cpik

C compiler for pic 18Fxxx

Alain Gibaud
alain.gibaud@free.fr
Version 0.4

January 21, 2009

Contents

1 Introduction

cpik is a C compiler which generates code for Microchip PIC 18 microcontrollers. The language recognized by this compiler is a subset of ANSI C language, but contains extensions specific to microcontroller domain. It has been developped 3 years ago, but never really released because I had no time to test it properly, for example for a real application development.

2 What `cpik` does (or doesn't), and why

My idea was to develop a compiler as simple as possible but conformant to the ANSI specifications. This is a huge work for a single developper (with many other activities), so I had to decide what is important and what is not. My underlaying idea is the following: it is better to drop a feature than to incompletely or inexactly implement it.

For example, I choosed to suppress bit fields support because bit fields manipulations can be easily performed using standard C operators such as `&`, `|`, `^` and so on.

I also dropped the `switch` statement, because it it always possible to replace a *switch* statement with cascaded if. The resulting code can be less efficient, but works.

The first version of `cpik` (0.2) did not recognize `typedef` instruction, and had no support for `struct`s or `union`s. `typdef` has been implemented in V0.3, and structures in V0.4.

Finally, two major issues remain in current version (0.4): floating point support is not implemented yet, and 32 bit integer arithmetic is still missing .. but is already in my head. The roadmap is the following: 32 bit arithmetic support is scheduled in V0.5, and floating point in V0.6.

Despite these drawbacks, `cpik` is very usable and produces good code. It is well supported by `pikdev` (my IDE for pic processors) so the `pikdev/cpik` couple is really very handy and pleasant to use.

Volunteers are welcome for any help, including *tests*, *benchmarking*, *documentation* and *libraries writing*. Please see the section ≪How to contribute to the `cpik` project ?≫ for details.

This compiler (written in C++) is recent, so any feedbacks concerning bugs, feature requests or criticisms can be addressed to Alain Gibaud (`alain.gibaud@free.fr`). `cpik` has been used to write an application which is 1600 lines long (including `stdio` library which is written in C). This application runs continuously 24/24 7/7 (x 2 items) and works perfectly since six months. Of course, this is not a proof that everything is perfect, by I feel this fact very encouraging. Send me informations about your project with cpik.

3 A special feature

cpik works in a unsusual way: unlike other compilers, it does not produce ordinary assembler code but *source libraries*.

A source library looks like a PIC 18 asm source file, with `.slb` extension. This file can be processed by an assembler (such as `mpasm` or `gpasm`) but contains special comments which are intended to be used as *directives* by an ad-hoc linker. This linker is included in cpik executable, so the `cpik` command can be used for both compilation and link tasks.

The important point is that cpik linker *works at assembly source code level*: it picks needed "modules" from source libraries and copies them in a single output file. In other words, `cpik` performs linking before assembly stage (on the opposite, other linkers work on the output of the assembler, which is object code).

The file generated by the linker is easy to manually verify, and I suppose (and hope) that advanced users will examine it and will report feedbacks about code.

This unusual approach presents for me several advantages:

- Any source library is simply a text file so it can be manually builded using assembly language and your favourite text editor (this point is important to bootstrap a totally new development environment)
- source libraries do not depend on any object/library format, and/or obscure, undocumented and volatile format versions.
- final executable code (ie: hex file) can be generated by a very simple assembler without any advanced feature (in fact, the target assembler is currently `gpasm` running in absolute mode - ie: without program sections)
- any output from the compiler is potentially a library, so there is no more differences between *object files* and *libraries*. As a consequence, we do not need any librarian utility.
- linking process is globally very simple and does not increase signicatively the complexity/size of `cpik` compiler/linker

In fact, the source file library approach might be rather slow, but, as microcontrollers applications are not huge, your computer will build ready-to-burn hex files at speed of light.

4 Command syntax

The `cpik` command can be used for both compilation or linking tasks, exactly as `gcc` frontend does. However, `cpik` is not a frontend and really merge this two functionnalities.

4.1 Compilation

```
cpik -c [-v] [-o output_file] [-I path] [-p device] [-d<value>] input_file
```

`-v` : print version number, then exit.

`-o output_file` : specify the output (source library) file name. By default, this name is generated from the source file name by appending `.slb` to the extensionless input file name.

`-I path` : specify the path to include (`.h`) files. This option follows the traditionnal behaviour of Unix C compilers. You can specify any number of include path, and they will be searched in the order of `-I` options. As usual, use "`-I .`" to specify the current directory. If your header file

is located in the default system directory (ie: `/usr/share/cpik/<version>/include/`), do not forget to use `#include <xxx>` instead of `#include "xxx"` in your source code.

`-p device` : specify the target pic device name. `device` must be a valid pic 18 name like `p18xxxx`. The exact name is not verified, excepted for `p18` prefix. And invalid device will cause the final assembly to fail. By default, the selected device is `p18f1220`.

`-d<value>` : debug option, *used for the development/debugging of the compiler itself.* The value is an integer which specify what debug information should be printed. Any number of `-d` options can be used.

value	meaning
`-d1`	print unoptimized intermediate code as comment in .slb file
`-d2`	print peep hole optimized intermediate code as comment in .slb file
`-d4`	print symbol tables with entities names and types
`-d8`	print internal expression trees before optimisations, with no type annotation
`-d16`	print internal expression trees before optimisations, with type annotations
`-d32`	print internal expression trees after optimisations, with no type annotation
`-d64`	print internal expression trees after optimisations, with type annotations

Use of `-d` option is never useful for normal operations with `cpik`. Produced outputs are hard to read for non developers.

`input_file` : specify source file name, with `.c` extension.

This command cannot be used to compile more than to one source file.

4.2 Link

```
cpik [-v] [-o output_file] [-L path] [-p device] input_file [input_file..]
```

`-v` : print version number, then exit.

`-o output_file` : specify the output (pure asm) file name. By default, this name is `a.asm`.

`-L path` : specify the path to libraries (`.slb`) files. This option follows the traditionnal behaviour of Unix C linkers. You can specify any number of lib path, and they will be searched in the order of `-L` options. The default include path always contains `/usr/share/cpik/<version>/lib/` which is searched in last position.

`-p device` : specify the target pic device name. `device` must be a valid pic 18 name like `p18xxxx`. The exact name is not verified, excepted for `p18` prefix. An invalid device will cause the final assembly to fail. By default, the selected device is `p18f1220`.

`input_file` : any number of `.slb` files. The library `/usr/share/cpik/<version>/lib/rtl.slb` (run time library) contains low-level modules for **cpik** generated code and is automatically referenced as the *last* library. Please do not reference this library yourself because it will change the order of library scanning, and might cause undesirable effects.

