Simple **cpik** Tutorial

by Alain Gibaud
alain.gibaud@free.fr
Version 0.5 Rev a

May 15, 2009

Contents

1 What you need

The figure 1 shows a schematic of the experimentaion board used in this tutorial. It has been designed to be as simple as possible. The device can be any 28-pin PIC18 device. Any other

PIC18 can be used, but pinout will be different. The presented software has been written for a PIC18F2525 device, but any other PIC18 can be used. In this case, it will be necessary to modify the CONFIG bits settings and to verify if the used functionnalities are available in the device you use.

Features:

1. No crystal, internal 8Mhz oscillator used. This is sufficient for experimentation but will not be convenient if high timing precision is needed.
2. In situ device programming (no need to unplug the device)
3. Programming connector is compatible with PicKit2 programmer from Microchip (any other TAIT or JDM programmer can be used with the appropriate connexion cable).
4. Board can be powered by an external power supply (15v DC), or from PicKit2 by setting the JP7 jumper. You can use an external power source which is always on: it doesn't prevent device to enter programming mode when programmer sets MCLR to VPP. If external power source is not used, the IC2 7805 regulator and D5 can be suppressed.
5. Data an Clock programming lines are decoupled by 47pF (or 33pF) ceramic capacitors to ground. These capacitors dramatically improve the programming reliability, which is near perfect.

This hardware is easy to build. Figure 2 shows a possible implementation (this is an experimentation board, with wires welded on the bottom side and support for LCD display and RS232 interface). More basic implementations are suitable for this tutorial. You can see a crystal on this picture, but it is not used here. The JP7 jumper is on the left side (red), near the programmer connector.

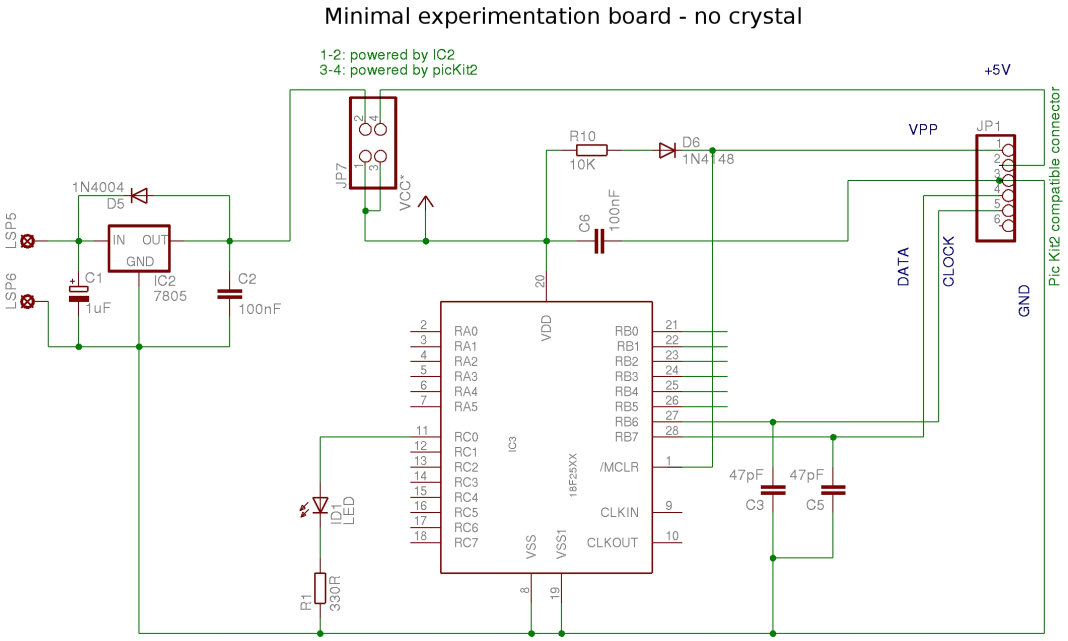

Figure 1: Tutorial #1 - base schematic

2 Tutorial #1 - Blinking LED

The "blinking LED" program is the "hello word" program for microcontrollers: no way to avoid it! It allows to test hardware and software installation.

2.1 Header files

The only mandatory header file to include is the register definition file for the target device. It contains register definitions (PORTA, TRISA, etc.) as well as addresses of these registers. Register definition headers are located in `/usr/share/cpik/<version>/include` so you must use the `#include <xxx>` syntax (not `#include "xxx"`). There is no obvious reason to bypass this inclusion and to write register definitions yourself.

Other headers used here are:

1. `types.h` : allows to use standard ANSI types instead of local ones, for portability purpose. This header typdef `int8_t`, `uint8_t`, `int16_t`, `uint16_t`, and so on.
2. `macros.h` : contains macro for bit manipulation (test, set, reset etc.).

```
#include <p18f2525.h>
#include <types.h>
#include <macros.h>
```

2.2 Config bits settings

Setting config bits is not difficult but needs attention. PIC18 devices have 14 config registers, each of them must be properly populated. For PIC182525 devices, CONFIG1L, CONFIG3L and CONFIG4H do not exist so you should not initialize them.

