cpik

C compiler for PIC©-18 devices

Version 0.7.0

Alain Gibaud

alain.gibaud@free.fr

(Documentation: rev A)

May 18, 2012

Contents

1 Introduction

cpik is an ANSI-C compiler for Microchip PIC 18 microcontrollers. It is a personnal project developped on my spare time. This is why this project took a long time to reach the current state.

2 What is new in version 0.7.0 ?

1. **Full support for bit-fields**
   Bit fields are now supported in the standard way.

2. **New headers files**
   A new header file is now provided for each pic18 device. These headers are compatible with Microchip's headers, but are not a copy of them.

3. **New `inc2h-v2` utility**
   This is a new version of the previous `inc2h` that is able to build the headers files from Microchip's ≪`.inc`≫ files.

4. **Explicit bit-fields**
   This feature is an extension of the standard that allows to slice any 8 bit variable in bit fields.

5. **Support for the `const` keyword**
   `const` is now supported as specified by the ANSI standard. Because the compiler is now more strict about constness, old codes may have to be slighly modified.

6. **Minor bug fixes**

3 What is new in version 0.6.0 ?

1. **Full support for 32 bit floating point arithmetic.**
   This implementation is compliant with the IEEE-754 standard on floating point representation, but the NAN and INF are not managed. This design choice has been made to reduce the size of code. The core floating-point library[1] has been carfully written in assembly language, so this implementation is likely to be fast. The FP library is provided as a separate library (`float.slb`).

2. **Support for IO on floating point data**
   The `printf` and `scanf` have been updated for this purpose and new format spécifications (`%e` and `%f` with user-selectable precision) have been introduced . Like for 32 bit integers, this support must be enabled to be active, so people who do not use FP arithmetic will not be penalized. New IO functions for floating point are also provided.

3. **Math library**
   This implementation of the math library is written in C and provides 22 usual functions for floating point calculation.

4. **Standard library**
   This is a first implementation of `stdlib` that contains 10 usual functions.

5. **Functions with variable argument-list**
   This a fully compliant implementation of the ANSI standard about functions with variable argument lists, using the ≪`...`≫ syntax. The standard header (`stdarg.h`) provides the necessary `va_xxx` macros.

[1] basic operators and conversion routines.

6. `errno` **support**
   The unix low-level mechanism for reporting errors during math or IO operation is supported, and the standard `errno.h` header is provided.

7. **New command line option**
   The traditionnal ≪`-D`≫ switch allows to pass macro definitions to the preprocessor.

8. **Minor bug fixes**

4 The ≪philosophy≫ behind `cpik`

My idea was to develop a compiler as simple as possible but conformant to the ANSI specifications. This is a huge work for a one 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 initially chose to suppress the support for bit fields 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 is always possible to replace this statement with cascaded `if`(s). The resulting code is generally less efficient, but works. Finally, this statement is supported since V0.5.3.

The first version of `cpik` (V0.2) did not recognize the `typedef` instruction, and had no support for `struct`s or `union`s. `typdef` has been implemented in V0.3, and structures/unions in V0.4. 32 bit integer arithmetic is supported since V0.5.

Floating point support exists since version V0.6.0, and comes with a very decent math library including trigonometric and logarithmic functions.

Support for bit fields exists since V0.7.0., so support for the ANSI-C standard is now almost complete. `cpik` 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 is written in C++. Any feedbacks concerning bugs, feature requests or criticisms can be addressed to Alain Gibaud (`alain.gibaud@free.fr`).

5 A very 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 itself, 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, that 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 send feedbacks about the code.

This unusual approach presents for me several advantages:

- Any source library is a simple text file, so it can be manually written in assembly language, using a standard text editor (this point is important to bootstrap a totally new development environment). For example, the LCD library has been developped from scratch with a text editor as unique tool, and used to support the very first program[2] compiled with cpik ever executed (see figure 1).

[2]Believe it or not, this program (a simple `for` loop) worked successfuly at the first execution. To be honest, this execution has been preceeded by numerous hours of manual check of the generated code, and many modifications of the compiler.