5 Implementation details

cpik generates code for PIC 18 processors running in legacy (ie: non-ehanced) mode. PIC 18 CPU is fundamentally a 8 bit processor with 16 bit pointers and distinct program/data spaces. From the C programmer point of view, up to 64K bytes of program space and 64K bytes of data space are available. Pointer generally points to data space, but pointer to function points to program space. In fact, programs can be larger than 64K bytes (depending on your device flavour), but only the lower 64Kbytes can be reached from pointer to functions. This is not an issue because it is easy to locate such function at addresses lower than `0xFFFF`.

`cpik` has been designed to produce stack based code. This kind of code is easy to understand, robust and potentially reentrant without any trick. Interruptions are easy to support (see Interrupt Service Routine section for details). Thanks to autoincremented and indirect adressing modes, this design leads to efficient code.

Memory space is flat and covers the totality of program/data spaces. `cpik` is based on a *unique memory model* (no banks, "small" stacks, "far" pointers or other tricky ways to save memory but to confuse programmers and compiler).

5.1 Memory usage

cpik-generated code uses two stack:

- *hardware return stack* (31 levels):

  This stack is part of the PIC18 architecture. It is only used to save return addresses for subroutines. It might be insufficient for really recursive intensive applications, but is quite suffcient for "normal" ones.

- *software data stack*:

  This stack is used to store local variables, function parameters and temporary results during expression evaluation. Due to the availability of address registers `FSRx`, and indirect, auto-incremented, and indexed addressing mode, the stack usage is very efficient. `FSR0` is used as the software stack pointer.

  `cpik` currently reserves a pool of memory at memory bottom. The size of this pool can be computed as *2+size of the biggest stucture used in the program*. However, this size cannot be less than 10 bytes. The default size of reserved pool is 66 bytes (it means that maximun default structure size is 64 bytes). In the case of device with small RAM, you can reduce the pool size by editing the prolog file
  `/usr/share/cpik/<version>/lib/cpik.prolog`.
  However, I plan to implement a more flexible way to specify pool size in the future.

  Bytes above pool are used for static data, and data stack is located above them.

  Stack grows upward and is used in a pre-incremented manner: pushing a byte onto the stack uses a `movxx source,PREINC0` instruction. Symmetrically, a `movxx POSTDEC0,dest` is used to pop the data back.

5.2 Register usage

`cpik` uses four 16 bit pseudo registers named `R0`,`R1`,`R2`,`R3`. Each `Rx` register can be splitted in `RxH` and `RxL`. These registers are located in page 0, and are efficiently accessed via Access Bank (a=0).

- `W` is used as a general purpose scratch register
- `R0` is the 16 bit equivalent of W,
- `R1` to `R3` are used by Run Time Library,
- `FSR0` is the software stack pointer,
- `FSR1` is a general purpose address register,
- `FSR2` is used for fast memory moves together with `FSR1`,
- `PRODL` and `PRODH` are used for arithmetics and temporaries

Of course, indirect adressing registers such as `INDFx`, `PREINCx`, `POSTDECx`, and `PLUSWx` are intensively used and also accessed in Access Bank for efficiency reasons.

5.3 Computing model

- *2 operands operators* are executed with 1st operand on the stack, and 2nd one in `W` (8 bit) or `R0` (16 bit) or `R0-R1` (32 bit). The result replaces the 1st operand on the stack, but may have a different size.
- *1 operand operators* take their operand from top of stack and replace the result at same location.

In fact, resulting code might be quite different depending on various optimizations done during compilation.

Due to hardware limitation, the total amount of local non static data used by a function can't exceed 127 bytes. This point covers parameters, local variables, and temporaries. Excepted for very complex expressions, temporaries never exceed a few bytes, so, as a rule of thumb, about 100 bytes are always available.

Here is an example of simple code

```
int  h(int u, int v)
{
  return (u+v)/3  ;
}
```

and the commented resulting module

```
C18_h
  movff INDF0,PREINC0 ; push u onto the stack
  movlw -2
  movf PLUSW0,W,0     ; move v to W
  addwf INDF0,F,0     ; replace stacked u by u+v
  movlw 3
  ICALL div8          ; divide top of stack data by 3
  movff POSTDEC0,R0   ; pop result to R0L
  return 0
```

5.4 Function calling conventions

All parameters are passed to function thru the software stack. They are stacked in reverse order (1st parameter pushed last). Stack cleaning is performed by caller[1].

These characteristics are common for C code because they are useful to implements functions with variable number of parameters, such as `printf`.

8 bit results are returned in `R0L` register, and 16 bit results are returned in `R0` register.

Here is a call of the previous function `h(int u, int v)`:

```
void caller()
{
  int res, k ;
  res = h(k, 25) ;
}
```

and the resulting code

[1]No aligment is done during parameter passing, so a 16 bit local data can be located at odd or even address.

```
C18_caller
  movf PREINC0,F,0          ; reserve stack space
  movf PREINC0,F,0          ; for k and res
  movlw 25
  movwf PREINC0,0           ; push param 25 onto the stack
  movlw -1
  movff PLUSW0,PREINC0      ; push  parameter k
  ICALL C18_h               ; call h()
  movf POSTDEC0,F,0         ; (partially) clean stack
  movff R0,INDF0            ; move result to temporary
  movlw -1                  ; pop result to res and
  movff POSTDEC0,PLUSW0     ; finish to  clean stack
  movf POSTDEC0,F,0
  movf POSTDEC0,F,0         ; (discard local variables)
  return 0
```

5.5 Optimizations

`cpik` performs many optimizations, but not all possible optimizations. Optimizations can be performed during code analysis, intermediate code generation or final (asm) code generation.

1. *NOP removal*

   Some expressions which has no effect are simply removed. For example `i = i + 0 ;` does not produce any code.

2. *Register value tracking*

   Value of `W` register is tracked whenever possible, so it is not reloaded when it contains the proper value. This feature only concern the `W` register, but I plan to extent it to `FSR1` register.

3. *Constant folding*

   Most constant subexpressions are computed by compiler at compilation time, so complex expressions are often compiled as a single constant (ie: `x= (1+2*4) << 1 ;`). However, a lot of constant foldings are done by peephole optimizer or expression simplifier (for example, `st.a.b.c = 34 ;` have exactly the same cost as `x = 34 ;`)

4. *Peephole optimization*

   Intermediate code is analyzed by grouping instructions into slices of 2, 3 or 4 items. Each slice are compared against a library of possible simplifications or transformations. Then simplified code is simplifed again, and so on. This optimization may lead to 25% to 50% gain.