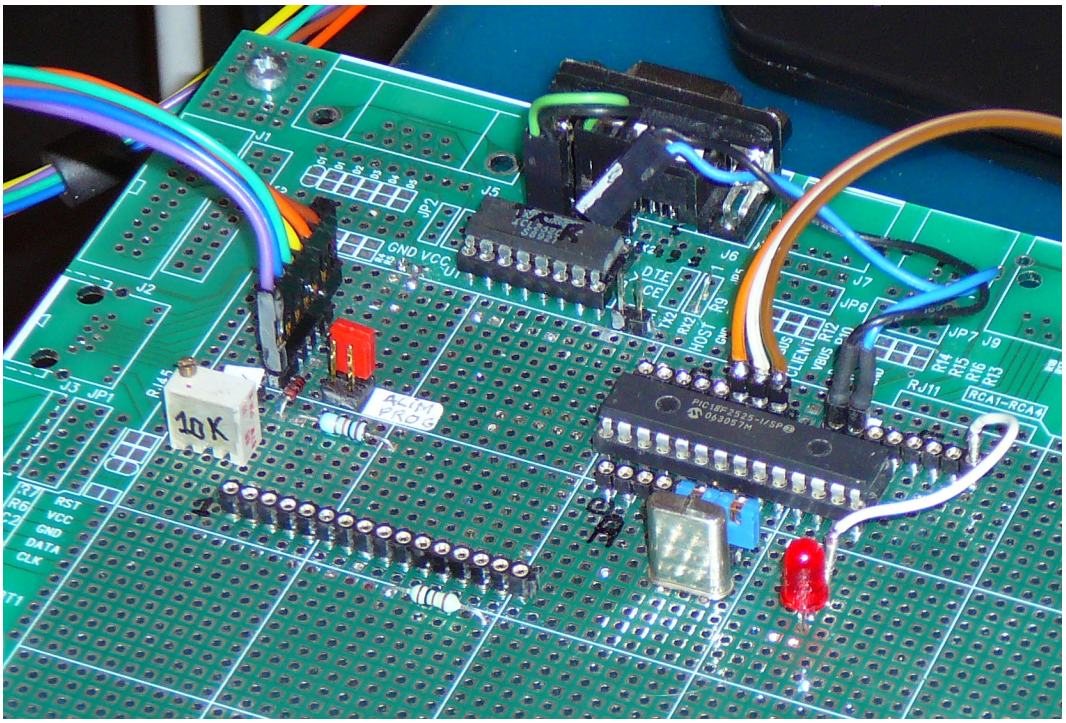

Figure 2: Tutorial #1 - experimentation board

The best way to know how to initialize config bits is to study the device data sheet.

Microchip standard asm headers contain symbolic constants to help config fuse initialisation but I don't feel them very handy. However, you can use these constants with `#pragma _CONFIGxx`, because they are not evaluated by the compiler, but directly passed to the assembler which generates final machine code. If you feel comfortable with them, do not hesitate to use them.

Another approach is to use `pikdev` as an assistant. Just open the programming dialog box (figure 3) choose the config word you want to set, and click on the proper bits. You will be informed about meaning of each bit group by a tooltip. When the configuration is finished, just copy and paste the hexadecimal value witch appears at the top of config box. In this example, CONFIG1H must be set to 0x08 (binary B'00001000').

Here are the config fuses initalization needed by the tutorial, with quick comments.

```
// not used
// #pragma     _CONFIG1L 0xFF
// internal 8Mhz clock, both RA6 anf RA7 are usable as I/O
#pragma   _CONFIG1H B'00001000'
// WDT not  enabled, will be enabled by software if needed
#pragma  _CONFIG2H B'00010000'
// PWRT disabled, BOR disabled
#pragma  _CONFIG2L B'00011001'
// MCLR disabled, no AD on port B
#pragma  _CONFIG3H B'00000000'
// not used
// #pragma _CONFIG3L 0xFF
```