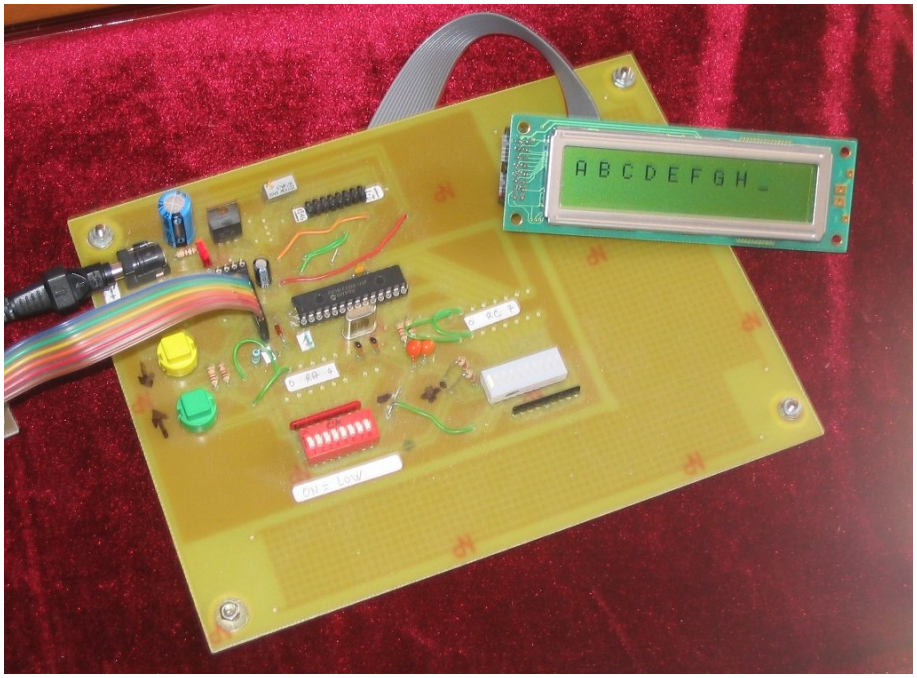

Figure 1: Result of the very first program compiled by `cpik` ever executed

- source libraries do not depend on any object/library format, and/or obscure, potentially 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` executable.
- This design has proven its flexibility for the implementation of support for data located in ROM, or jumps optimisations
- Symbolic calculations depending on the location of entities in memory can be deferred to assembly stage.

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.

6 Installation of `cpik`

Note:

- In the following, `<version>` is a 3-digit string corresponding to the version number of cpik (eg: `060` for V0.6.0).
- The `cpik` archive is supposed to be extracted.

6.1 Manual build

Although is is not my preferred option, it is possible to build cpik manually:

1. Compile each `.cpp` file separatley, using the ≪`g++ -Wall -O2 <filename>.cpp`≫ command
2. Link with ≪`g++ -o cpik<version> <all the .o files>`≫
3. ≪`su root`≫
4. Copy the `cpik<version>` executable to `/usr/bin/`
5. Copy the directory `0.6.0` (use here the proper version number) to `/usr/share/cpik`

6.2 Build using qmake

`qmake` is the build tool that comes with `Qt`. You can download the complete Qt toolkit at the address `http://qt.nokia.com/`, but it is certainly available as a package with your Linux distribution.

1. `qmake -o Makefile cpik<version>.pro`
2. `make`
3. `su root`
4. `make install`

You can install cpik in a non-standard directory using the command

```
qmake PREFIX=<abs-dir> -o Makefile cpik<version>.pro
```

where `<abs-dir>` is an *absolute* path to the directory where you plan to install `cpik`. Be certain that `cpik` *will not work if you use a relative path.*

This option is provided for very special situations, and my advice is to avoid it.

6.3 `cpik` under Windows

`cpik` can be built under Windows, using the `qmake` tool and the gcc for windows toolchain. No installation procedure is provided but you can use the following instructions to install to the `C:\cpik` directory:

1. `qmake -o Makefile cpik<version>.pro`
2. `make`
3. Copy the directory `0.6.0` to `C:\cpik`
4. Create a `C:\cpik\bin` directory
5. Copy the `cpik<version>` executable to `C:\cpik\bin`

Do not copy the executable to any other place, because `cpik` uses its own location to find the files it needs.

7 Command syntax

The `cpik` command can be used for both compilation or linking tasks, exactly like the `gcc` frontend. However, `cpik` is not a frontend and really performs these two tasks. Since V0.5.3, cpik can also directly generate the final `.hex` file, and is able to optimize jumps after the linking process.

7.1 Compilation

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

`-v` : prints version number, then exits immediatly.

`-o output_file` : Specifies 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.

`-Dmacro[=value]` : specifies a macro definition that is passed to the `cpp` preprocessor. Notice that there is no white space after «`-D`»

`-I path` : specifies 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 the `-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. Notice that «`-I A,B,C`» is an allowed shortcut for «`-I A -I B -I C`».

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

`-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, without type annotation
`-d16`	print internal expression trees before optimisations, with type annotations
`-d32`	print internal expression trees after optimisations, without type annotation
`-d64`	print internal expression trees after optimisations, with type annotations

The usage of the `-d` option is never useful for normal operations with `cpik`. Produced outputs are hard to interpret for non developers.

`input_file` : specifies the source file name, with `.c` extension.

This command *cannot be used to compile more than to one source file* in a single invocation.

7.2 Link

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

`-v` : prints version number, then exit immediatly.

`-o output_file` : specifies the output file name. By default, this name is `a.asm`. This file can be immediatly processed by the assembler and does not require any additionnal support.

`-L path` : specifies 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/` that is searched in last position. Note that ≪`-L path1,path2`≫ is a shortcut for ≪`-L path1 -L path2`≫

`-p device` : specifies the target pic device name. `device` must be a valid pic 18 name like `p18xxxx`. An invalid device will cause the final assembly to fail. By default, the selected device is `p18f1220`.

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

7.3 Final assembly and jump optimizer

```
cpik -a [-d<value>] [-o output_hex_file] -p device
                                    [-A gpasm_executable_path] input_asm_file
```

The `gpasm` assembler can be invoked directly from `cpik`. This stage builds the final `.hex` file, from the `.asm` file generated by the linker. During this step, long jumps are replaced by short jumps whenever possible. Therefore, the resulting code is shorter and faster than the code directly generated by `gpasm`.