5. *Code generator optimization*

   This is the only phase which depends on target architecture. Bit operations are excellent candidates for optimization. For example, I often use the following macro to reset a single bit:

   ```
   #define CLRBIT(var,bit) ((var) &= ~(1 << (bit)))
   ```

   so

   ```
   CLRBIT(i,3) ;
   ```

   is expanded as

   ```
   ((i) &= ~(1 << (3))) ;
   ```

which is optimally translated as:

```
bcf INDF0,3,0
```

This example is a combination of constant folding and code generator optimizations.

`cpik` does not perform yet any global analysis, so common subexpression optimization, or dead code removal are out of scope. Anyway, code needing such optimizations reveals poor programming style because the C language offers may ways to "hand optimize" code.

5.6 Constant data Management

PIC 18 processors are based on two separate address spaces (program and data spaces). This architecture causes problem to store initialized data (such as literals like `"hello !"`). For a compiler the only way to store initialized data is to locate them in program space. Unfortunatly, literals have to be accessed as ordinary data (ie: locating in data space) during program execution.

In order to keep the compiler simple, `cpik` adds a loader routine to startup code, which copies all initialized data from program space to data space. This routine is automatically activated before `main()` function.

This loader is not included if your program does not use statically initialized data.

As a consequence, constant data are located in RAM during execution. However, equal literals are merged to save space.

6 Implemented features

cpik compiler is not a fully ANSI-compliant compiler because it lacks some ANSI features, but implemented features are as close as possible to this standard.

6.1 Preprocessor

cpik uses `cpp` (the GNU preprocessor), and is ANSI-compliant for preprocessing capabilities.

6.2 Data types

cpik only allows integer data types, which are either 8 or 16 bit long:

- `char` (8 bit)
- `unsigned char` (8 bit)
- `int` (8 bit)
- `unsigned int` (8 bit)
- `long` (16 bit)
- `unsigned long` (16 bit)

The `void` type is recognized in the traditionnal way, and any valid pointer can be declared.

Types are carefully checked, and mixed-type pointer expressions are rejected. `cast` operator allows type conversion, and type promotion is implemented, as specified by the standard.

Other PIC compilers consider `int`s as 16 bit integers. I prefer to consider them as 8 bit, because, as stated by the C language definition the `int` type *represents the natural integer for the target processor*. This definition guarantee to give optimal performance when `int` is used. PIC 18 devices are based on 8 bit data registers, so I suppose `int` should be 8 bit for this kind of processor.

For people who are not happy with 8 bit `int`s, a couple of `#define` directive or `typedef` instruction will do the job:

```
#define INT8   int
#define INT16  long
```

or

```
typedef int int8 ;
typedef long int16 ;
```

6.3 Data structuration

Traditionnal C tools for data structuration are arrays and `struct`s. **cpik** currently supports arrays in any valid way.

`struct` an `union` are also supported since version 0.4, and this support is efficient. Unlike several other well known compilers, `cpik` offers full support: `struct`s can be passed to functions as parameters and can be returned.

`struct`s are passed to function by value, as specified by the standard. Anonymous `struct`s are supported. `struct`s are perfectly compatible with `typedef`. `struct`s entities can be affected to other entities of same type.

The only restriction is due to device architecture: `struct`s cannot be larger than 128 bytes, but I suppose it is not a terrible limitation for a 8 bit microcontroller.

6.4 Storage classes

Variables can be either automatically or statically allocated.

Local variables and function parameters are truely `auto` entities: They are allocated on the data stack (which is distinct from return stack). It means that (unlike several pic C compilers) **cpik** can compile recursive algorithms[2], and can be used to produce re-entrant code.

6.5 Static initialization

All statically allocated variables or arrays can be statically initialized, as usual in C. Partial initialization of arrays is supported, and constant expressions in initializers are evaluated at compile time.

Initalization of pointers with address of static data or address of function is not supported, but will be in the future. Static initialization of `struct`s is not supported yet.

```
/* the following is supported */
int k = 4 ;           /* simple initalization */
char str[] = "hi !" ; /* array size deducted from initializer */
int array[2][3] = {{1,2,3},{4,5,6}} ; /* array of array initialization */
long z[10] = {0};     /* partial initalization, missing data replaced by 0s */
void f(char t)
```

[2]However, remember that the hardware stack is limited to 31 levels.

```
{
  static char t='Z'-26 ; /* compile-time constant folding */
  /* ... */
}
/* but the following is not */
int *pk = &k ;        /* arrggh !! constant adr initializer not supported */
void (*pf)(char) = f ; /* idem */
struct xxx a = { 1, 55  ; } ; /* not supported yet */
```

6.6 Scope control

The keyword `static` is implemented for data local to function, but not for data local to files. As a consequence, a global variable cannot be hidden to other compilation units. In other words, all global variable can be referenced via `extern` declarations.

```
static int x ; // not supported

void f()
{
  static char c ; // supported
  /* ... */
}
```

The keyword `extern` is implemented, so you can use `extern` to reference entities which are defined within another compilation unit, or manually located entities (see next section).

6.7 Address allocation

Entity addresses are determined during final assembly and depends on link process, so you cannot make any assumption about regular variable/function locations.

However, each entity can be manually located, using the `@`*address* extension. For example:

```
int x@0x12 ;  // manually located definition
extern unsigned char STATUS@0xFD8 ; // manually located declaration
```

This is very handy to access Special Function Registers. `cpik` provides a set of header files containing the declaration of each SFR register, for each processor. These headers are automatically generated by a program I wrote which takes the informations form Microchip ≪`.inc`≫ files. For example, here is the beginning of the `p18f242.h` header file:

```
#ifndef PROCESSOR
#define PROCESSOR
#define p18f242
//
// SFR and bits definitions for p18f242 device
// generated from p18f242.inc by inc2h utility
// Alain GIBAUD 2005, (alain.gibaud@free.fr)
//
// Register Files
//
unsigned int TOSU@0xfff ;
unsigned int TOSH@0xffe ;
unsigned int TOSL@0xffd ;
```

```
unsigned int STKPTR@0xffc ;
unsigned int PCLATU@0xffb ;
unsigned int PCLATH@0xffa ;
unsigned int PCL@0xff9 ;
...
```

Please note that **cpik** *does not perform any verification of specified addresses.* It is the programmer responsability to insure that there is really free memory at the specified location.

6.8 Instructions

All C instructions are implemented, excepted `switch`.

6.9 Operators

All C operators are implemented.

