 4 data lines
36  lcd_init()                       -- initialize LCD
```

The led-related stuff from line 13 to 30 is left out, since they are not relevant for the LCD example, but you are encourage to take a look at the whole sample. We jump to line 31 to 33, where the LCD I/O pins are set to output. On line 35 the library is included and on the next line, a call to lcd_init() activates the library. At this point, the LCD display is ready.

The next code example shows various ways to get text on the LCD. The two basic ways are shown on line 56 and 57. Line 57 shows the classic way: call the output-procedure lcd_write_char() with a character as a parameter.

Line 56 is functional equivalent. The space-character is assigned to a pseudo-variable called 'lcd' and the library handles this just like the call to lcd_write_char(" ") . The advantage of the pseudo-variable 'lcd' over the procedure call to lcd_write_char() is that 'lcd' can be passed to a procedure as a parameter, as shown on line 38 to 43.

Line 38 includes print.jal, a library that supports printing variables. On line 40 a constant string is defined, as an array[13] of bytes. Note that JAL automatically determines the length of the string, so you don't have to specify its length within the brackets.

On line 42, the cursor is directed to the first position of the first line of the display (parameters are 0-based).

On line 43, print_string() is called with two parameters: the pseudo-variables of the device and the string. This way, pseudo-variables can be used as 'device' or 'file handle' and you don't need to create a print-string function for each device. When the call is executed, the string "Hello world!" is displayed on the lcd screen.

```
38  include print                       -- formatted output library
39
40  const byte str1[] = "Hello world!"  -- define strings
41
42  lcd_cursor_position(0,0)            -- to 1st line, 1st char
43  print_string(lcd, str1)             -- show hello world!
44
45  var byte counter = 0
46
47  forever loop                        -- loop forever
48
49     counter = counter + 1            -- update counter
50     lcd_cursor_position(1,0)         -- second line
51     print_byte_hex(lcd, counter)     -- output in hex format
52     delay_100ms(3)                   -- wait a little
53
54     if counter == 255 then           -- counter wrap
55        lcd_cursor_position(1,1)      -- 2nd line, 2nd char
56        lcd = " "                     -- clear 2nd char
57        lcd_write_char(" ")           -- 3rd char, equivalent to
58                                      -- the previous line
59     end if
60
```

[13] We will take a closer look at arrays later.

```
61  end loop
```

On line 51, the byte variable counter is printed on the display in hexadecimal format. You should be able to figure out the rest of the code by now. And when you have done so, take a look at the full API documentation of lcd_hd44780_common, which provides most of the functionality of lcd_hd44780_4[14].

Print.jal converts variable content to text. It allows you to:

- Print a byte, sbyte, word, sword, dword or sdword in decimal format.
- Print a byte, sbyte, word, sword, dword or sdword in hex format.
- Print a bit value in as 1/0, high/low or true/false
- Print a literal string
- End a line by sending a carriage return and linefeed.

Check out the JalApi page of print.jal to see more details.

Before we go to the next chapter, let’s have one more look at the example, lines 40, 56 and 57. They all use double quotes, but there is a subtle difference. The common use of quotes is to get the value of a single character, like on line 56 and 57. If you specify more than one character in this case, the first character is used and the rest is (silently) ignored. For example:

```
lcd = "abc”
```

gives a warning (only 'a' of "abc" is used in this expression) and is functional equivalent to

```
lcd = "a”
```

There is only one exception to this rule: when you initialize an array, you can use a quoted string of characters, as shown on line 40.

[14] How could you know? Since procedures like lcd_setcursor() are not shown in the JalApi page of lcd_hd44780_4, it must come from one of the files shown in the ‘dependencies’ section. You may also notice that there is a private section with procedures starting with an underscore (‘_’). These procedures are intended to be called from within the library only. So only use them in your application when you fully understand how to and be aware they may change (in name or behavior) in future releases of Jallib.

Serial port

The serial port of a PIC controller enables you to exchange information with other devices. Personally, I hook up every PIC to my PC via the serial port to debug my programs in real time. For this, I use the usart peripheral, which uses pin RC6 and RC7. Figure 4 shows these pins connected to a connector, together with the power supply. You do need an addition interface to connect this Vcc-level (5V) serial port to a RS232 port. See http://www.justanotherlanguage.org/content/jallib/tutorials/tutorial_serial_communication for a tutorial on this subject.

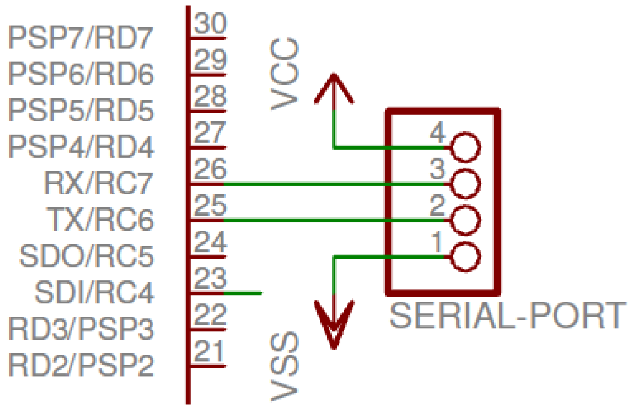


Figure 5

The tutorial mentioned uses hardware (which means it uses the peripheral 'usart', present in most PIC's) for serial communications. Serial hardware features:

- Only little code (and processing time) is required to send or receive a byte.
- The processor can execute other tasks while maintaining communication.
- Timing is handled by the hardware, which gives high accuracy.
- High speeds (like 115200 baud with a 20 MHz crystal) are supported.
- Interrupts on transmit and receive facilitate queuing of bytes to transmit and bytes received

Let's take a look at the sample 16f877a_serial_hardware.jal:

```
include 16f877a

pragma target clock 20_000_000                  -- xtal frequency
pragma target OSC        hs
pragma target   LVP disabled
pragma target   WDT disabled

include delay

-- set all IO as digital
enable_digital_io()

-- ok, now setup serial;@jallib section serial
const serial_hw_baudrate = 115_200
include serial_hardware
serial_hw_init()

include print                        -- output library

const byte str1[] = "Hello serial world"    -- define a string
print_string(serial_hw_data, str1)   -- output string to serial

-- inform user PIC is ready !
serial_hw_write("!")

```

```
26  -- let's build our loop
27  var byte char -- will store received char
28  var word counter = 10
29  forever loop
30    if (serial_hw_read(char)) then
31          serial_hw_write(char)   -- that's the echo...
32    end if
33
34    counter = counter - 1;
35    if (counter == 0) then
36       counter = 50000
37        serial_hw_data = "."
38     end if
39  end loop
```

The first nine lines will look familiar by now. At line 13 to 16, we setup the serial communications. We define the desired speed at 115200 baud.

Next we include the serial_hardware library and call serial_hw_init() to initialize the serial port. You don't need to define any pins, since the serial hardware is fixed to a specific transmit and receive pin. Take a look at the datasheet to see which pin your PIC uses and make sure leave the pins as input for the library to work properly.

In the previous chapter, we showed how print.jal can be used with the 'pseudo-variable' interface of an lcd. On lines 18-21 we use print.jal with the pseudo-variable serial_hw_data to print "Hello serial world" to the serial port.

But just like the lcd library, the serial library also has a procedure interface. This is shown on line 24, where serial_hw_write() is called to output the character '!' to the serial port.

The last line of the sample program we look at is 30:

```
30    if serial_hw_read(char) then
31          serial_hw_write(char)            -- that's the echo...
32    end if
```

Here we call the function[15] serial_hw_read() with a variable named 'char' as parameter. The function checks if there is a byte received by the serial hardware. If this is not the case, the value of 'char' is undefined and false is returned. And if 'false' is returned, the if-statement will not be valid and execution will continue after the 'end if' statement of line 31.

If there is a byte received by the serial hardware, the byte received is assigned to 'char' and true is returned, making the if-statement valid. As a result of this, execution will continue after 'then', so the character received is sent back via the serial port.

To conclude this chapter, we take a brief look at two more serial libraries that might suit your needs.

The first one is serial_hw_int_cts.jal. This library queues transmit and receive bytes (the JalApi doc shows an example of a 32 byte transmit queue and 64 byte receive queue). Use this if you can't

[15] We will take a look at procedures and functions later.

service your serial port fast enough to prevent receive overrun[16] or when you don't want to your program to wait for each individual character to be transmitted.

Another serial library is serial_software.jal. It takes quite a lot of attention to execute serial communications in software. The library disables interrupts during transmit and receive, you can't transmit and receive at the same time and receive is always blocking (the procedure waits up to a specified time or until a character is received, preventing execution of any code while waiting for a character).

But serial_software has one major advantage: it can be used on any digital I/O pin. This feature is sometimes used to provide serial communications on the PGD and PDC pins. With a wisp648 programmer, this enables serial communications without any additional hardware. But if you hook up pins 7 and 8 of the wisp648 to the usart pins (26 and 25 on an 16f877a and many other 40-pin PICs), you can use the hardware libraries[17].

[16] You need to read a received character before the next one is received or the character will be lost. At 115200 baud, a byte (which in this case takes 10 bits on the serial line) could arrive only 87 microseconds after the previous one…

[17] Wisp programmers use software that operates on digital I/O for pass-through. The good thing about is that it works independent of the setting of your serial port. Downside is its limited speed: it works reliable up to 19200 baud, while serial_hardware works well at 115200 baud and beyond.

PWM

Let’s use the PWM library to fade the led. Pulse Width Modulation or PWM is a technique of switching a pin on and off, mostly at a fixed-frequency signals. The percentage of the time the signal is high is not fixed to 50%, like common signals, but varies. This is a way to provide a digital to analog conversion which, in our case, fades the LED on an off. If you want to use PWM, check out the tutorial at http://justanotherlanguage.org/content/jallib/tutorials/tutorial_pwm_intro

PWM is one of many features provided by the ccp-peripheral in cooperation with a timer. Check out your PICs datasheet to see all features of the ccp-module. In combination with the library code, you gain understanding on how the PWM is implemented. Is this necessary? Not to understand this chapter and common use of the library. But it is the way to truly understand JAL, Jallib and microcontrollers…

Below the sample 16f877a_pwm_led.jal is shown, which is written by the Jallib father, Sebastien Lelong. You know the header of the Jallib files and the chip setup by now, so we left this out. And since it uses the LED that is already connected to RC2, we don’t have to go into hardware this time either.