`-A gpasm_executable` : specify the absolute path to the `gpasm` tool. This option is generally not needed but, when used, the specified name *must contains* the name of the executable itself (eg: `/a/b/gpasm` instead of `/a/b/`)

`-p device` : specifies the target pic device name. `device` must be a valid pic 18 name like `p18xxxx`. An invalid device will cause the final assembly to fail. By default, the selected device is `p18f1220`. This specification is not optional because it allows `cpik` to check the program against an eventual memory overflow.

`-o output_hex_file` : specify the `.hex` file name. The default name is `<input_asm_file>.hex`.

`-d<value>` : ask optimizer to print debug informations when `value=2` or statistics on how many words are saved when `value=1`.

8 Information about the implementation of C language

cpik generates code for PIC-18 processors running in legacy (ie: non-ehanced) mode. The PIC-18 core 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. Programs can be larger than 64K bytes (when the device has enough memory), but pointers to functions can only reach the lower 64 KB of memory. This is not an issue because it is easy to force the adresses of target functions to be less than `0xFFFF`.

`cpik` has been designed to produce a 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*. There is no banks, "small" stacks, "far" pointers or other tricky ways to save memory but to confuse everybody.

8.1 Stacks

The code generated by `cpik` uses two stacks:

- *hardware return stack* (31 levels):

  This stack is part of the PIC-18 architecture. It is only used to save the return addresses before subroutines calls. 31 levels of nested calls are generally largely sufficient for most applications.

  However, recursive routines may provoke overflows of the return stack, this point being under the responsability of the programmer.

- *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 manipulation is very efficient. `FSR0` is used as the software stack pointer.

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

8.2 Memory layout

The current memory layout used by cpik is the following[3]:

Name	Addresses	Size	Usage
Soft Stack	`[GG+1->TT]`		software stack, grows upward to top of memory
Globals	`[PP+1->GG]`		global variables
Scratch	`[22->PP]`	`SCRATCH_SIZE-20`	returned value area
FP aux	`[14-21]`	8	auxiliary zone for FP routines
Registers	`[2->13]`	12	R0,R1,R2,R3,R4,R5 pseudo-registers
`C18_errno`	`[1->1]`	1	reserved by libraries
IT mask	`[0->0]`	1	reserved by RTL

[3]This layout has changed from V0.5.3 to V0.6.0

Addresses from 0 to 21 (22 bytes) are reserved for the run-time library. The *Register* zone (20 bytes) is used for both integer and floating point calculation, and is also used to store the values returned by functions. Because a function can return more then 20 bytes, the *Register* zone can be extended by the *Scratch* zone. The size of this area can be ajusted by editing the prolog file `/usr/share/cpik/<version>/lib/cpik.prolog`
The default size specified in this file is sufficient to handle functions returning 40 byte long structures. If you are not happy with this size, just change the `SCRATCH_SIZE` definition to the value you want. However, remember than `SCRATCH_SIZE` cannot be less than 20 bytes.

The *Globals* zone is used to store the static data (ie: global or static variables).

Finally, the *Soft stack* begins at the end of the *Globals* zone and uses the remaining of the memory.

There is currently no reserved zone to implement a heap for dynamic memory allocation (`malloc(), free()`). However such a zone could be obviously implemented at the end of physical memory, and must expand from top (high addresses) to bottom.

8.3 Register usage

`cpik` uses 6 16 bit pseudo registers named `R0`,`R1`,`R2`,`R3`,`R4` and `R5`. 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 `R5` are used by the Run-time library (RTL),
- `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.

8.4 Computation model

- *operators with 2 operands* 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.
- *operators with 1 operand* take their operand from top of stack and replace the result at the same place.

In fact, the code generated by `cpik` might differ from this scheme, depending on various optimizations performed by the compiler.

Due to hardware limitation, the total amount of local non static data declared in a function can't exceed 127 bytes. Data local to a function are *formal 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.

In the following example, 2 bytes are used for parameters `u` and `v`, and a third one is used for storing a temporary.

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

Here is the result of the compilation:

```
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 the stacked copy of u by u+v
  movlw 3
  ICALL div8          ; divide top of stack data by 3
  movff POSTDEC0,R0   ; pop result to R0L
  return 0
```

Notice that the space used to store the local variables is not necessarily the sum of space needed for each variable. For example, in the following code, `j` and `z` are stored at the same address, so only 2 bytes are used on the stack to store `k`, `j` and `z`.

```
int func2(int k)
{
   if( k > 27)
   {
      int  j = 3 ;
      k += j ;
   }
   else
   {
      int  z = 23  ;
      k += z ;
   }
   return k ;
}
```

8.5 Function calling conventions

All parameters are passed to functions on the software stack. They are stacked in reverse order (1st parameter pushed last)[4]. Moreover, the stack cleaning is performed by caller : these characteristics are common for C code because they are useful to implement functions with variable number of parameters, such as `printf`[5].

8 bit results are returned in `R0L` register, 16 bit results are returned in `R0` register and 32 bit values are returned in the `R0-R1` pair.