6.10 Extensions

6.10.1 Binary constants

Binary integer constants can be specified, using the following syntax:

```
int i = 0b1010 ; // synonym for 0x0A or 10
```

6.10.2 Digit separator

Decimal, octal, hexadecimal or binary integer constants can by made more readable with '_' character. This extension (inspired by ADA language) is useful for highlighting bit fields.

```
T2CON = 0b0_1110_0_11 ; // will be interpreted as  0b01110011
```

6.10.3 Assembler code

Assembler code can be included in C code using the `__asm__` directive. The syntax of this extension mimic `gcc __asm__` extension.

```
void f()
{
  __asm__("mylabel") ;
  __asm__("\tmovlw 0\n"
          "\tmovwf INDF0,0"
         ) ;
}
```

`__asm__` directive does not insert leading blank, so you can use it to insert labels. On the other hand, a trailing newline is automatically appended to asm code.

6.10.4 Interrupt service routines

Two empty interrupt service routines are provided by the run-time library (`/usr/share/cpik/<version>/rtl.slb`).

- `hi_pri_ISR` for high priority interrupts,
- `lo_pri_ISR` for low priority interrupts.

A user specific interrupt service routine can be written at C level by using the (non-standard) `__interrupt__` keyword:

```
__interrupt__ void hi_pri_ISR()
{
  /* interrupt service code */
}
```

Due to scanning order of libraries, this routine will replace the default one, because user libraries are scanned before `rtl.slb`.

The keyword `__interrupt__` insure that

- `W, BSR, FSR1, FSR2, STATUS`, registers are properly saved and restored on the data stack. `FSR0` is not saved because it is the stack pointer itself.
- `retfie 0` is used as return instruction instead of `return 0` [3] .

The body of an ISR routine can be written in pure assembly language, using the `__asm__` directive.

In this case, all previously mentionned registers can be freely altered, as long as `FSR0` (the software stack pointer) is not altered when the ISR exits.

In the case of C language interrupt code (or mix of C and asm code), registers used by the run-time library and user code must be saved and restored using the `SAVE_REGS` and `RESTORE_REGS` macros. Here is a typical example:

```
__interrupt__ void hi_pri_ISR()
{
   SAVE_REGS ;

   if (INTCON & (1 << TMR0IF))
   { // IT has occured on timer 0
      timer0_ISR() ; // any code can be used here
   }

   RESTORE_REGS ;
}
```

`SAVE_REGS` and `RESTORE_REGS` macro are defined in the `interrupt.h` header, which should be included.

It is also possible to take full control of context saving. In this case, just omit the `__interrupt__` directive and insert all needed code yourself with `__asm__` instructions.

6.10.5 Why and how to write interruptible code

The code generated by `cpik` is intrinsically interruptible. The run-time library, which is written in assembly code is also interruptible.

[3] The `reftie 1` instruction is not used because it is explicitely mentionned as bogus by errata documents from Microchip.

In order to implement a multi-priority interrupt system, low priority interrupt code must also be interruptible.

If you plan to use assembly code and interruptions together, you must enforce a simple rule : the software stack pointer (`FSR0`) must alway point to top of stack (ie: *to the last byte pushed onto the stack)*. Violating this rule will lead to stack corruption and data loss when an interruption occurs. Please read the following section, which is related to this point.

6.10.6 Disabling and enabling interrupts

In some very rare situations one can have to violate the interruptibility rule. In this case, interruptions must be masked. In order to keep the code consistant with interruptions usage you must use the following macro to manage interrupts (ie: *never* change the `INTCON` enabling bits (`GIE/GIEH/GIEL`) directly).

1. `MASK_HI_PRI_IT` : for disabling high priority interrupts.
2. `MASK_LO_PRI_IT` : for disabling low priority interrupts.
3. `UNMASK_HI_PRI_IT` : for enabling high priority interrupts.
4. `UNMASK_LO_PRI_IT` : for enabling low priority interrupts.

Before entering a critical (ie: non-interruptible) section, just use the `DISABLE_IT` macro : it will atomically disable all interrupts. When you leave the critical section, use the `ENABLE_IT` macro : interrupts will be restored to the previous state, no matter they was enabled or not. Of course, never change the interrupt status in a critical section.
All these macros are defined in `interrupt.h` header.

6.10.7 Pragmas

1. `#pragma processor device_name`

   This pragma has the same effect as the `-p` option in the link command. *device name*, must be a string like `p18f2550`. A program containing more than one `processor` pragma with different device names will have an unpredictable behaviour.

2. `#pragma _CONFIGxy value`

   This pragma allows to specify config bits.

   x must be a config register number ($1 \leq x \leq 7$) and y must be either `H` or `L`.

   Config bits are not processed by compiler, but directly passed to assembler, so you can use here constants not defined at C level, but defined in the `<processor>.inc` file. For this reason, you cannot use here the '`_`' character as field separator.

   A program containing more than one `_CONFIGxy` pragma with different values will have an unpredictable behaviour.

3. `#pragma _IDLOCx value`

   This pragma allows to specify ID data.

   x is the id location number ($0 \leq x \leq 7$).

   Values are directly passed to assembler, so you can use here constants not defined at C level.

   A program containing more than one `_IDLOCx` pragma with different values will have an unpredictable behaviour.

7 Hints and tips

Using `cpik` is not tricky, due to a simple and orthogonal design. However, some points may cause problems, due to special features of PIC 18 processors or peripherals.

7.1 Access to 16 bit SFR

16 bit SFRs (Special Function Register) are splitted in two 8 bit registers. You can access them as single 16 bit variables located at same address as low part of the data. For example, AD converter specific registers are defined as

```
unsigned int ADRESL@0xfc3 ;
unsigned int ADRESH@0xfc4 ;
```

In order to access the result of an AD conversion as single 16 bit value, just declare the following:

```
unsigned long  ADresult@0xfc3 ;
```

7.2 Access to 16 bit SFR - second part of the story

Viewing two contiguous 8 bits SFRs as a single 16 bit register in correct for almost all situations. However, it fails when accessing the `TIMER0` registers, because `TMR0L` and `TMR0H` SFR are continuously modified by hardware clock and need to be managed in a very special way (this is also true for `TMR1H/L`, `TMR2H/L`, etc.).

A write in `TMR0H` only store data in a temporary register. During write to `TMR0L`, this temporary register is transferred to real `TMR0H`, so `TMR0L` and `TMR0H` are simultaneously updated. This feature is a trick imagined by Microchip guys to allow atomic writes to 16 bit data on a 8 bit processor, but needs to write *hi byte first*, then *low byte*. Please see Microchip documentation for details.

Unfortunatly, code generated by `cpik` writes low byte first, so `TMR0H` is loaded with spurious data, and dataH is lost. The solution is simple, but need to split access in two parts:

```
TMR0H = value_to_load >> 8 ; // hi byte first
TMR0L = value_to_load & 0xFF ;
```

The same feature exists for reads, but doesn't cause problem because reads are performed in the correct order. However, this point might be changed if code generator or run-time library is changed, so I recomand to perform reads with care.