```
30  -- Configure PWM
31  pin_ccp1_direction = output
32  include pwm_hardware
33  pwm_max_resolution(1)
34  pwm1_on()
35
36  forever loop
37     var byte i
38     i = 0
39     -- loop up and down, to produce different duty cycle
40     while i < 250 loop
41        pwm1_set_dutycycle(i)
42        _usec_delay(10000)
43        i = i + 1
44     end loop
45     while i > 0 loop
46        pwm1_set_dutycycle(i)
47        _usec_delay(10000)
48        i = i - 1
49     end loop
50     -- turning off, the LED lights at max.
51     _usec_delay(500000)
52     pwm1_off()
53     _usec_delay(500000)
54     pwm1_on()
55
56  end loop
```

On line 31, the pin ‘ccp1_direction’ is set to output and you might wonder what pin this is. Well, it is the pin that is connected to ccp1 on your PIC. This pin also has a ‘normal’ name and you have to look in the datasheet or device file to see what that is. On the 16f877A, pin_ccp1 is pin_c2. And similar to this, all pins have multiple names – one for each function they support. This is very useful, but can be tricky since it is not obvious that a command like pin_c2_direction = input or portc_direction = all_input eliminates the effect of line 31. So be careful if you must mix different pin names.

The rest of the sample is left as an exercise for the reader. We get to ‘while loop’ later in this document and the API reference of pwm_common.jal[18] (yes,’ pwm_common’, not ‘pwm_hardware’, like with LCD) provides you the info you need.

[18] http://jallib.googlecode.com/svn/trunk/doc/html/pwm_common.html

Some language features.

By now, you have seen quite a lot of JAL code and this is a good time to take a closer look at the JAL language. For this we use the sample 16f877a_startersguide.jal. This sample is atypical in the sense that it is just a collection of small blocks of code that have no relationship and do not explain much on its own. But it provides you the code used in this chapter, so you can try, modify and extend the examples shown. For your convenience, the numbers shown in the code fragment correspond with the line numbers of the sample.

You have probably noticed by now that JAL is case-insensitive. This means that

```
135      if (Alpha > 0) then
136         Alpha = Alpha + 1
137      end if
```

is equivalent to

```
139      If (Alpha > 0) THEN
140         ALPHA = AlPhA + 1
141      END IF
```

But these examples make it also clear that consistent casing enhances the readability of the code. At Jallib, we choose to use lower case only with underscores to separate words. This convention makes it easy to remember how a procedure name is exactly written, since you know SerialData is not compliant and serial_data is.

Other features that are solely used to enhance readability are lines and indentation. The code above could be written like:

```
if(alpha>0)then alpha=alpha+1 end if
```

Or

```
if(alpha>0)then
alpha=alpha+1
end if
```

The same for the compiler, but not for the human reader!

If you look at http://www.casadeyork.com/jalv2/jalv2/index.html (which contains the official language reference), you see that assembly is also supported. The reference also states that assembly is only to be used as a last resort. So don’t use assembly, unless you really, really need to!

Literal constants

Literals constants are basically the numbers you specify in your program. By default literals are decimal, but in some cases it is more appropriate to use another base:

```
125      alpha   = 0b_0100_0011 -- binary
126      bravo   = 0q203        -- octal
127      charlie = 0x43         -- hex
128      delta   = "c"          -- ascii
```

Note the underscore shown in the binary number, used to make the number easier to read. Underscores are allowed in each constant number and ignored by the compiler. The obvious exception to this is of course an underscore in a quoted string, where is not ignored.

Variables

Jal supports 3 types of variables: bit, byte and sbyte. A bit is what you expect: a single bit. A byte is an 8-bit unsigned variable ranging from 0 to 255 and an sbyte is an 8-bit signed variable that ranges from -128 to 127.

JAL also supports variables that are multiple bytes. Predefined types are word / sword (2 bytes) and dword / sdword (4 bytes). But you can use any arbitrary number of bytes. For example:

```
var byte*4 bravo
```

is (except for the variable name) equivalent to

```
var dword  charlie
```

and

```
var byte*3 delta
```

specifies a 3-byte (24 bit) variable.

When you assign the content of a longer variable to a shorter one (like alpha = charlie), some data obviously gets lost. The compiler warns you in this case with the message:

```
16f877a_startersguide.jal:150: warning: assignment to smaller type;
truncation possible
```

You are encouraged to inspect your code and make sure your application does not require the higher (truncated) bits. And if you did, omit the warning by casting it to the smaller type as shown in the sample below. This way you avoid that relevant warnings get buried between irrelevant ones.

```
var byte   alpha
var byte*4 bravo

alpha = byte(charlie)
```

Cast to all types mentioned are possible, like sbyte, word and byte*3.

In some examples, we used array of variables like:

```
var byte   foxtrot[10]
```

This creates a variable array of 10 bytes so the index, a number or variable between the square brackets, can be a value from 0 to 9. Arrays can be of any data type, except bit. Bit arrays are not supported by JAL, but take a look at library bit_array_1.jal if you need one.
You can initialize arrays at definition like:

```
110  var byte   hotel[]   = { 1, 2, 4 }
111  var byte   india[]    = "foxtrot"
```

When the size of the array is not specified, this is determined by the compiler. And be aware: these arrays are variables, using your scarce RAM memory. Text strings often don't change and in that case, you better declared them as constant so they are stored in the larger flash memory:

```
114  const byte str1[] = "\r\n\nHello Jallib world! \\ \r\n"
```

Note the escape sequences '\r' and '\n' and '\\'in the string. They represent carriage return, newline and a single '\'. The escape sequences supported by JAL are:

Sequence	Value
"\0qqq"	octal constant
"\a"	bell
"\b"	backspace
"\f"	formfeed
"\n"	line feed
"\r"	carriage return
"\t"	horizontal tab
"\v"	vertical tab
"\xdd"	hexidecimal constant
"\\"	Single backslash

Next, let's take a look at more advanced use of variables:

```
107   var byte*3 delta
108   var byte   echo[3] at delta

149   charlie = 0x12345678
150   alpha = charlie
151
152   alpha = byte(charlie)
153
154   print_byte_hex(sg_output, alpha)
155   sg_output= " "
156   print_byte_hex(sg_output, echo[0])
157   sg_output= " "
158   print_byte_hex(sg_output, echo[3])
```

A byte array can be used to access individual bytes of larger variable by specifying the location of the array with 'at'. Above, line 108 defines the array echo at the location of delta, so echo[0] is at the location the lower byte of alpha and echo[3] at the highest byte. On line 152, the lower byte of charlie is assigned to alpha. Line 154-159 output "78 78 12", the hex values of alpha, echo[0] and echo[3].

In a similar way, bit variables are mapped to registers. In the device file 16f877a.jal, see the definition of pin_B2[19]:

[19] Use of the mapping a bit variable to a byte variable is demonstrated with the variable 'julia' in the sample.