Structures are returned in a block of memory that begins at address `R0`, with the same size than the returned structure. Enough space is reserved by default for structure up to 40 bytes. This pool can ajusted to fit you needs or the hardware requirements. See section 8.2 for details.

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

```
void caller()
{
```

[4]No aligment is done during parameter passing, so any data can be located at odd or even address.
[5]This feature will change in future versions.

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

and the resulting code

```
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
```

8.6 Optimizations

`cpik` performs many optimizations, but not all possible optimizations. Optimizations can be performed during code analysis, intermediate code generation, asm code generation and suprisingly *after* the code generation.

1. *NOP removal*

   Most expressions that have 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 extend it to `FSR1` register.

3. *Constant folding*

   Most constant subexpressions are computed by the compiler, so complex expressions are often compiled as a single constant (eg: `x= (1+2*4) << 1 ;`). However, a lot of constant foldings are done by the peephole optimizer or the expression simplifier (eg: `st.a.b.c = 34 ;` is exactly compiled like `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, the 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 that depends on the 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.

6. *Dead code removal*

   `cpik` takes into account constants when testing boolean conditions. For example, instuctions like

   ```
   if(0) { ... }
   ```

   or

   ```
   while(0) { ... }
   ```

   do not generate any code. In the same way,

   ```
   while(1) { .. }
   ```

   generates a genuine infinite loop with no test, exactly like `for(;;) { .. }` does.

   However `cpik` does not perform a global analysis of code, so common subexpression removal are out of scope at this time.

7. *Jumps optimization*

   `cpik` contains a special optional[6] optimizer that allows to use short jumps instead of long ones, whenever possible. This step is executed after the asm source code generation and can reduce the memory size by 20%.

8.7 Data in ROM

The PIC-18 processors are based on two separate program and data spaces. Data space is RAM, and program space is ROM. This architecture causes problem to store initialized data (such as literals like `"hello !"`). Indeed, the only way to store initialized data is to place them in program space.

In order to keep the compiler simple, `cpik` adds a loader routine to startup code. This routine is automatically activated before the `main()` function and copies all initialized data from program space to data space[7].

As a consequence, initialized data is located in RAM during execution. This feature is necessary when the data can be modified, but is not desirable when the data is read-only because it wastes RAM space.

Since version 0.5.3 `cpik` offers several simple ways to use data located in program space. In the following sections, program space is simply called ROM. The ROM support presented here is fully

[6] See section 7.3

[7] The loader is not included if your program does not use statically initialized data.

implemented with macros and a couple of run-time routines. These macros are defined in the `rom.h` header.

The support presented below is experimental because it prepares a definitive implementation based on a `__rom` keyword. However, it is perfectly usable and very efficent from both time and memory points of view.

8.7.1 Creating a block of data in ROM

The following macros allow to insert data in program space:

1. `ROM_TXT(text)`

   Insert the specifed text in ROM, at the current program location. The text must be enclosed by double quotes. The resulting block *is not nul-terminated by default*, so an explicit zero must be included in the specified string when needed. (eg: `ROM_TXT("my string\0") ;`). It is important to note that the text is encoded with 2 chars in each program word, so texts with an odd number of chars are padded with a nul char (ASCII `0`).

2. `ROM_BYTES(data)`

   Insert the specified sequence of bytes in ROM. The `data` parameter can be any sequence of bytes, encosed by double quotes (eg: `ROM_BYTES("0,1,2,0x33,5") ;`). Since the parameter of `ROM_BYTES` is processed by the assembler, your can use any syntax recognized by the assembler, for example a calculation on constant values (eg: `ROM_BYTES("'A', 'A'+1") ;`) Like strings, a nul byte is added for padding when the number of specified bytes is odd.

3. `ROM_WORDS(data)`

   Insert the specified sequence of words (ie: 16 bit) in ROM. The `data` parameter can be any sequence of numbers, enclosed with double quotes
   (eg: `ROM_WORDS("1000,200,123, 0xFFFF") ;`).

8.7.2 Passing immediate ROM data to a subroutine

The first way to access ROM data from program is to embed that data in the code itself, exactly like the immediate operands are hard-coded in instructions. This goal can be achieved by locating the data at the return address of the subroutine.

Since the return address can be found in the `TOSL/TOSH/TOSU` registers, the subroutine is able to access the ROM data. Although it seems freestyle, this way is very handy from the end-user point of view. For example, the LCD library provides the `void lcd_RIprint_() ;` routine that uses this type of parameter passing. As a ≪first step≫ example, the following code sends a message to a LCD display:

```
lcd_RIprint_();
ROM_TXT("Hello !\0") ;
```

A better way is to use a new macro that hides the real nature of the message:

```
#define lcd_RIprint(txt) { lcd_RIprint_() ; ROM_TXT(txt) ; }
```

Finally, the following code will send the message:

```
 lcd_RIprint("Hello !\0") ;
```

In this example, the 'R' stands for ROM and the 'I' stands for immediate.