7.3 How to initialize EEPROM data

There is currently no specific `#pragma` to force EEPROM data value during chip programming. If you plan to initialize EEPROM, the following will do the job: just remember that EEPROM data are located at (conventionnal) address `0xF00000` in hex files, so you just have to force this address in assembly code. This can be done if you insert the following in a function/module you are certain to use in your program:

```
void your_function()
{
    /* Do the job this function is written for */
   return ;
   /*
   the following sequence is just a hack to insert
```

```
   data at eeprom addr in hex file
   It does not correspond to executable code
   (and cannot be reached by execution flow)
   */
   __asm__("ee___sav equ $") ;
   __asm__("\torg 0xF00000") ;
   __asm__("\tfill 0,1024") ; // 1K byte eeprom memory for 18F2525
   __asm__("\torg ee___sav") ;

}
```

Here, I initialize all EEPROM space of 18F2525 with zeros (default value for erased chip is all 0xFF).

7.4 Avoiding global namespace pollution increase modularity

`struct` usage is a good way to avoid global namespace pollution. This point allows to decrease the probability of global names clashes.

For example, one can group related data into a global `struct`, so only one name is visible at global level. For example:

```
/* global data */
int a,b ;
long c ;
char t[10] ;
```

could be replaced by an (anomymous or not) `struct`

```
struct
{
  int a,b ;
  long c ;
  char t[10] ;
} mycontext ;
```

Data should be adressed by expressions such as `mycontext.c = 23 ;` wich is verbose but *has exactly the same cost* as `c = 23 ;`

7.5 Do not use uppercase only symbols

Device-specific headers (ie: `p18fxxx.h` files) contain variable declarations and constant definitions. This may lead to clash with your own declarations. For example

```
struct Z { int a,b ; } ;
```

will leads to unexpected and hard to understand error message. The reasons is simple: `Z` is defined by a macro as a numeric constant so your code will be expanded by the preprocessor to something like:

```
struct 2 { int a,b ; } ;
```

which is hard to understand for the compiler. Hopefully, this situation is simple to avoid because the headers define uppercase only symbols. Do not use this kind of symbols yourself.

7.6 How to write efficent code

The following simple rules helps the compiler to produce more efficient code.

1. Use `unsigned` variant of integers, whenever possible
2. Use `int` (or `char`) instead of `long`, whenever possible
3. Prefer `++` to `+` or `+=`
4. Avoid to compute twice the same sub-expression
5. Array access are not very efficient, so prefer to access arrays content thru pointers.
6. Do not hesitate to use structs, they are generally very efficient.
7. Do not hesitate to use constant array indexes, they are generally very efficient.
8. Global data usage leads to smaller and faster code. Be careful, it also lead to lack of modularity and memory wasting. See previous section for modularity issue.
9. Implement left shifts with x2 products: due to availability of hardware multiplication the code will be fast. This rule does not apply to right shifts.

8 Libraries

For now, a small number of libraries are available, and they have been developed for my own needs, and/or compiler testing so they are not always versatile.

Excepted for `rtl.slb` and `lcd.slb` (which are directly written in assembly language) each library `x` is written in C, and related to 3 files:

- `x.c` : the source code (when written in C),
- `x.h` : the header file,
- `x.slb` : the source library (or object) file.

Obviously, sources files are not needed to use libraries, but can be useful because the `cpik` project is under development, so libraries are far from being stabilized.

Sources libraries are installed in `/usr/share/cpik/<version>/lib` and headers in `/usr/share/cpik/<version>/include`.

8.1 standard IO library

Basic support for standard-like IOs. This library can be used to perform IO on character oriented devices. Attachment to one device is based on redirection of an input and output function.

Redirections can be changed at any time to use several devices simultaneously. However, remember that input buffer is unique and should be flushed when input device is changed.

8.1.1 IO redirection

1. `output_hook set_putchar_vector(output_hook)`

   Sets indirection vector for character output. This function gets and returns a pointer to a function receiving `char` and returning `void`. This feature allows redirection of outputs to virtually any device, and to save the previous output vector. Output vector is not initialized, so this function must be used prior any output.
   `output_hook` is defined by `typedef void (*output_hook)(char) ;`

2. `input_hook set_getchar_vector(input_hook)`

   Sets indirection vector for character intput. This function gets and returns a pointer to a function returning `char` and receiving `void`. This feature allows redirection of inputs from virtually any device, and to save the previous intput vector. Intput vector is not initialized, so this function must be used prior any input.
   `input_hook` is defined by `typedef char (*input_hook)() ;`

8.1.2 output functions

1. `void putchar(char c)`

   Writes character `c`.

2. `int puts(char *s)`

   Writes string `s`. Always returns 0.

3. `void outhex(unsigned long n, char up)`

   Writes unsigned long `n` in hexadecimal. If `up` is `'A'` uppercase letters are used for `A B C D E F` digits, else `up` must be equals to `'a'`, and lowercases are used. Leading 0 are suppressed, so this function can be also used for 8 bit numbers.

4. `void outdecu(unsigned long n)`

   Writes unsigned long `n` in decimal. Leading 0 are suppressed, so this function can be also used for 8 bit numbers, which are promoted to `unsigned long` before call.

5. `void outdec(long n)`

   Writes long `n` in decimal. Leading 0 are suppressed, so this function can be also used for 8 bit numbers.

6. `void printf(char *fmt)`

   Mini implementation of the standard `printf()` function. Recognized format specifications are :
   `%c %s %d %u %x %ld %lu %lx`

   Despite the fact `cpik` does nor recognize the ANSI ≪`...`≫ syntax, this function supports a variable lengh argument list, and its prototype should be ≪`void printf(char *fmt,...);`≫

   Due to memory size available, width specifiers (ie: `%03d` for example) are not supported.

8.1.3 input

1. `long getch()`

   Returns the next available character, or EOF if no character is available. This character is not echoed thru output vector. This function does not use input buffer.

2. `long getche()`

   Returns the next available character, or EOF if no character is available. This character is echoed thru output vector. This function does not use input buffer.

3. `long getchar()`

   Returns the next available character, or EOF if no character is available. This character is echoed thru output vector. This functiopn does use input buffer. (See `ungetchar()`). The input buffer is 80 bytes long, but this can be easily changed at source code level.

   All ≪high level≫ functions like `scanf()` or `gets()` use `getchar()` as low level input function.

   Please note that (like `getch()` or `getche()`) this function returns a `long`, so all values ranging from `0` to `255` can be returned.

4. `unsigned int fillbuf(char p[],unsigned int nmax, int *eof_flag)`

   This function is used to fill input buffer when it is empty. `p` points to this buffer, which can contains up to `nmax` characters. Read is stopped when buffer is full (`nmax` characters) or when `'\n'` character is encountered.