```
var volatile bit pin_B2 at PORTB : 2
```

The ': 2' at the end indicates that the bit-variable pin_B2 maps to bit 2 of PORTB.
In the variable definition above the keyword volatile tells the compiler to handle the variable pin_B2 in a special way:

- The variable won't be removed by the optimizer[20].
- When the variable is used as input for a formula, it is read exactly once (for each occurrence in the formula). This means it is not read multiple times, nor is a cached value of the variable used.
- When the variable is assigned, it is written exactly once.

Volatile is used often in the device files. You will only need it in specific cases like:

- For variables that are used both in an interrupt service routine and in your main program.
- For procedure parameters that pass a volatile parameter. For example, a procedure that takes a pin as a parameter and waits for this pin to get low will only work if the pin parameter is defined volatile.

Now that we have covered most of the variables, let's finish with a very powerful JAL feature: pseudo variables. Using a pseudo variable is equal to using a normal variable: you can assign a value to it, read its value and pass the pseudo variable to procedure. The difference is in the way the variable is implemented: instead of some reserved memory space, a pseudo variable is implemented by a piece of code that is called when the variable is assigned or read. Let's take a look at the example of the pseudo-variable pv below.

```
073  var byte pv_store
074
075  procedure pv'put(byte in invar) is
076     pv_store = invar + 1
077  end procedure
078
079  function pv'get() return byte is
080     return pv_store * 2
081  end function
```

The pseudo-variable behavior is triggered by the `'put` and `'get` postfix of the procedure and function[21]. When you assign a value to pv like at line 171, the procedure pv'put() is called with invar = 7, so 7+1 =8 is assigned to the variable pv_store.

```
171     pv = 7
172     print_byte_dec(sg_output, pv)

175     print_byte_dec(sg_output, pv_store)
```

[20] To save space, the optimizer part of the compiler removes variables that are not used. And 'not used' does not only mean variables that are not used at all, but also variables that get value's assigned, but never get read. In this case, not only the variable is removed, but also the code that assigns a value. Unless of course the variable is defined 'volatile'.
[21] we have a closer look at procedures and functions shortly

The use of ‘pv’ when calling print_byte_dec() at line 172 triggers a call to pv’get(), which returns 8*2=16. That value is printed to the serial port.
Pseudo variables are used a lot in libraries. Actually, serial_hw_data is a pseudo variable that sends a character to or receives a character from the serial port. In the LCD example, we saw ‘lcd’, which is a pseudo variable to write a character to the display. There is no support for reading information from the display, so lcd’get() is not implemented.

Procedures & functions

We’ve seen that a JAL program can be a sequence of statements. The blink example we saw at the start of this document had a few configuration statement and then statements that generate code: enable_digital_io(), set the pin to output and the loop that blinks the LED. If your code gets larger, you can use procedures and functions to structure your code. First you define your procedure[22] and after the procedure is defined[23], you can call it. Procedures which are defined but not called are removed by the optimizer so they don’t enlarge your program.
Let’s take a look at some examples:

```
procedure sgd_initialize() is
   ;
end procedure

function sgd_receive_function() return byte is
   var byte data = 0
 ;
   return data
end function

procedure sgd_receive_procedure(bit in ACK, byte in out data) is
   data = 0
 ;
end procedure
```

A procedure takes zero or more parameters, in the example above it takes one. Each of the parameters has a type and is defined ‘IN’, ‘OUT’ or ‘IN OUT’. ‘IN’ tells the compiler that the corresponding parameter has to be passed to the procedure when it is called and ‘OUT’ indicates that the value of that variable has to be passed to the calling parameter at the end of the procedure. So when you have an input parameter, e.g. for a calculation, use ‘IN’ since you mostly don’t want your calling parameter to be changed. In cases where you want to give information back from the procedure, use ‘IN OUT’. The use of ‘OUT’ only is not recommended – it rarely serves a purpose expect for a hair smaller code and gives counter-intuitive behavior in some cases[24].

[22] Procedures and functions have much in common and we will go into the details shortly. When we refer to procedure and if it is not obvious we mean ‘procedure as opposed to function’, we probably mean ‘procedure or function’.
[23] To be more precise, the compiler must know how to call a procedure. The easiest way is to define the full procedure before you use it, but in exceptional cases you might need a procedure prototype. A prototype in JAL is the start of a procedure definition, up to ‘is’, e.g. ‘procedure sgd_receive_procedure(bit in ACK, byte in out data)’.
[24] See find_the_bug() procedure in the sample program if you want to know more about the pitfall mentioned.

A function is a procedure that has, in addition to the named parameters like a procedure, one unnamed return value which type is defined at the first line (after the calling parameters and before 'is'). At every return-point within the function, a value of this type has to be passed to the calling procedure. The return-value can be used in a calculation of assigned to a variable as shown below on line 123.
As stated before, procedures and functions can have zero parameters. In this case, the parentheses could be left out. This is however considered bad programming practice and I advise you to use the parentheses both when you define and call a procedure or function without parameters, to distinguish a procedure from a (pseudo) variable. So:

```
122     sgd_initialize()
123     alpha = sgd_receive_function()
```

Flow control

In this paragraph, I'll show you flow control examples. First, let's take a look at the most basic conditional statement, 'if-then':

```
191     if (x > 0) then
192        x = x - 1
193     end if
```

The code block (x = x – 1) is executed if the condition (x > 0) is true. If the condition is false, execution is resumed after 'end if'. It's also possible to supply a code block for the condition is false, using 'else':

```
195     if (x > 0) then
196        x = x - 1
197     else
198        x = x + 1
199     end if
```

But what if you want to supply code for (x == 7), condition (x >7) and all other values? Well, that's where elsif[25] comes in:

```
202     if (x == 7) then
203        x = x + 1
204     elsif (x > 7) then
205        x = x + 2
206     else
207        x = x + 3
208     end if
```

Line 203 is executed if x equals 7 and else, and if x is larger than 7, line 205 is executed. And if both conditions are false, line 207 is executed. And after one of these statements is executed, all resume execution after line 208.

[25] This is not a typo – the statement is elsif, not elseif.

If you need a different action for fixed values of a variable (or expression), you can use the case statement. The case statement on line 216 evaluates the expression between parentheses (in this case, reads value x). Then it searches for the matching case-value (the constants followed by':'). If it matches, the code block[26] gets executed. After the code block, execution is resumed after the 'end case' statement[27].