Writing a function such as lcd_RIprint_() is not easy because it needs a clear undestanding of the PIC-18 instruction set and how the program is compiled. People interested by this point can read the code in the file lcd.slb, that is written in assembly language. However, it is perfectly possible to write such a function in C. For this purpose, the rom.h header provides several very handy macros:

1. PREPARE_ROM_ACCESS

   This macro mainly copy the TOSx registers to the TBLPTRx registers, and set bits needed to access ROM.

2. READ_ROMBYTE

   Reads one byte of data from ROM. The fetched data is stored in the prodl C variable, that is just an alias for the PRODL register. Consecutive invocations of READ_ROMBYTE will read consecutive data from ROM.

3. READ_ROMWORD

   Reads one word of data from ROM. The fetched data is stored in the prodhl C variable, that is just an alias for the PRODL/PRODH pair of registers. Consecutive invocations of READ_ROMWORD will read consecutive data from ROM.

4. FINISH_ROM_ACCESS

   Ends the transaction with ROM. The TBLPTRx registers are copied back to TOSx registers. The macro takes care of alignment, so the address stored in TOSx, is always even. Obviously, the use of this macro is mandatory in this context.

Here is an example of how to uses the proposed macros. This example implements a ROM version of the following puts() routine.

```
void puts(char *p)
{
  for( ; *p ; ++p)
    putchar(*p) ;
}
```

The first step is to define a macro for convenience.

```
#define RIputs(str) { RIputs_() ; ROM_TXT(str) ;}
```

The second step is to write a function that read from ROM memory every char to be printed. The number of char can be odd or even because the FINISH_ROM_ACCESS macro restores a correct parity.

```
void RIputs_()
{
  PREPARE_ROM_ACCESS ;  READ_ROMBYTE ;
  while( prodl )
  {
    putchar( prodl ) ; READ_ROMBYTE ;
  }
  FINISH_ROM_ACCESS ;
}
```

Please note an important point: *all the data stored in ROM must be read.* Violating this rule will lead to execute data instead of machine code and will crash the processor.

Here is another example: this routine fetches 16 bit data from a ROM table, and displays it. In this example, the data is preceeded by a word indicating the size of the table.

```
#define RIputwords(list) { RIputwords_() ; ROM_WORDS(list) ; }

void RIputwords_()
{
  int k ;
  PREPARE_ROM_ACCESS ; READ_ROMWORD ;
  for( k = prodl ; k ; --k)
  {
    READ_ROMWORD ;
    outdec(prodhl) ; putchar(' ' ) ;
  }
  FINISH_ROM_ACCESS ;
}
```

This function is very simple to use:

```
RIputwords("3, 1000, 2000, 3000") ;
```

Notice that despite the parameter of `RIputwords()` is a literal, the data really stored in ROM are words (not a string of chars).

8.7.3 Passing ROM data to a subroutine with a pointer to ROM

In the previous section, blocks of data in ROM were anonymous.

The `ROM_ENTRY()` macro allows to attach an identifier to a location in ROM as following:

```
ROM_ENTRY(hello)
{
   ROM_TXT("hello guys !\0") ;
}
```

The `hello` identifier has the type `ROMptr` and can be passed to any routine recieving this kind of pointer. For example, the `void lcd_Rputs(ROMptr) ;` (from the LCD library) allows such an usage.

```
void f()
{
  lcd_Rputs(hello) ; // displays ≪hello guys !≫
}
```

As previously, it is easy to write such a code at C level. For that purpose, the `rom.h` header provides a macro `ROM_POINTER` that allows to declare that a `ROMptr` will be used to access ROM.

For example, suppose we want to implement a new version of `puts()` that access the character to be printed from a ROM pointer.

```
void Rputs(ROMptr p)
{
```

```
  ROM_POINTER(p) ;
  READ_ROMBYTE ;
  while( prodl )
  {
    putchar( prodl ) ; READ_ROMBYTE ;
  }
}
```

Not really complex, isn't it ? But the next way to access ROM is even more simple, and more powerful.

8.7.4 Accessing data in ROM with a ROM accessor

In the previous sections, ROM data were traversed in sequence. Using a ROM accessor, ROM data can be traversed randomly. A ROM accessor is simply a function that mimic the behaviour of an access to array's elements.

Five macros are available to declare a ROM accessor. For example, the `ROMF_TXT` macro allows to declare a text, *and* the way to access it:

```
ROMF_TXT( atext , "whiizz !\0")
```

Here, `atext` is a ROM accessor for the specified string. It means that `atext(0)` returns `'w'`, `atext(1)` returns `'h'` , and so on. The type of `atext` is `ROMF_i8_t`. The `F` stands for function, because accessors are technically functions receiving an `unsigned int`.