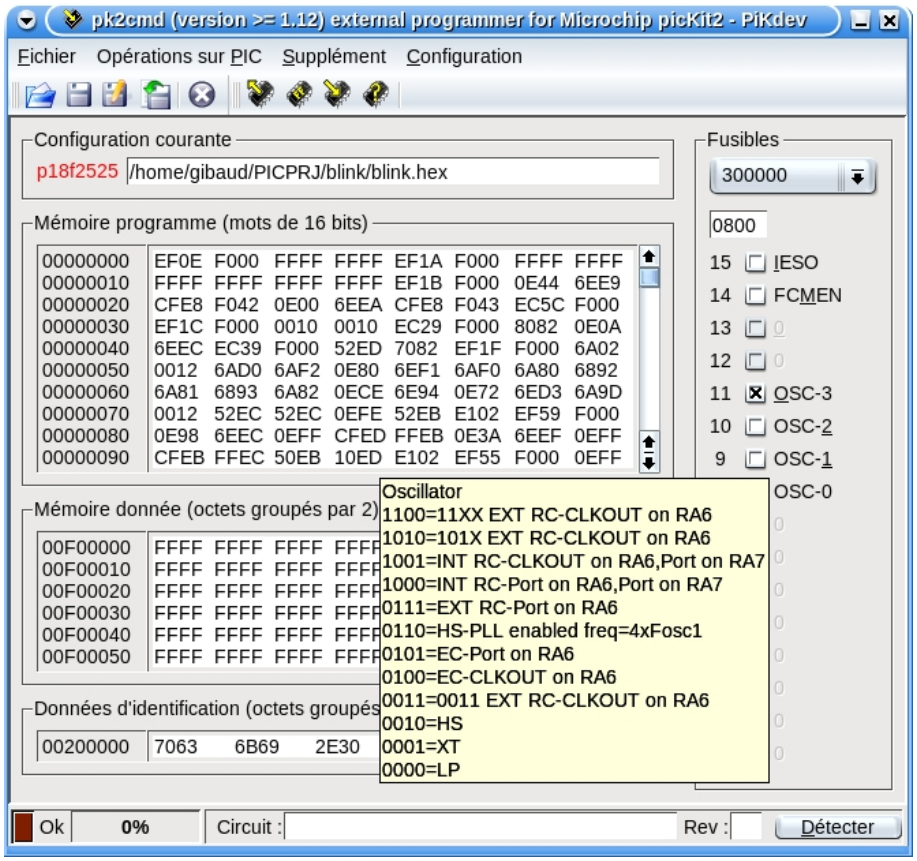

Figure 3: Using pikdev as an assistant for config bits settings (french interface)

```
// not used
// #pragma _CONFIG4H 0xFF
// no DEBUG, reset when stack overflow, no LVP
#pragma  _CONFIG4L B'10000001'
// no protection
#pragma  _CONFIG5H B'11000000'
// no protection
#pragma  _CONFIG5L B'00001111'
// no protection
#pragma  _CONFIG6H B'11100000'
// no protection
#pragma  _CONFIG6L B'00001111'
// no protection
#pragma  _CONFIG7H B'01000000'
// no protection
#pragma  _CONFIG7L B'00001111'
```

2.3 Microcontroller initialization

Here begins the real job. We have to set various device registers to get the desired behaviour. The main points are port direction configuration, interrupt configuration and clock speed configuration. Initializations are grouped in a function, for structuration reason. Device registers are simply accessed as C `unsigned int` variables using the standard names provided by Microchip. All these variables are declared in the `<p18f2525.h>` header, so you don't have to declare them.

```
void MCU_init()
{
   RCON = 0 ;
   INTCON = 0 ;
   INTCON2 = 0b10000000 ; // PORTB pullups disabled
   INTCON3 = 0 ;
   PORTA = 0 ;
   TRISA = 0b11111111 ; // all bits from port A as input
   PORTB = 0 ;
   TRISB = 0b11111111 ; // B as input
   PORTC = 0 ;
   TRISC = 0b11_000000;  // port C<7-6> as input, other pins as output
   OSCCON = 0b01110010 ; // internal clock 8Mhz - see also config bits
   PIE1 = 0 ;  // disable all interrupts
}
```

2.4 Delay function

This program needs a delay to control the blinking speed. The most simple way to do it is an empty loop (in fact, we need 2 nested loops because processor is too fast). The following function provides a $0.1\times$ `i` delay (ie: for a 1s delay, call `delay(10)`). The delay is absolutely not accurate, but it doesn't matter for this application.

```
void delay( uint8_t i )
{
   uint16_t k  ;
   for( ; i  ; --i)
      for( k = 15000UL ; k ; --k)  ;
}
```

2.5 Main function

Finally, here is the `main()` function. As you can see, `main()` returns an `int`, and you may wonder why, because -*in this application*- `main()` never returns. The reason is simple: the ISO standard[1] explicitely specify than `main()` should return an `int`.

I use here two macros (defined in `<macro.h>`): `BIT_1(PORTC, 0 )` sets bit 0 of PORTC, and `BIT_TOGGLE(PORTC, 0 )` toggles it.

```
int8_t  main()
{
   // MCU registers inits
   MCU_init() ;

   BIT_1(PORTC, 0 );
   for( ;; )
   {
      delay(10) ;
      BIT_TOGGLE(PORTC, 0 ) ;
   }
   return 0 ;
}
```

Despite the apparent complexity of `BIT_1` and `BIT_TOGGLE`, the resulting code is very simple. Here is this code (blink1.slb).

```
;<+C18_main> int main()
C18_main
;
;    MCU_init() ;
  ICALL C18_MCU_init
;
;    ((PORTC) |= (1<< (0)));
  bsf C18_PORTC,0,0
;    for( ;; )
L18_blink1_main_10
;    {
;       delay(10) ;
  movlw 10
  movwf PREINC0,0
  ICALL C18_delay
  movf POSTDEC0,F,0 ; clean stack
;       ((PORTC) ^= (1<< (0))) ;
  btg C18_PORTC,0,0
;    }
L18_blink1_main_12
  IBRA L18_blink1_main_10
L18_blink1_main_11
;
;    return 0 ;
  clrf R0,0
; }
L18_blink1_main_9
  return 0
;<?C18_MCU_init>
```

[1] ISO/IEC 9899 standard, section 5.1.2.2.1 "It shall be defined with a return type of `int` ... "

```
;<?C18_PORTC>
;<?C18_delay>
;<-> int main()
```

2.6 How to build the application

It is very easy to use **cpik** from **pikdev** : just open a new C project, choose a processor and add the file `blink1.c` to this project. Then hit `F5` (or select the `assemble` function from menu or toolbar). If you get no error, the hex file is now ready to be transfered to your board (`F6` displays the programmer window).

From command line, this is not difficult too.