   `fillbuf` returns the number of character stored in the input buffer, including the `'\n'` terminator.

   As a side effect, `eof_flag` is set to 1 if end of file condition is reached, and 0 if not.

   `fillbuf` interprets Backspace character, so it provides a primitive but useful line editing capability.

5. `char *gets(char *t)`

   Read input characters until `'\n'` is encountered, and store them into buffer pointed to by `p`. Terminating `'\n'` is not stored into buffer. Always return `t`.

6. `int scanf(char *fmt, addr [,addr ..])`

   Mini implementation of `scanf()` function. Recognized format specifications are :
   `%c %s %d %u %x %ld %lu %lx`

8.2 rs232

Minumum support for pic rs232 interface.

1. `void rs232_init()`

   Configures rs232 interface in polling mode. Provides 9600 bauds communications at 16Mhz. Source code must be modified for other speeds

2. `void rs232_putchar(char c)`

   Sends character `c` to rs232 interface when this one becomes available.

3. `char rs232_getchar()`

   Waits for a character, and returns it when available. Can indefinitly block.

8.3 lcd

Support for classic HD-44780 based LCD display, in 4 bit mode. This display must be connected to port A. See source code for details about connections..

The following are low level functions, but LCD display can also be used from hi-level functions (such as outdec or outdecu), if the proper output redirection is programmed.

1. `void lcd_init(int delay) ;`

   Initializes display. The `delay` parameter is used by internal temporisation loops. Delay depends on LCD display capabilities and device clock. The following values work for me.

CPU frequency	delay
4Mhz	8
8Mhz	15
16Mhz	30
32Mhz	60
40Mhz	75

2. `void lcd_putchar(char c);`

   Displays character `c` at current cursor position

3. `void lcd_move(int pos) ;`

   Moves cursor to `pos` position. Coordinate system depends on LCD type.

4. `void lcd_clear() ;`

   Erases display.

5. `void lcd_hex4(unsigned int c) ;`

   Displays low nibble of `c`, as an hexadecimal digit.

6. void `lcd_define_char(char c,char bitmap[8]) ;`

   Defines a new character with code `c`.

   Definable characters codes range from 0 to 7, and the character matrix is 5x8 pixels large. `bitmap` array is an image of the character, each array element corresponds to one line of the matrix.

7. `void lcd_hex8(unsigned int c) ;`

   Displays `c` as two hex digits.

8. `void lcd_hex16(unsigned long n) ;`

   Displays `n` as four hex digits.

9. `void lcd_puts(char *s) ;`

   Displays a nul-terminated character string.

10. `void lcd_putcmd(char cmd) ;`

    Enters command mode, then send command `cmd` to LCD display.

8.4 AD conversion

This library is really minimal : I wrote it for my own needs, so source code must be edited to adapt it to yours.

1. `void AD_init()`

   Initialize AD conversion system for 16MHz processor. AN0,AN1,AN2 and AN4 are used as analog inputs. AN3 is used as voltage reference input.

2. `unsigned long AD_get(unsigned int ch)`

   Starts AD acquisition and conversion on channel `ch`. Channel number can be `0` (`AN0`), `1` (`AN1`), `2` (`AN2`) or `3` (`AN4`).

8.5 EEPROM read/write

This library allows to perform EEPROM read/write in polling mode. It also contains code to statically initialize EEPROM data (see section hints and tips for details). Please comment out or modify this code (see `ee_init()` routine to fit to your own needs).

1. `void ee_init()`

   Initializes EEPROM subsystem in polling mode.This routine also contains code to statically initialize EEPROM data (see section hints and tips for details). This code may have to be edited to fit your needs.

2. `unsigned int ee_read8(unsigned long addr)`

   Returns the 8 bit data located at address `addr`.

3. `unsigned long ee_read16(unsigned long addr)`

   Returns the 16 bit data located at address `addr`.

4. `void ee_write(unsigned long  addr, unsigned int  value)`

   Writes 8 bit value at address `addr`.

5. `void ee_write16(unsigned long  addr, unsigned long  value)`

   Writes 16 bit value at address `addr`.

6. `unsigned int ee_inc8(unsigned long addr)`

   Increments 8 bit value located at address `addr`. Return the incremented value.

7. `void ee_inc16(unsigned long addr)`

   Increment 16 bit value located at address `addr`.

8. `void ee_refresh()`

   Performs EEPROM refresh as recomended by Microchip data sheet, for very long time data retaining. This routine is not really tested, but I used it, and data have not been destroyed.

8.6 Timer 0

Basic implementation of a slow real-time clock, with $1s$ and $1/10s$ ticks. This module can provide up to 8 independant $1s$ 16 bit clocks. It also provides one 32 bit $1s$ clock. Moreover, a flag is toggled each $1/10s$, and provides a faster clock.

1. `void timer0_init()`

   This function initialize timer0 sub-system (mainly prescaler register). It calls `reload_timer0()`, then starts timer0 activity.

2. `void reload_timer0()`

   Reloads timer0 for $1/10s$ delay.

3. `void timer0_ISR()`

   Interrupt Service Routine for timer0 interrupts. You must install an interrupt handler which calls this ISR. The following code will do the job.

```
__interrupt__ void hi_pri_ISR()
{
   SAVE_REGS ;
   if (INTCON & (1 << TMR0IF)) // does interrupt comes from timer0 ?
   {
      timer0_ISR() ; // yes, call interrupt handling code
   }
   RESTORE_REGS ;
}
```

4. `void start_clock( unsigned clocknum)`

   Sets clock count of clock number `clocknum` to 0, then start it.

5. `void stop_clock(unsigned int clocknum)`

   Stops clock `clocknum`. Stopped clocks can be restarted.

6. `void restart_clock(unsigned int clocknum)`

   Restarts a stopped clock.

7. `unsigned long get_clock(unsigned int clocknum)`

   Gets number of seconds elapsed since clock `clocknum` has been started or restarted.

8. `unsigned long get_clockm(unsigned int clocknum)`

   Gets number of minutes elapsed since clock `clocknum` has been started or restarted.

9. `void clear_clock(unsigned int clocknum)`

   Explicitely sets clock `clocknum` to zero.

10. `unsigned long *get_globalclock()`

    Returns addr of first element of an array of two `unsigned long` containing global clock. First element of this array contains low part of global clock. Global clock is statically initialized and started when `timer0_init()` is called. There is no way to stop it.

11. `insigned int timer0_flags()`

    Returns current state of clocks flags. One bit of the value returned by this function is toggled each second. Another bit is toggled each 1/10 second.