Here is another flavor of the usual `puts()` function :

```
void RFputs(ROMF_i8_t p)
{
  uint8_t k ;
  for( k = 0 ; p(k) ; ++k)
    putchar(p(k)) ;
}
```

The following table shows the available accessors, and their corresponding types.

accessor declaration	accessor type	value type	example
ROMF_TXT	ROMF_i8_t	int8_t	ROMF_TXT(a,"hello\0")
ROMF_DATA8	ROMF_i8_t	int8_t	ROMF_DATA8(b,"-1,2,0xFF")
ROMF_DATA8U	ROMF_ui8_t	uint8_t	ROMF_DATA8U(c,"1,2,0xFF")
ROMF_DATA16	ROMF_i16_t	int16_t	ROMF_DATA16(d,"-1,0xFF34")
ROMF_DATA16U	ROMF_ui16_t	uint16_t	ROMF_DATA16U(e,"1,12300")

9 Features

9.1 Preprocessor

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

9.2 Data types

9.2.1 Numeric data types

cpik supports the following numeric data types:

cpik type	**representable values**	**constant suffix**
`char`	[-128..+127]	none
`unsigned char`	[0..255]	none
`int`	[-128..+127]	none
`unsigned int`	[0..255]	none
`long`	[-32768..32767]	L or l
`unsigned long`	[0..65535]	UL or ul
`long long`	[-2147483648..2147483647]	LL or ll
`unsigned long long`	[0..4294967295]	ULL or ull
`float`	$[-3.402823 \times 10^{38}..+3.402823 \times 10^{38}]$	optional F or f
unsigned bit field	*depends on field size*	none
signed bit field	*depends on field size*	none

Signed integers are represented in 2's complement. Floating point support (`float`) is compliant with the IEEE 754 standard. As stated by the standard, the precision is 6 or 7 decimal digit, depending on the situation. The floating point library is fully written in assembly language, so it is likely to be fast.

9.2.2 ANSI types

Most other PIC compilers consider the type `int` as a 16 bit integer. I prefer to consider it 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 gain 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.

People who are not happy with 8 bit `int`s, or `unsigned long long` declarations, can use the standard ANSI types declared (with a `typedef` instruction) in the `<types.h>` header file as following:

cpik type	**ANSI type**
`char`	`char_t`
`unsigned char`	`uchar_t`
`int`	`int8_t`
`unsigned int`	`uint8_t`
`long`	`int16_t`
`unsigned long`	`uint16_t`
`unsigned long`	`size_t`
`long long`	`int32_t`
`unsigned long long`	`uint32_t`

9.2.3 `void` type

The `void` type is recognized in the traditionnal way.

9.2.4 Pointers

Any valid pointer can be declared.

9.2.5 Type safety

Types are carefully checked, and mixed-type pointer expressions are rejected.

9.2.6 Cast and type promotion

The `cast` operator allows type conversion, and type promotion is implemented, as specified by the standard.

9.2.7 `const` qualifier

The `const` qualifier is implemented, as specified by the ANSI standard. However, text literals like `"Hello"` are not considered as arrays of `const char` (as they should be), but as arrays of `char`. This is consistant with most existing C compilers. For this reason, the following definitions are both correct, but the first one is obviously preferable.

```
const char *p = "xxx" ;
char *q = "zzz" ;
```

Remember that `const` objects must be initialized.

```
const int data1 ;     // rejected
const int data2 = 2 ; // fine
const int *pdata1  ;  // fine (pointed data is constant but pointer is not)
int * const pdata2 ;  // rejected  (pointer is constant)
```

Here are some frequent usages of `const` with formal parameters.

```
void f(const char *p)
{
  ++p ;    // fine
  ++(*p) ; // rejected
}
void g(char * const p)
{
  ++p ;    // rejected
  ++(*p) ; // fine
}
void h(const char * const p)
{
  ++p ;    // rejected
  ++(*p) ; // rejected
}
```

9.3 Data structuration

9.3.1 Array

cpik currently supports arrays in any valid way.

9.3.2 Struct and Union

`struct` and `union` are 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 by functions.

`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.

`struct`s are guaranteed to be compact: as PIC-18 architecture does not impose any alignment constraint, the size of a `struct` is the sum of the sizes of its members[8].

`struct`s cannot be larger than 128 bytes[9], but I suppose it is not a terrible limitation for a 8 bit microcontroller.

Members of `struct`s can be signed or unsigned bit fields. As stated by the standard, a bit field cannot cross an `int` boundary, so the size of a bit field can range from 1 to 8 bits. For example, in the following code

```
struct XXX
{
  unsigned a: 4 ,
           b: 4 ;
} xxx ;

struct YYY
{
  unsigned a: 5 ,
           b: 4 ;
} yyy ;
```

the variable `xxx` is exactly 1 byte long, but the variable `yyy` is 2 bytes long because the second field is one bit longer to be inserted in the first byte. Because signed bit fields often need a sign extension when used, is is much more efficient to use unsigned bit fields.

Notice that bits are specified from low to high (so, in the above examples, `a` is the low nibble)

9.4 Symbolic constants

Symbolic integer constants can be defined with the traditionnal `enum` declarator.

```
enum numbers { one=1, two, three } ;
```

or

```
typedef enum  { peugeot=1, renault=2, citroen=4, other } cars ;
```

[8] However, this feature does no apply to structures composed of bit-fields, as explainded below.
[9] This feature is due to hardware limitation

In these examples, values of `one`, `two`, `three` are respectively `1`,`2` and `3`, and values of `peugeot`,`renault`,`citroen` and `other` are respectively `1`,`2`,`4`, and `5`.

Remember that constant defined with enum are `int`s, so the specified values must range from `-128` to `255` (`255` and `-1` being in fact the same constant).