1. Compile `blink1.c`
   `cpik -c -p p18f2525 blink1.c`
   cpik generates the file `blink1.slb`

2. Link the application
   `cpik -o blink1.asm -p p18f2525 blink1.slb`
   the "linker part" of cpik links together the "object" file `cpik.slb` and the run-time library (`/usr/share/cpik/<version>/lib/rtl.slb`). We do not need another library here. The generated file is `blink1.asm`.

3. Build `blink1.hex` from `blink1.asm`.
   `gpasm -L -o blink1.hex -w 2 blink1.asm`
   The code is around 54 bytes long, including startup stuff.

3 Turorial #2 - Blinking LED with variable speed

Suppose we need to tune the blinking speed of the LED. For this purpose, we add a minimal 3 buttons keybord to our experimentation board (see figure 4). It is composed of 3 switches, connected to pins `RB<0-2>` of the microcontroller. Resistors `R1`, `R2` and `R3` pull the port pins up to 5V. When a key is pressed, pin voltage is forced to GND, so the buttons are active-low.

The capacitors `C1`, `C2`, `C3` just avoid the switches to bounce. They are not needed if you use very good quality switches or if you implement a software debounce. Resistors could be ommited if you connect the keybord to a port which have pull-up capability. This is the case of port B, but we do not use this feature here.

3.1 A naive approach

To change the blinking speed of the LED we simply put the needed delay in the variable `d`. The keyboard is sampled in the main loop. If a key is found to be pressed (bit == 0) the variable `d` is changed as following:

- key 0 : `d` is incremented, (blinking slows down).
- key 1 : `d` is reset to default value (10)
- key 2 : `d` is decremented, (blinking speeds ups).

The macro `BIT_TST()` apply a mask to its parameter and return `0` if the bit is not set, else a non-nul value considered as `true`. Please see the `macros.h` header file for details.

```
int8_t  main()
{
   uint8_t d = 10 ;
   // MCU registers inits
   MCU_init() ;

   BIT_1(PORTC, 0 );
```

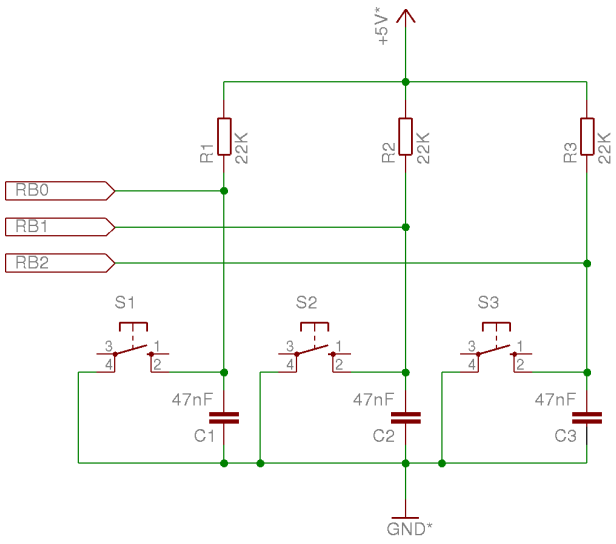

Figure 4: Mini keyboard for tutorial #2

```
   for( ;; )
   {
      delay(d) ;
      BIT_TOGGLE(PORTC, 0 ) ;
      if( !BIT_TST(PORTB, 0))
         ++d ;
      else if ( !BIT_TST(PORTB, 1))
         d = 10 ;
      else if (!BIT_TST(PORTB, 2))
         --d ;
   }

   return 0 ;
}
```

This implementation "works" but is really poor because:

1. The buttons are sampled when the delay is finished, so if the delay is very long, the keyboard is inactive most of the time
2. When the `d` variable is decremented from 0, it becomes equal to `0xFFFF` ($2^{16} - 1$), so you must wait about 1.82 hours before being able to change the speed again. Not really handy.

3.2 Using interrupts: a better approach

I plan here to use interrupts to update the delay variable. I suppose that the delay loop must be resetted each time the delay is changed, so the change will be immediately taken into account. For this pupose the delay variable (`blink_delay`) becomes global. During delay loop, `blink_delay` is continuously sampled and loop is aborted if its value changes.

```
uint8_t blink_delay = 10;

void delay( uint8_t i )
{
   uint16_t k ;
   uint8_t  my_delay  ;
   for(  my_delay = blink_delay; i  ; --i)
      for( k = 5000 ; k ; --k)
         if(blink_delay != my_delay) return  ; // abort delay loop
}
```

PIC18 are able to generate an interrupt when the state of some input pins of port B change. This feature must be configured as following before use: we must decide if interrupt is generated during low to high or high to low transitions, and if interrupts have low or high priority. Please see the PIC18F2525 data sheet for details.

As buttons are active-low, the following code configures the port to emit an interrupt when a high to low transition occurs.

After this configuration, high priority interrupts are enabled by the `UNMASK_HI_PRI_IT` macro, which is defined in `<interrupt.h>`. Please see the "Interrupt Service Routine" and "Disabling and enabling interrupts" sections of `cpik` documentation for details.