```
216     case (x) of
217        1 : block
218           x = 1
219        end block
220        2 : block
221           x = 3
222        end block
223        3 : block
224           x = 2
225        end block
226        otherwise block
227           x = x / 2
228        end block
229     end case
```

If there is no matching case-value, the 'otherwise' block (line 226-228) is executed. The otherwise-clause is optional, so you can leave it out when you don't need it.

Now let's take a look at the loops of JAL. Each program contains an indefinite main loop like:

```
118  forever loop
…
301  end loop
```

You don't want to exit the main loop, but might want to exit another forever-loop. You can do this with the 'exit loop' statement or with return if you are in a procedure.

```
237     forever loop
238        sg_output = "!"
239        x = x + 1
240        if (x > 10) then
241           exit loop
242        end if
243     end loop
```

If you want to run a loop a pre-determined number of times, say 7, use:

```
249     for 7 loop
…
```

[26] Block/end block groups multiple statements and also limits the scope of variables defined within the block. You can use block at any place in your program. It is good practice to use block/end block for each case value, since a case value can't be followed with multiple statements. Since all case values above have only one statement, the block/end block could be left out for all case-values in this particular case.
[27] So unlike C, no explicit break is required.

```
end loop
```

And if you want access to the loop counter:

```
var byte lc
for 7 using lc loop
   print_byte_dec(sg_output, lc)
   sg_output = " "
end loop
```

This loop outputs number 0 to 6. Of course, 'exit loop' can be used here too, as in all other loops.

If you need a loop that executes at least once and continues to repeat until a specific condition is met, use the repeat loop:

```
var byte x = 0
repeat
   sg_output = "#"
   x = x + 1
until (x == 0)
```

Can you predict what the last two values are – if there are two or more – that are printed by this loop?
In other cases, it is required to test the condition in advance and if the condition is false, the loop is not execute at all. For this, we have the while-loop.

```
x = 0
while (x > 0) loop
   sg_output = "$"
   x = x - 1
end loop
```

In the example above, variable X is set to zero. This makes the while-condition (x > 0) false, so the loop is not executed. In the next example, the loop is executed 4 times and also has a conditional 'exit loop' statement.