9.5 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[10], and can be used to produce re-entrant code.

9.6 Static data initialization

All *statically allocated*[11] variables/arrays/structures/unions can be statically initialized, as usual in C. Partial initialization of arrays or structs/unions is supported, and constant expressions in initializers are evaluated at compile time, whenever possible.

Initialization from a symbolic constant expression is supported (eg: expressions such a `(t1-t2)+1`, where `t1` and `t2` are static arrays). In this case, symbolic expressions are generated and address calculation is done at assembly time.

When initializing an union, the type of the initializer must match the type of the first member of the union.

```
/* The following is supported  */
int k = 4 ;                 /* simple initialization */
float  speed = 31.4259e-1 ; /* simple initialization */
char str[] = "hi !" ;       /* array size inferred from initializer */
int array[2][3] = {{1,2,3},{4,5,6}} ; /* initialization of array of arrays */
long z[10] = {0};           /* partial initalization, missing data replaced by 0s */
void f( float ff )
{
  static char t='Z'-26 ;      /* compile-time constant folding */
  static char msg1[] = "OK" ; /* 3 elements static array */
 /* ... */
}
int *pk = &k +1;            /* address calculation deferred to assembly stage */
void (*pf)( float ) = f ;   /* idem */
struct xxx a = { 1, 55 } ;  /* structure initialization */
struct xxx b = { 5 } ;      /* partial  structure initialization */
```

9.7 Non static data initialization

Automatic scalar variables can be initialized. However automatic array/structure/union variables cannot be initialized: this point is conform with the old K&R standard, but is a deviation from the ANSI standard.

```
void f()
{
  float x = 3.14, x2 = 2 * x  ;  // OK
  int t[] = { 1, 2, 3 } ;        // NOT SUPPORTED
```

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

[11] Local variable are not statically allocated, unless they are declared with `static`.

```
  /* ... */
}
```

9.8 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 variables 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).

9.9 Address allocation

The address of each global entity is 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 placed at any address, using the `@`*address* extension. For example:

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

This feature 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 that takes the informations from Microchip's ≪`.inc`≫ files. For example, here is the beginning of the `p18f2525.h` header file:

```
#ifndef DEVICE
#define DEVICE p18f2525
#define p18f2525

// =====================================
//         PROCESSOR : p18f2525
// =====================================

// This file has been automatically generated from Microchip's "p18f2525.inc" file.
// with the inc2h-v2 utility.             Please do not edit by hand.
// Do not use with cpik versions prior V0.7, report problems to author.
// (C) Alain Gibaud  2012    (alain.gibaud@free.fr)

#pragma firstsfr 0xf80
// ------------------------------
//        PORTA
// ------------------------------
unsigned int PORTA@0xf80 ;
```

```
union
{

struct
{
  unsigned int
  RA0 : 1 ,
  RA1 : 1 ,
  RA2 : 1 ,
  RA3 : 1 ,
  RA4 : 1 ,
  RA5 : 1 ,
  RA6 : 1 ,
  RA7 : 1 ;
} ;

struct
{
  unsigned int
  : 4,
  TOCKI : 1 ,
  AN4 : 1 ;
} ;

struct
{
  unsigned int
  : 5,
  SS : 1 ;
} ;

struct
{
  unsigned int
  : 5,
  NOT_SS : 1 ;
} ;

struct
{
  unsigned int
  : 5,
  LVDIN : 1 ;
} ;

struct
{
  unsigned int
  : 5,
  HLVDIN : 1 ;
} ;

} PORTAbits@0xf80  ;
...
```

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

9.10 Instructions

All the instructions of the C language are implemented.

The support for the `switch` instruction has been developed by Josef Pavlik. This support is especially effective and implements different strategies for code generation, depending on case values.

This implementation only supports case-values that can be coded with 8 bits (eg: these values must range from -128 to 255, and -1 and 255 are aliases). For this reason, `long` or `long long` case selectors are truncated to `int` before executing the code selection. I have decided to keep this feature because this is not a severe limitation, and it allows to guarantee high speed and compact code.

9.11 Operators

All the operators of the C language are implemented.

9.12 Extensions

9.12.1 Binary constants

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

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

9.12.2 Digit separator

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

```
T2CON = 0b0_1110_0_11 ; // same as  0b01110011, but highlight bits fields
```

9.12.3 Assembler code

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

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

The `__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.

9.12.4 Interrupt service routines

Two empty interrupt service routines are provided by the run-time library (`/usr/share/cpik/<version>/lib/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 */
}
```

Note that this routine will replace the default one, because user libraries are scanned before `rtl.slb`.

The keyword `__interrupt__` insures 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` [12] .

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[13].

When the interrupt code is written in C (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 ;  /* MANDATORY */

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

   RESTORE_REGS ; /* MANDATORY */
}
```

The `SAVE_REGS` and `RESTORE_REGS` macro are defined in the `<interrupt.h>` header, that should be included. These macros save the registers that are used by integer arthmetic routines, but *do not* save the registers that are used for floating point calculation. For this reason, *do not use* floating point data in an interrupt routine. However, if you really need to handle FP data in an interrupt routine, it is easy to write your own macro to save the FP context. See section 8.2 for details about the registers usage.