    The `T0_1S_FLAG` and `T0_0_1S_FLAG` constants must be used to get the flag you need.

    Here is an example of code executing a task each second.

```
unsigned int old_flag = timer0_flags() & T0_1S_FLAG, new_flag ;

for( new_flag = old_flag ; ; )
{
  if( (new_flag = timer0_flags() & T0_1S_FLAG) != old_flag )
  {
    old_flag = new_flag ;
    // do something each second
  }
}
```

9 Source library structure

A source library is an assembly language source file, with special comments interpreted by **cpik** linker. Each special comment begins with `";<"`, located at first column, and ends with `">"`. Any information inserted after the final `">"` are really comments and will be ignored by the linker.

Source libraries are structured in *modules*, each module can contains either data or code.

Here is the list of recognized special comments:

1. **Begin of module definition** : the specified module follows the comment.

   ```
   ;<+module_name>
   ```

2. **End of module definition**

   ```
   ;<->
   ```

3. **Module reference** : the specified module is needed by the current module.

   ```
   ;<?module_name>
   ```

4. **Static initializer** : the specified data must be used by the linker to initialize the current module (this module corresponds to an array or structure). A module can contain several static initializers.

```
    ;<= byte1 byte2 ... >
```

Example:

```
int table[3] = { 1, 2 } ;

unsigned char x2(unsigned char c)
{
  return c * 2 ;
}
```

will generate:

```
;<+C18_table>
  CBLOCK
  C18_table:3
  ENDC
;<= 1 2 0 >
;<->

;<+C18_x2> unsigned char x2(unsigned char c@0)
C18_x2
;         return c * 2 ;
  movff INDF0,PREINC0
  movlw 2
  ICALL mul8u
  movff POSTDEC0,R0
; }
L18_main_x2_0
  return 0
;<?mul8u>
;<->
```

10 Needed software

The GNU `cpp` preprocessor must be installed in your system. As `cpp` is de facto installed with all Linux distributions, this is not a strong requirement.

11 Tutorials

11.1 Hardware

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 pins 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.

11.2 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.

11.2.1 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

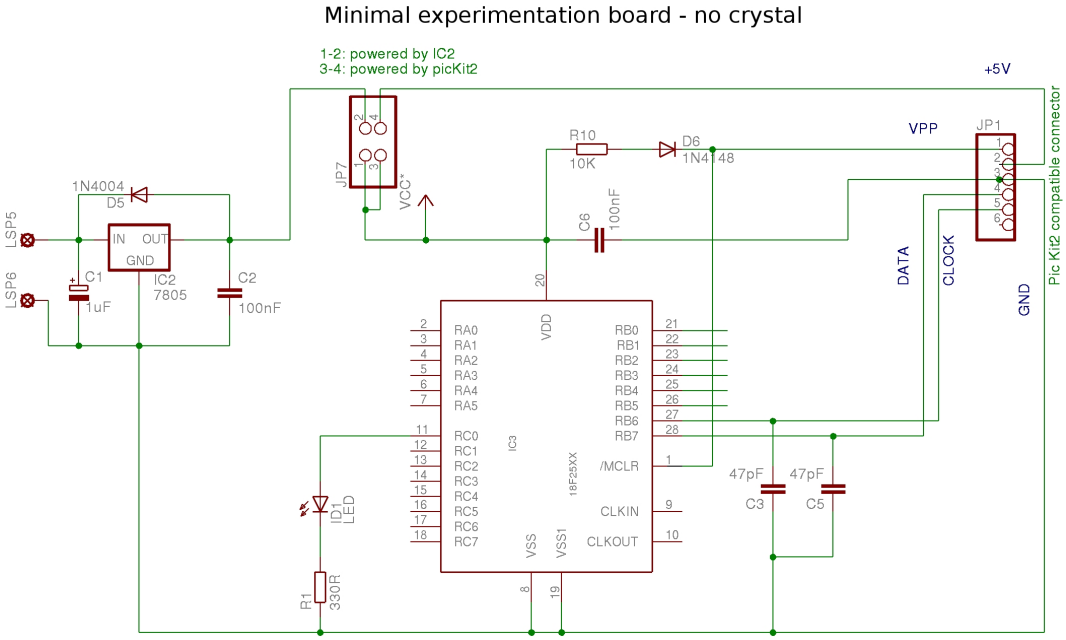

Figure 1: Tutorial #1 - base schematic

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>
```

11.2.2 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. In PIC182525 devices, CONFIG1L, CONFIG3L and CONFIG4H do not exist so you should not initialize them.

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').

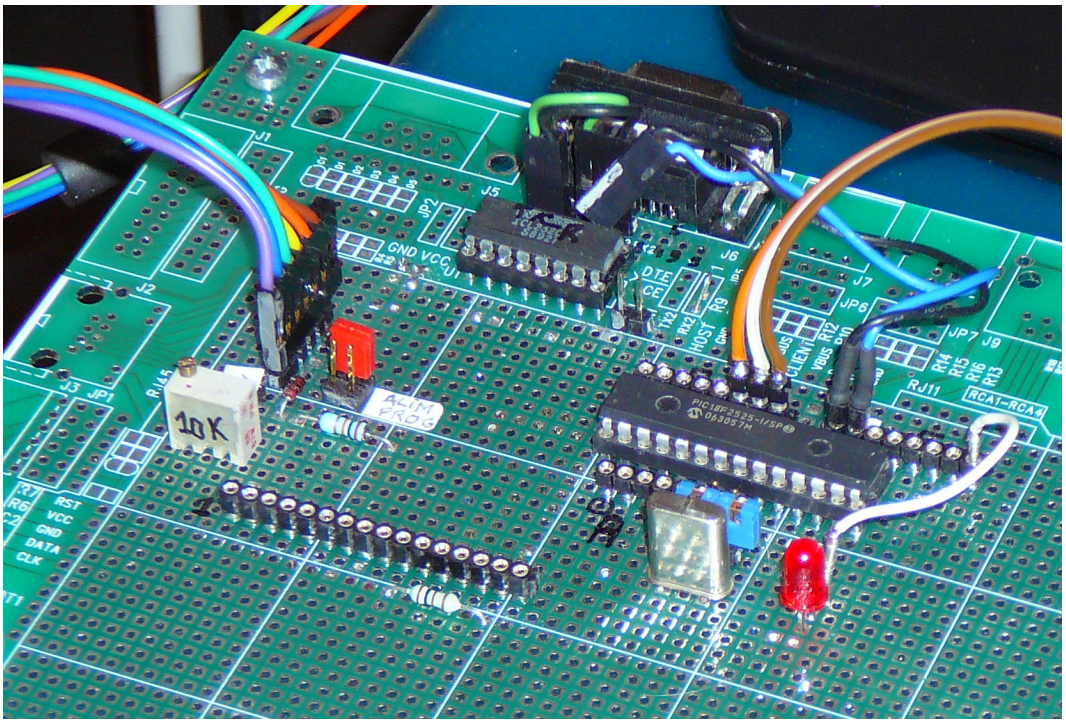

Figure 2: Tutorial #1 - experimentation board

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
// 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
```