```
x = 4
while (x > 0) loop
   sg_output = "%"
   x = x - 1
   if (x == 14) then
      exit loop
    end if
end loop
```

In this chapter, we introduced many language features of JAL. This much information would take half a dozen chapters in a book, so it is quite likely that you do not know all this info by heart at this point. A good way to learn more is to try it yourself by writing code or modify the sample.

I²C

At this point, you have blinked a led with JAL and the device file, used libraries for peripherals like serial, ADC and PWM. You also had a look at a library for an 'external' component, the LCD screen and we covered most of the JAL language itself.

But there is much more in Jallib, like support for MMC and USB. Too much to cover in this guide, but do take a look at the tutorials.

In this last chapter, we take a look at i2c. I2c (or actually I²C, pronounced /aɪ-skwɛərd-si/ or /aɪ-tu-si/) stands for Inter-Integrated Circuit and is a two-wire bus, developed to attach low-speed[28] peripherals to a motherboard, embedded system, or cell phone.

In embedded applications, i2c is frequently used to communicate with slaves like eeproms (non-volatile memory), sensors for temperature, ultrasonic rangers and displays. And you can create your own i2c slave, using Jallib libraries.

In this guide, we focus on the libraries for hardware and software i2c master[29] and show the layered (modular) approach of Jallib. An overview of the i2c master libraries is shown in figure 6.

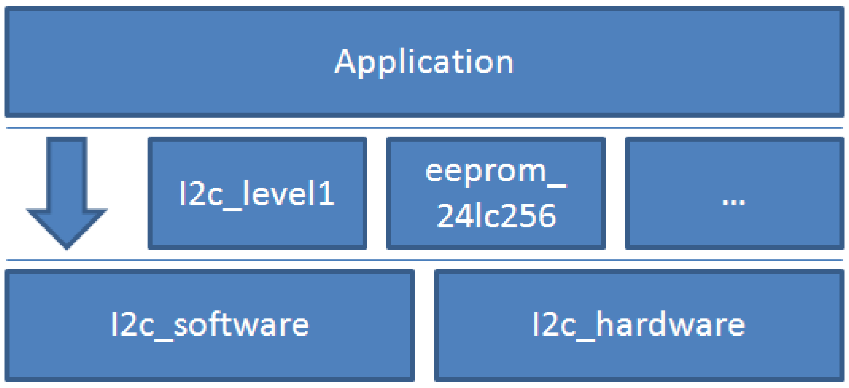


Figure 6

On the basic level (level0), there are two libraries, i2c_hardware and i2c_software. The first one implements i2c using the 'synchronous serial port' (SSP) peripheral, while the second one is a 'bit-bang' library – a library that handles the full protocol in software and operates on generic I/O pins.

Both libraries provide the same (level0) API, so they have procedures with the same name and functionality, providing all primitives to use the i2c bus.

From your application (top layer in the figure above), you can access the level0 API (the arrow on the left). This requires the application to handle start/stop sequences, i2c bus addressing and acknowledgements, while in most cases, you just want to send or receive a message from a slave. In this case, it is probably more convenient to include i2c_level1 which provides a generic message interface. For some i2c slave devices like a 24lc256 eeprom, there is a specific level1 interface that is even easier to use.

[28] I2c slave devices support at least 100 kHz clock, but many support 400 kHz or 1 MHz. Transfer of a single byte a 100 kHz takes 90 microseconds, excluding protocol overhead for addressing.
[29] i2c is a multi-master protocol but currently, only single-master protocol is supported by Jallib libraries.

Hardware setup

Figure 7 shows the I2C part of the schematics diagram from appendix A . The two wires of the i2c bus, labeled SCL and SDA, are clearly visible in the middle. The lines are pulled high with two resistors (R4 and R5) and the i2c slave device is a 24lc256 i2c eeprom.

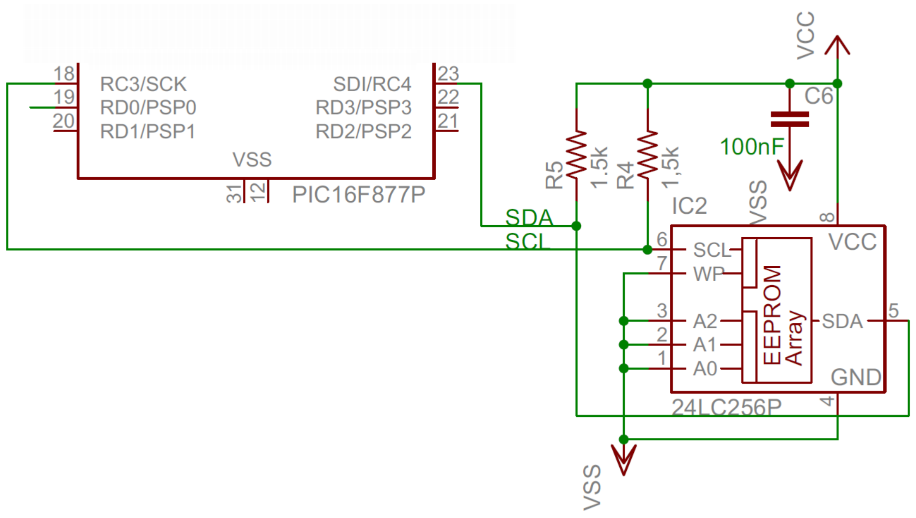


Figure 7

For more details on this, check the i2c slave datasheet or look on the Internet.

Software setup

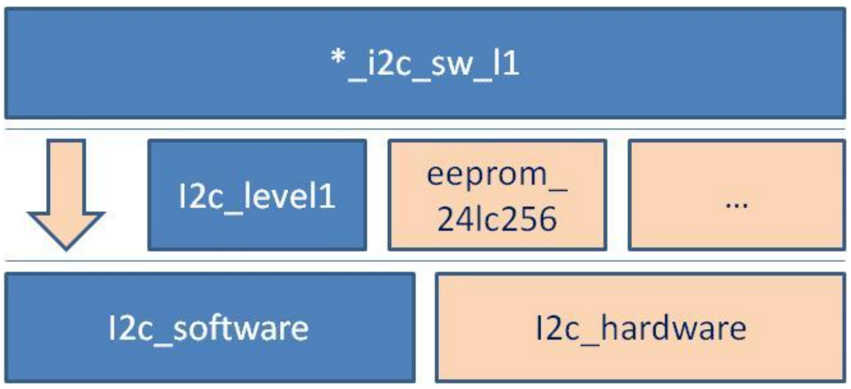
The first decision you have to make is if you want to use hardware or software i2c.

Hardware i2c is faster (especially at 1 MHz clock rate) and has a smaller footprint , but is also bound to the pins of the synchronous serial port (SSP) peripheral. Software i2c is a bit slower and larger, but can operate on any generic I/O pins which make it very flexible.

We will use software i2c, but our hardware configuration also supports hardware i2c so feel free to give this a try. Sample *_i2c_sw_l1 uses the generic i2c_level1 library over software i2c at level 0 to

Figure 1

demonstrate communication with a 24lc256 i2c eeprom. So we use the blue stack, shown in figure 8.



Let’s take a look at the configure part of *i2c_sw_l1.jal sample:

```
042  -- I2C io definition
043  alias i2c_scl            is pin_c3
044  alias i2c_scl_direction  is pin_c3_direction
045  alias i2c_sda            is pin_c4
046  alias i2c_sda_direction  is pin_c4_direction
054
055  -- i2c setup
```

Figure 7

```
056  const word _i2c_bus_speed = 1 ; * 100kHz
060  include i2c_software
061
062  i2c_initialize()
```

On line 042 to 046, define the both the pin and the direction bit of the pins you selected[30] and (on line 056) set the desired bus speed at 1, 4 or 10* 100 kHz. On line060 and 062, the 'level0' i2c library is included and initialized[31].

The library provides an interface to send and receive bytes. You can build complex messages with this interface, but you have to do this yourself and know about the i2c protocol. In most cases, it is easier to use the level1 layer. To use this, define an array to store the transmit messages and one for the receive messages. Then include the library:

```
063  var byte i2c_tx_buffer[10]
064  var byte i2c_rx_buffer[10]
065
066  include i2c_level1
```

The size of the two buffers is the maximum size of a message you will send or receive. When a buffer is too short, you might get compile errors or - worse - unexpected behavior. When the buffers are too long, memory will be exhausted faster. When in doubt: enlarge the buffer a few bytes!

Read from an i2c eeprom

Now we are ready to communicate with the i2c slave. The format of the messages depends on the device, so you should consult the slave's documentation, the 24lc256 in this case.

First thing to know of a slave is its address. An i2c address is 7 bits and is stored in the higher 7 bits of a byte. The lowest bit is set to zero. The 24lc256 address is 0xA0 (160 decimal)[32].

[30] With i2c_hardware the pins are fixed so you don’t have to specify them. Despite this, the pin definition is shown in the some i2c_hw samples as a result of the sample generation process.
[31] Lines 057 to 59 of the sample (which are not shown in this document) contain the statements of i2c_hardware. Lines 59 and 60 are the only real differences between these samples…
[32]Some other libraries use the lower 7 bits of an address, which would be 0x50 in this case. Others mention 0xA0 as the read address and 0xA1 as the write address, incorporating the r/w bit into the address.

Now we want to read data from the 24lc256. Let's assume we want to start reading at internal[33] location 1234 and need 3 bytes. From the datasheet, we learn that we have to write a 2-byte (= one word) location. Subsequently, we can read data. Or in JAL:

```
-- Send 2 bytes address to device 0xA0 and
-- then read 3 bytes of data
r = i2c_receive_wordaddr(0xA0, 0x1234, 3)

print_bit_truefalse(serial_hw_data, r)
serial_hw_data = " "

print_byte_hex(serial_hw_data, i2c_rx_buffer[0]);
serial_hw_data = " "
print_byte_hex(serial_hw_data, i2c_rx_buffer[1]);
serial_hw_data = " "
print_byte_hex(serial_hw_data, i2c_rx_buffer[2]);
serial_hw_data = " "
```

So only one call to i2c_receive_wordaddr() to do the i2c write and read! This function takes 3 params: the slave device address, the value of the 2-byte location code and the number of byte to read after the location code is sent. The result is stored in i2c_rx_buffer[] and is printed on the serial port by the example.

You might have noticed the return value r in the sample above. This bit value is true if the operation was successful. If false, communication has failed. Add code to check the return value and handle errors in your program!

i2c_receive_byteaddr() is a similar function which uses a 1-byte location code.

Arbitrary location code lengths can be sent by using i2c_send_receive(). This function is used in the example below.

Write to an i2c eeprom

Reading from the 24lc256 is not very useful without writing to it. So next some code that writes value 99 to location 2 of the eeprom:

```
-- write part (increment 3rd byte at 0x0002)
i2c_tx_buffer[0] = 0x12     -- high byte addr within i2c eeprom
i2c_tx_buffer[1] = 0x34 + 2 -- low byte addr within i2c eeprom
i2c_tx_buffer[2] = i2c_rx_buffer[2] + 1  -- data
r = i2c_send_receive(0xA0, 3, 0)
print_bit_truefalse(serial_hw_data, r)
```

Function i2c_send_receive() takes the slave address as the first parameter. The second one defines the number of bytes to be sent to the slave from i2c_tx_buffer and the third parameter defines the number of bytes to be received from the slave (and stored in i2c_rx_buffer). In the example above

[33] There are two types of addresses in this example. The first one is the i2c address of the slave device, 0xA0. This is used by the i2c master to indicate it want to talk to the eeprom. This eeprom stores 32k bytes of data and has an internal address to indicate the location of an individual byte.

(which is, just like previous code samples, slightly modified code from *_i2c_sw_l1.jal), we don't want to receive any information so the third parameter is 0 and i2c_rx_buffer is not used.

We've setup an i2c master, using the software library. We read data from a 24lc256 i2c eeprom and also wrote new data to is. You could repeat the previous samples with *_i2c_hw_l1.jal with the hardware configuration of appendix A.

I2c slave

Jallib also provides you with i2c slave libraries. These libraries require a PIC with a synchronous serial port (SSP) peripheral.

If you want to implement an i2c slave, you have two options:

- Keep tight control and maximum flexibility by handling each byte received or to be sent by your own code. If this is your choice, look at the sample *_i2c_hw_slave_echo.jal, which communicates with *_i2c_sw_master_echo.jal.

- Let the library collect incoming data and when a complete message is received, it is passed to your code. At this point you can synthesize a reply message, hand it over to the library and you are done. If this is your choice, look at sample *_i2c_hw_slave_eeprom_simulator.jal, which simulates the behavior of a 24lc256 alike eeprom of 32 bytes.

The use of first set of samples is described in detail in the Jallib i2c tutorial at http://www.justanotherlanguage.org/content/jallib/tutorials/tutorial_i2c1. This tutorial also provides useful information when you want to connect a second PIC with the eeprom-simulator to your current hardware. This simulator works with the code from this chapter. Just don't forget to disconnect the real 24lc256 when you connect the simulator OR change the i2c address of the simulator (e.g. to 0xB0), both in the simulator itself and in the master software.

Next steps

At this point, we covered most of the JAL language, but not all. First of all, we left out details that are not too relevant when you start with JAL and Jallib. We used some of the operators in examples and there is a list of all operators in appendix B. And we did not mention interrupts. Why? Well… interrupts are not a topic to start with if you are new to microcontrollers. Create a few working programs first, practice debugging and then give interrupts a try if you need them. Check out the language reference and have a look at Jallib libraries that use interrupts to get the idea.

We did not cover much of Jallib though. We just showed you some basic stuff to get you started. But Jallib provides libraries for peripherals, USB memory card access, etc. Too much to cover and too fast growing to keep up with for us authors. But with the basics we showed you, you should be able to give it a try on your own. And if you get stuck, you know where to get more information or help.

So, good luck and many joyful hours of JAL programming!

Appendix A – Evaluation hardware

To test the examples in this guide, you can buy an evaluation board or create your own. In most case, the examples can be adapted to the configuration.

All samples are tested on a dwarf board DB001, on which I tested all samples. Set it up with an analog signal on pin A0, a 16x2 LCD connected through a DB017 LCD interface to port D and a custom board on port C, with a LED, button and 24lc256. For more information on dwarfboards, see www.voti.nl.

For your reference, a functional equivalent schematics diagram of this configuration is shown below.

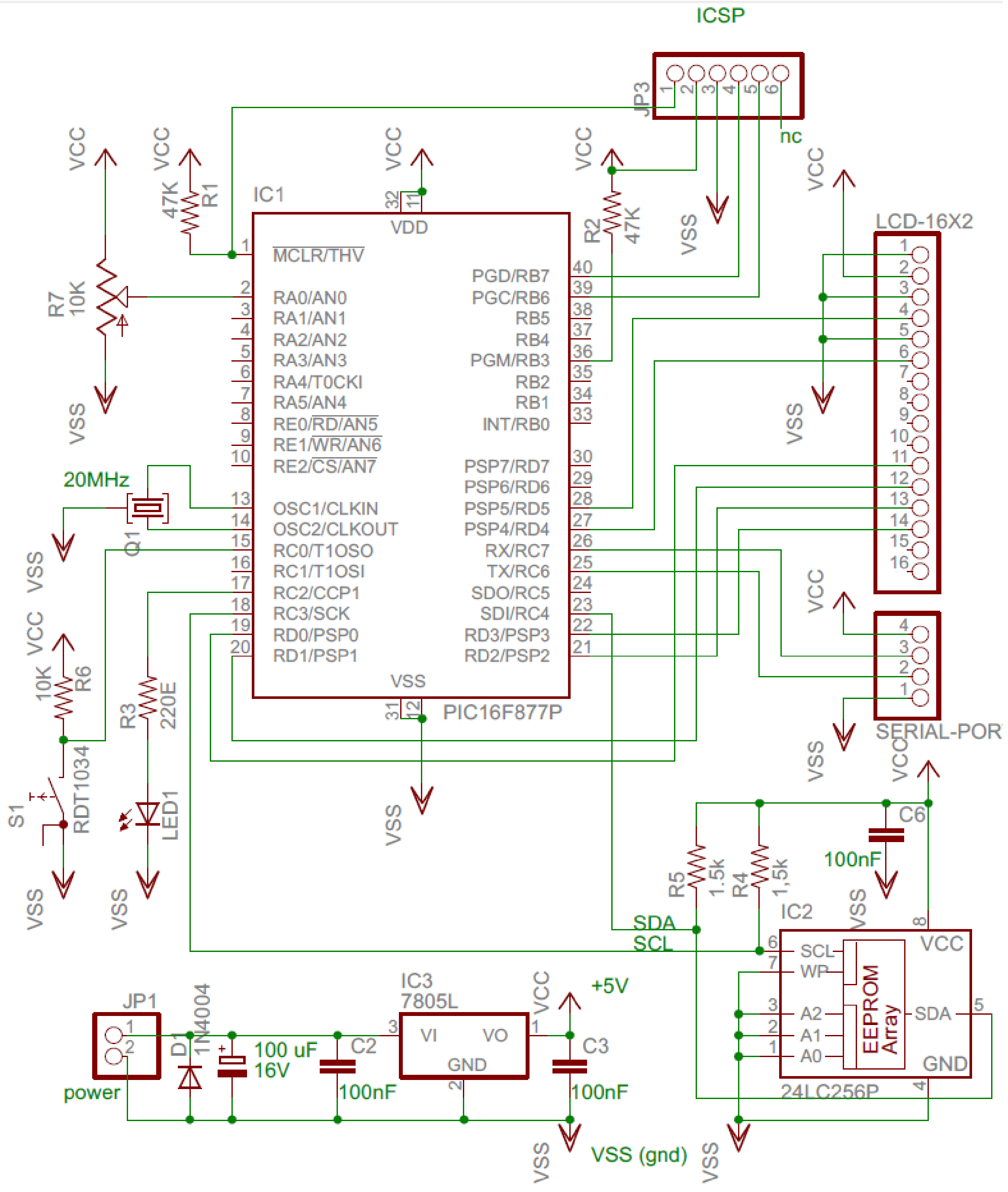


Figure 8

Appendix B – Operators

The operator table below is copied from http://www.casadeyork.com/jalv2/language.html.

Operator	Operation	Result
count	returns the number of elements in an array	UNIVERAL
whereis	returns the location of an identifier	UNIVERSAL
defined	determines if an identifier exists	UNIVERAL
-	Unary – (negation)	Same as operand
!	1's complement	Same as operand
!!	Logical	BIT
+	Unary + (no operation)	Same as operand
*****	Multiplication	Promotion[2]
/	Division	Promotion[2]
%	Modulus division (remainder)	Promotion[2]
+	Addition	Promotion[2]
-	Subtraction	Promotion[2]
<<	Shift left	Promotion[2]
>>[1]	Shift right	Promotion[2]
<	Less than	BIT
<=	Less or Equal	BIT
==	Equality	BIT
!=	Unequal	BIT
>=	Greater or Equal	BIT
>	Greater Than	BIT
&	Binary AND	Promotion[2]
|	Binary OR	Promotion[2]
^	Binary Exclusive OR	Promotion[2]

[1] shift right: If the left operand is signed the shift is arithmetic (sign preserving). If unsigned, it is logical.

[2] promotion: The promotion rules are tricky, here are the cases:

- If one of the operands is `UNIVERSAL` and the other is not, the result is same as the non-`UNIVERSAL` operand.
- If both operands have the same signedness and width, the result is that of the operands.
- If both operands are the same width, and one is unsigned, the result is unsigned.
- If one operand is wider than the other, the other operand will be promoted to the wider type.

Appendix C – Versions

V1.00	March 2010	Initial version
V1.01	July 2012	Fix of i2c connections in figure 7 and 9