The rest of `main()` function doesn't contain any code to manage keyboard: this point becomes the job of Interrupt Service Routine.

```
#include <interrupt.h>
```

```
int8_t  main()
{
   // MCU registers inits
   MCU_init() ;


   // INT0 on falling edge
   BIT_0(INTCON2, INTEDG0) ;
   // INT1 on falling edge
   BIT_0(INTCON2, INTEDG1) ;
   // INT2 on falling edge
   BIT_0(INTCON2, INTEDG2) ;

   // set IT hi priority
   BIT_1(INTCON3, INT1P) ;
   // set IT hi priority
   BIT_1(INTCON3, INT2P) ;
   // note: INT0 has always hi priority

   // enable INT0
   BIT_1(INTCON, INT0E) ;
   // enable INT1
   BIT_1(INTCON3, INT1E) ;
   // enable INT2
   BIT_1(INTCON3, INT2E) ;

   //  enable hi pri interrupts
   UNMASK_HI_PRI_IT ;

   BIT_1(PORTC, 0 );
   for( ;; )
   {
      delay(blink_delay) ;
      BIT_TOGGLE(PORTC, 0 ) ;
   }
   return 0 ;
}
```

For PIC18 devices, interrupts may have either low or high priority. Low priority Interrupt Service Routines (ISR) can be interrupted by high priority interrupts, but high priority ISR cannot be interrupted.

With `cpik`, ISRs must be named `lo_pri_ISR` or `hi_pri_ISR`, depending on the kind of interrupts they serve. ISR must preserve informationnal context. `cpik` provides two macros for context saving: `SAVE_REGS` and `RESTORE_REGS`.

The following routine test interrupt flags located in `INTCON` and `INTCON3` in order to detect where the interrupt comes from. Interrupt flags are cleared before the ISR returns.

Note the `__interrupt__` nonstandard keyword which insures that function is considered as an ISR.

```
__interrupt__ void hi_pri_ISR()
{
   SAVE_REGS ; //  defined in interrupt.h

   if ( BIT_TST(INTCON, INT0F) )
```

```
   {
      ++blink_delay; // key 0 -> slow down blink
      BIT_0(INTCON, INT0F) ;
   }
   else if ( BIT_TST(INTCON3, INT1F) )
   {
      blink_delay = 10; // key 1 -> set speed to default
      BIT_0(INTCON3, INT1F) ;
   }
   else if ( BIT_TST(INTCON3, INT2F) )
   {
      --blink_delay ; // key 2 -> speed up blink
      BIT_0(INTCON3, INT2F) ;
   }

   RESTORE_REGS ;
}
```

Thanks to interrupts, keyboard is always responsive and blinking delay can now be changed at any time. The "roll from 0 to 0xFFFF" issue is still here but can be easily fixed.

4 Tutorial #3 - Using an HD44780 compatible LCD display

This tutotrial will show how to use an HD4478 based LCD display from `cpik`. We will use a pre-defined library (`lcd.slb`) , wich is written in assembly language. You can copy this library (located in `/usr/share/cpik/<version>/lib/`) to your working directory, or reference it directly at link time, using the `-L` option.

The LCD display is used in 4 bit mode (ie: each byte is transmitted to display as two nibbles). this solution allows to use only 6 bits from port B, as showed in figure 5. However, these details are totally hidden by the API provided by the LCD library.

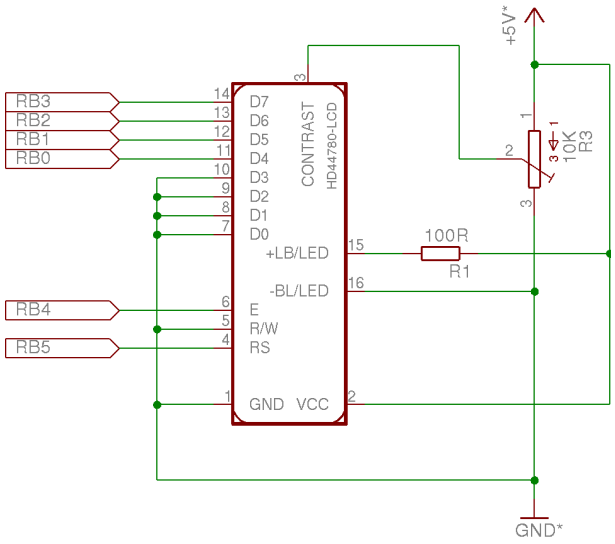

Figure 5: How to connect an LCD display in 4 bit mode

4.1 Using the LCD library API

Our goal is just to set up the display, and to display an "hello" message.

As the display is connected to port B, we need to update the initialization routine used for previous examples to configure all pins of port B as output.

```
void MCU_init()
{
   RCON = 0 ;
   INTCON = 0 ;
   INTCON2 = 0b10000000 ; // PORTB pullups disabled
   INTCON3 = 0 ;
   PORTA = 0 ;
   TRISA = 0b11111111 ; // all bits from port A as input
   PORTB = 0 ;
   TRISB = 0b00000000 ; // B as output
   PORTC = 0 ;
   TRISC = 0b11111111;  // port C as input

   OSCCON = 0b01110010 ; // internal clock 8Mhz (no crystal) see also config bits

   PIE1 = 0 ;  // disable all interrupts
}
```

The main routine is easy to write : after CPU initializations, we call the LCD initialization routine, `lcd_init()`. The parameter 15 is just used by an internal delay routine, and is convenient for a 8MHz clock CPU. Please see the cpik documentation if you use another clock speed. This parameter is not critical, but depends on the the display you use. If you get incorrect behaviour, increase this value.

```
int8_t  main()
{
   // MCU registers inits
   MCU_init() ;

   lcd_init(15) ; // delay loop  value for 8MHz clock

   lcd_clear() ;  // erase display

   lcd_move(0x46) ;  // go to 2nd line, 6th column

   lcd_puts("Hello LCD !") ;  // my message

   for( ;; ) ; // infinite loop

   return 0 ;
}
```

The `0x46` value corresponds to the physical address of the cursor target location on display. This kind of specification is not very handy, because it depends on line an column numbers, but also on the display you use (8, 16, 20, 24 or 40 characters per line displays use different addresses).