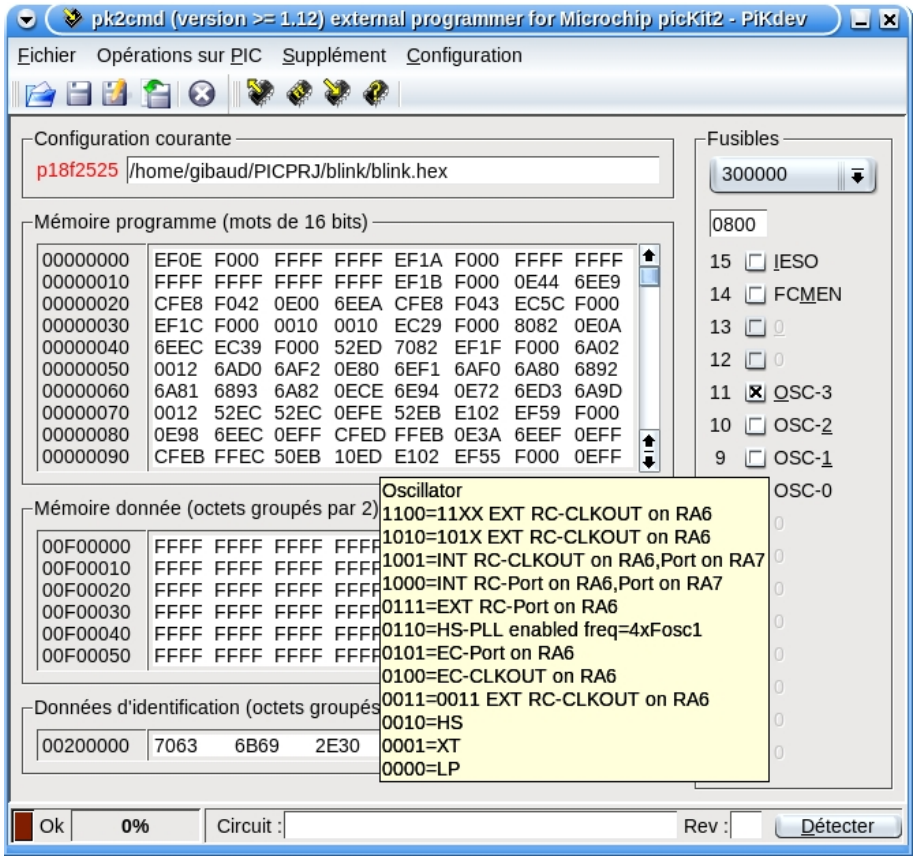

Figure 3: Using pikdev as an assistant for config bits settings (french interface)

```
#pragma  _CONFIG6L B'00001111'
// no protection
#pragma  _CONFIG7H B'01000000'
// no protection
#pragma  _CONFIG7L B'00001111'
```

11.2.3 Microcontroller initialization

Here begins the real job. We have to set various device register 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 `<p18f2525.h>`, 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
}
```

11.2.4 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)  ;
}
```

11.2.5 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[4] explicitely specify than `main()` should return an `int`.

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

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

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

11.2.6 Build the application

When used from `pikdev`, `cpik` usage is staighforward: 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`).

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. assemble the file `blink1.asm` and generates the hex file `blink1.hex`.
   `gpasm -L -o blink1.hex -w 2 blink1.asm`
   The code is around 54 bytes long, including startup stuff.

11.3 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.

11.3.1 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:

- key 0 : `d` is incremented, (blinking slows down).

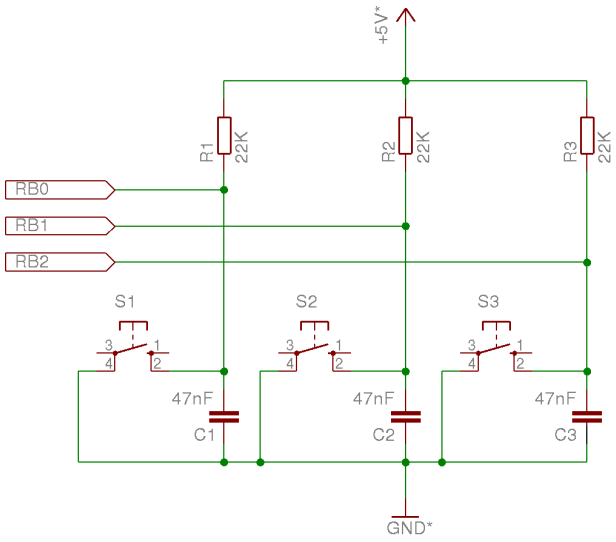

Figure 4: Mini keyboard for tutorial #2

- 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 );
   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.

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.

11.3.2 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 somes input pins of port B changes. 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 interrupts 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 this 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 ;
}
```

With 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, derpending 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.

12 How to contribute to the cpik project ?

cpik needs contributors !. Writing compiler, libraries, tutorials, pikdev support, WEB site, etc. is an exciting but huge work for only one man.

I think that this project is really viable. cpik code is not perfect but has many interesting qualities, compared to other free compilers. So far, everything I write with it works (I cross my fingers), so it should work for other people too.

If you are interseted by bringing a new compiler to free software, you can contribute in many manners :

12.1 Feedbacks and suggestions

When cpik works for you or doesn't, please send an email. Explain what you do with it, and why it fit (or doesn't) your needs.

12.2 Bug reports

If you detect a bug, please send me the most simple source code which provoke this bug. I will be able to analyse the generated code and fix the problem.

12.3 Documentation

Feel free to send fix or extension for the documentation. To native english speakers: help me to fix incorrect english sentences.

12.4 Libraries

`cpik` needs libraries. All sorts of them. Some libraries are easy to write (`stdlib/string`) but I have no time to do it. Some are really hard to code (IEEE754 compliant floating point).

Basically, each device peripheral (`timer`, `AN conversion`, `USB` etc.) needs a library.

12.5 Code contribution

Contribuing to `cpik` code is not a trivial task, because it needs to understand how the compiler works. More generally, a knowledge of compilation techniques is also needed. You should also understand the philosophy of C language, and of course be a PIC18 assembly language programmer. Well, don't be discouraged: there is a lot of places where you just need to know the C++ language to improve `cpik`.