To make the code less dependent on the physical display, the following routine returns the cursor address of the line/col location.

```
#define LCD_COLS 20

uint8_t lcd_cursor_addr( uint8_t line,  uint8_t col)
{
#if LCD_COLS == 16
   static uint8_t laddrtab[4] = { 0x0, 0x40 , 0x10, 0x50 } ;
#elif LCD_COLS == 20
   static uint8_t laddrtab[4] = { 0x0, 0x40 , 0x14, 0x54 } ;
#elif LCD_COLS == 24
   static  uint8_t laddrtab[4] = { 0x0, 0x20 , 0x40, 0x60 } ;
#else
   #error "LCD_COLS should be either 16 20 or 24"
#endif
   return   laddrtab[line]+ col ;
}
```

It is now easy to convert line/column coordinates to LCD addresses, assuming the first line is the line 0.

s You can now replace `lcd_move(0x46) ;` with a more readable `lcd_move(lcd_cursor_addr(1,5)) ;` statement.

4.2 Using the `stdio` API

Sometime, using the standard `printf()` function is the most handy way to output a text. This example shows a pretty complicated way to write the "Hello LCD" text.

The most important point is the following: before using `printf()`, you must redirect the output of `printf()` to the LCD device, by using the `set_putchar_vector()` routine. The output vector remains active as long as you do not change it again.

```
int8_t  main()
{
   // MCU registers inits
   MCU_init() ;

   lcd_init(15) ; // 8 MHz clock
   // redirect standard output to LCD
   set_putchar_vector(&lcd_putchar) ;

   lcd_clear() ;

   lcd_move(lcd_cursor_addr(1, 5)) ;

   printf("Hello %c%s", 'K'+1, "CD !") ;

   for( ;; ) ;
   return 0 ;
}
```

5 Tutorial #4 - A digital Voltmeter

This section shows how to build a simple, but effective voltmeter. We will use the analog to digital converter included in most PIC18 devices.

For testing purpose, a voltage source may be provided by an ajustable resistor, wired as voltage divisor, as shown in figure 6. Microchip recommends to use a voltage source with an output impedance less than 2KΩ, so a 470Ω adjustable is OK.

If you plan to use a high impedance voltage source, you must add a buffer. A simple and reliable buffer can be built with an operational amplifier, wired as voltage follower, as also shown in figure 6. The same amplifer could also be used to apply a gain to the measured voltage.

The 22nF (or more) capacitor allows to minimize effects of poor quality ajustable, and acts as an "energy tank" if fast AD conversion is needed on hi-impedance sources.

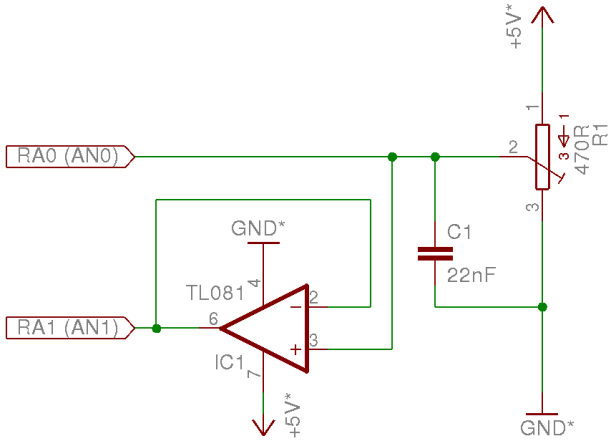

Figure 6: Unbuffered (on RA0) and buffered (on RA1) voltage sources for AD converter

5.1 Analog to Digital module configuration

We plan to use the A/D module on channel 0 (AN0), which corresponds to pin RA0 of the 18F2525 device. For simplicity, we will use the module in polling mode (ie: without interrupts).

The A/D module needs two voltage references: VREF+ and VREF-. When the converted voltage is equal to VREF-, the converter returns the minimum value (ie: 0) and when the voltage is equal to VREF+, the converter returns the maximum value (ie: 1023). Voltages outside these limits are illegal.

In order to get optimal results, a very precise and constant voltage reference is needed. Integrated chips such as LM336 or REF25Z are convenient for that purpose. For simplicity, we use GND as VREF-, and VCC as VREF+, and we will suppose that VCC is exactly 5V[2].

The A/D module works in two stage. The first one is the acquisition. It consists to charge an internal capacitor with voltage source. When charged, the capacitor is disconnected from the source and connected to the converter itself. This capacitor is a "voltage memory" which retains the voltage value during conversion. After the "sample and hold" stage, the conversion stage begins. Because the hold capacitor is not perfect (and also because converter sink a little current), its charge tends to vanish with time and voltage to decrease. For this reason, the conversion process must be as fast as possible.

[2]As this is rarely the case, do not use this solution if accuracy is needed.

When conversion is finished, a special bit is reset, to tell the program that a result is available.

The A/N module can be initialized as following:

1. Choose VREF+ and VREF-,
2. choose which pins are usable as analog inputs,
3. select an acquisition time which ensure a full charge of the sample and hold capacitor,
4. select conversion clock, as fast as possible (but not too fast for the converter capabilities),
5. disable interrupts,
6. enable A/D conversion subsystem.

Please refer to Microchip documentation for details about ADCONx registers.

```
void AD_init()
{
   // AN0=analog input, other pins=digital , VREF- =GND, VREF+ = VCC
   ADCON1 = 0b00_00_1110 ;
   // AD conv freq = Fosc/32 (OK for 8 Mhz device clock), acquisition time = 16 TAD,
   ADCON2 = 0b10_110_010 ;
   /* RA0..RA7 as inputs */
   TRISA |= 0b11111111 ;

   /* no AD conversion termination interrupt */
   BIT_0(PIE1, ADIE) ;
   /* enable  AD */
   BIT_1(ADCON0, ADON) ;
}
```

5.2 Analog to Digital conversion

Conversion itself is easy to program:

1. Load `ADCON0<5-2>` with channel number.
2. set `ADCON0<GO>` bit, to start the acquisition process (sample and hold, then conversion)
3. wait for `ADCON0<GO>` bit to be resetted by hardware (this is done by a simple loop polling the bit)
4. return the value of `ADRESL/ADRESH` register

This function uses the ADRESL (0xFC3) and ADRESH (0xFC4) registers as a 16 bit pair. For this purpoase an external, manually located, variable ADRESHL is declared.

```
uint16_t  ADRESHL@0xfc3 ;

uint16_t  AD_get(uint8_t channel )
{

   // choose  channel
   ADCON0 &= 0b11000011 ;    // mask channel bits
   ADCON0 |= (channel << 2) ;  // then set channel number
```

```
   // start measurement
   BIT_1(ADCON0, GO);

   while( BIT_TST(ADCON0, GO) ) ; // wait for conversion to be done

   // CAUTION: symbol ADRES exported by p18f2525.h is an unsigned int8
   // use ADRESHL variable to get the result as an unsigned int16
   return ADRESHL ;
}
```

5.3 Voltage display

The value AD returned by the A/D converter must be converted to volts before being displayed.

$$Voltage = \frac{VREF+ \ - \ VREF-}{1023} \times AD = \frac{5V}{1023} \times AD$$

With the current configuration, the resolution of the converter is approximativly $0.005V$, so it seems reasonable to display voltages with 2 digits after the decimal point.

As no floating point type is available yet, the conversion must be done in 10mV units, using 16 bit unsigned integers.

$$Voltage = \frac{500}{1023} \times AD$$

To avoid rounding error, multiplication must be done before division. Unfortunatly, this will lead to overflow in 16 bit mode, because 500×1023 is not representable with a 16 bit unsigned integer. For this reason, one can use the following approximation:

$$Voltage = \frac{50}{102} \times AD$$

which leads to an acceptable 0.29% error.

Since version `0.5`, `cpik` supports 32 bit integers, so a better solution is to perform the conversion with 32 bit arithmetic. The 32 bit result can be casted back to 16 bit variable without any risk, because it cannot overflow.

After conversion, the voltage must be displayed like a floating point value, with two digit following the decimal point. Note that an `uint16_t` data is an `unsigned long`, so we have to use a "`%lu`" format specification.

```
void display_voltage(uint16_t v)
{
   uint16_t  vi = v / 100UL ;
   printf("%lu.", vi) ;
   vi = ( v % 100UL) ;
   if (vi < 10UL) putchar('0') ;
   printf("%lu v", vi) ;
}
```

Finally, here is the `main()` function. Thanks to implicit type promotion, the voltage conversion is done using 32 bit arithmetics, despite the fact `voltage` is a 16 bit variable:

`voltage = voltage * 500ULL / 1023` is indeed equivalent to
`voltage = (uint16_t)((uint32_t)voltage * 500ULL / (uint32_t)1023)`

```
int8_t  main()
{
   uint16_t voltage ;

   // MCU registers inits
   MCU_init() ;
   // A/D module initialization
   AD_init() ;

   lcd_init(15) ; // 8 MHz clock
   set_putchar_vector(&lcd_putchar) ;
   lcd_clear() ;

   for( ; ; )
   {
      voltage = AD_get(0) ;
      // convert to 10 mV units
      voltage = voltage * 500ULL / 1023 ;
      lcd_move(lcd_cursor_addr(0, 0)) ;
      display_voltage( voltage) ;
   }

   return 0 ;
}
```

5.4 Voltmeter with bargraph display

To make our display more user friendly, we now display the voltage in an analog manner, with a bargraph, as presented in figure 7.

For this purpose, we need special characters which are not available in the LCD display ROM. Hopefuly, the HD44780 allows to upload 10 user-defined characters. Each new character must be defined in a 5x8 matrix. The characters we need must be 5, 4, 3, 2 or 1 pixels wide. By joining together 10 such characters, we can display an acceptable "analog" bar.

Here is the characters definition: each character is coded by 8 bytes, each byte being the image of the one line of the matrix. All character definitions are grouped in a static array of arrays, local to the initialization routine `define_bargraph_chars()`[3].

```
void define_bargraph_chars()
{
   /* user-defined chars for progress bar */
   static char chr[][8] =
      {
         {0} , // 0 bar
         { 0, 0, 0b00010000,  0b00010000 ,  0b00010000,  0b00010000, 0, 0 }, // 1 bar
         { 0, 0, 0b00011000,  0b00011000 ,  0b00011000,  0b00011000, 0, 0 }, // 2 bars
         { 0, 0, 0b00011100,  0b00011100 ,  0b00011100,  0b00011100, 0, 0 }, // 3 bars
         { 0, 0, 0b00011110,  0b00011110 ,  0b00011110,  0b00011110, 0, 0 }, // 4 bars
         { 0, 0, 0b00011111,  0b00011111 ,  0b00011111,  0b00011111, 0, 0 }  // 5 bars
      } ;
```

[3]The advantage of this solution is to make impossible any identifier collision with a global entity.

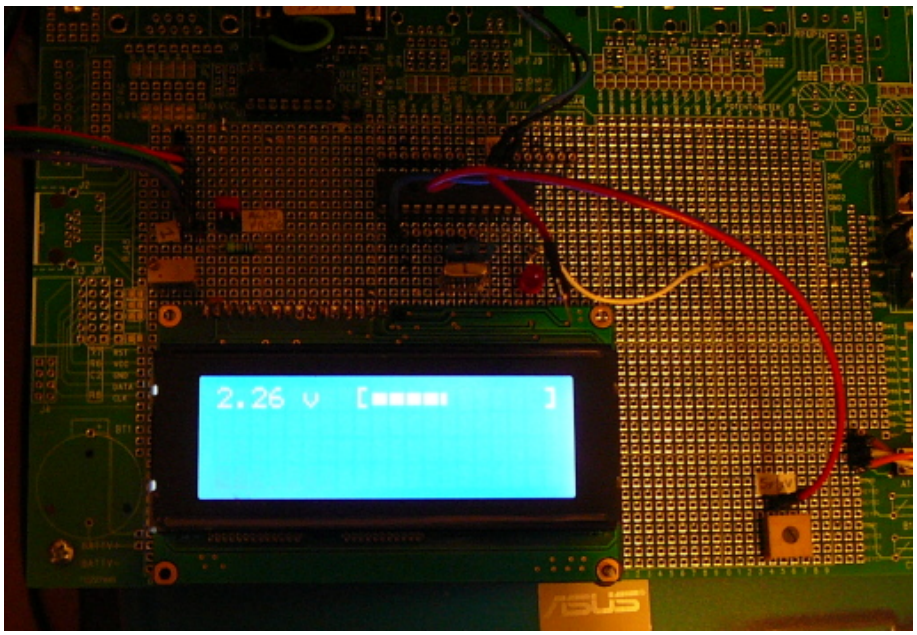

Figure 7: Voltage display, with bargraph

```
   uint8_t  i ;
   for(i = 0 ; i < 6U ; ++i)
      lcd_define_char(i, chr[i]) ;
}
```

The `voltage_bar()` routine displays the bar corresponding to the "`v`" voltage. As maximum voltage is 5.00V and bar is 10 characters wide, each character of this bar corresponds to a 0.5V increment. As characters are 5 pixels wide, each pixel represents 0.1V.

The characters coding has been choosen as following: character with code 0 represents 0.0V , character 1 represent 0.1V, etc.

The display algorithm decompose the voltage into 0.5V slices, and display character 5 as many time as necessary to represent each slice. The rest of the decomposition is either 0.4V, 0.3V, 0.2V, 0.1V or 0.0V, so we just need to display the proper character to complete the bar. Finally, a sequence of blanks is displayed up to 10 characters.

```
void voltage_bar(uint16_t  v)
{
   uint8_t n = 0 , step, charcod   ;

   // displays progress bar
   for(step = 50U, charcod = 5 ; step >= 10U ; step -= 10U, --charcod )
   {
      while( v >= step)
      {
         printf("%c", charcod) ;
         v -= step ;
         ++n ;
      }
   }
   // adds  spaces if needed
   for( ; n < 10U ; ++n) printf(" ") ;
```

```
}
```

The main function is very simple :

1. Initialisations,
2. user-defined characters upoload and display erase,
3. data acquisition,
4. data to voltage conversion,
5. numeric display,
6. analog display,
7. repeat process from step 3

```
int8_t  main()
{
   uint16_t voltage ;

   // MCU registers inits
   MCU_init() ;
   AD_init() ;

   lcd_init(15) ; // 8 MHz clock
   set_putchar_vector(&lcd_putchar) ;
   define_bargraph_chars() ;

   lcd_clear() ;

   for( ; ; )
   {
      voltage = AD_get(0) ;
      // convert to 10 mV units
      voltage = voltage * 500ULL / 1023 ;
      lcd_move(lcd_cursor_addr(0, 0)) ;
      display_voltage( voltage) ;
      lcd_move(lcd_cursor_addr(0, 8)) ;
      printf("[") ;
      voltage_bar(voltage) ;
      printf("]") ;
   }

   return 0 ;
}
```

5.5 Contact

Please send remarks, suggestions, topic requests or error reports to: `alain.gibaud@free.fr`