It is possible to take a full control of context saving. In this case, just omit the `__interrupt__` keyword and insert yourself the code you need with `__asm__` instructions. Beware! in this case you must know what you are doing.

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

[13] Caution: Never alter the FSR0 stack pointer in a low level ISR if high level interrupts are enabled.

9.12.5 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.

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

If you plan to use asm code and interruptions together, you must enforce a simple rule : the software stack pointer (`FSR0`) must *always* 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.

9.12.6 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` : disable high priority interrupts.
2. `MASK_LO_PRI_IT` : disable low priority interrupts.
3. `UNMASK_HI_PRI_IT` : enable high priority interrupts.
4. `UNMASK_LO_PRI_IT` : enable 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, and never enter a critical section inside a critical section.
All these macros are defined in `<interrupt.h>` header.

9.12.7 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 the device configuration bits.

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

   Config bits values 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.

9.12.8 Explicit bit fields

This very handy feature allows to use any bit slice of a 8 bit variable in an expression. For this purpose, `cpik` provides the non-standard syntax

```
var.OFFSET:SIZE
```

which corresponds to a slice of `SIZE` bits starting at bit number `OFFSET` of `var`.

Notice that :

1. The above syntax is **not** the invocation of an operator, it just a way to define a temporary sub-variable corresponding to a bit slice. As a consequence, `var` must be an existing variable identifier and *cannot be an expression.*
2. `OFFSET` and `SIZE` must be integer constants, or integer constant expressions. (`OFFSET` can range from `0` to `7` and `SIZE` can range from `1` to `8`).
3. Obviously, `OFFSET+SIZE` cannot be higher than 8 because a bit field cannot cross a byte boundary.
4. A bit field that is 8 bit wide is not rejected, but is viewed as a plain byte by the compiler.

As an example, suppose we need to copy `PORTB<0-3>` (configured as input) to `PORTB<7-4>` (configured as output).

- Without using bit fields (`cpik` $< V0.7$) , you will probably have to write something like:

  ```
  uint8_t  x ;
  x = PORTB << 4 ;
  PORTB &= 0x0F ; // clear dest bits
  PORTB |= x ;
  ```

- Another alternative is to define a structure, and to map it to PORTB's address :

  ```
  struct
  {
    unsigned low:4,
             hi:4 ;
  }
  my_portb@0xF81 ;

  my_portb.hi = my_portb.low ;
  ```

  Of course, this code supposes that `PORTB` is at address `0xF81`, and will fail if it is not the case anymore.

- The third solution is to use explicit bit fields: the corresponding code is really straightforward:

  ```
  PORTB.4:4 = PORTB.0:4 ;
  ```

  Of course, if you need your code to be easily reconfigurable, a couple of macro will do the job:

  ```
  #define LNIBBLE    0:4
  #define HNIBBLE    4:4

  PORTB.HNIBBLE = PORTB.LNIBBLE ;
  ```

Note that explicit bit fields are *always* unsigned.

cpik is now released with header files which allow to handle device registers as structures or explicit bit fields. Please see section 11.1 for details.

10 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.

10.1 Access to 16 bit SFR

16 bit SFRs (Special Function Register) are made of two 8 bit registers. You can access them as single 16 bit variables located at same address than the 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 ;
```

10.2 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 the 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 stores 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's designers to allow atomic writes to 16 bit data on a 8 bit processor, but forces to write *hi byte first*, then *low byte*. Please see Microchip documentation for details.

Unfortunatly, the code generated by `cpik` may write 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 recommend to perform reads with same care.

10.3 How to initialize EEPROM data

There is currently no specific `#pragma` to force EEPROM data value during chip programming. If you plan to initialize the EEPROM, the following will do the job: use the `__asm__()` instruction to insert one (or more) ≪`DE`≫ directive in the emitted code. See the `gpasm` documentation for details about this directive.

Another option is to explicitely put the data at the correct address, as showed by this example.

```
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 a 18F2525 device with 0x00 (the default value for an «erased» chip is all 0xFF). Remember that the function must be used to be included in the final executable file.

10.4 Use `struct` to increase modularity

`struct` usage is a good way to avoid global namespace pollution and 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 ;`

10.5 Do not use uppercase only symbols

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

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

will lead to unexpected and hard to understand error message. The reason 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.

10.6 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 `+=` and `+=` to `+`
4. Avoid to compute twice the same sub-expression (using operators such as `+=` is a good way to enforce this rule)
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. Using global variables leads to smaller and faster code. Be careful, it also leads to lack of modularity and memory wasting. See previous section for modularity issue.
9. Implement 8 bit left shifts with $\times 2$ products: due to availability of hardware multiplication the code will be fast. This rule does not apply to right shifts.
10. Use `switch` instead of cascaded `if` whenever possible. The code will be faster and smaller.
11. Prefer `unsigned` bit fields to `signed` ones. Signed bit fields need frequent sign extensions which are resource consuming.
12. Avoid accessing bit fields from pointers (ie: `s->member`). This operation involves an indirection which is also resource consuming.

11 Headers

This section covers several headers that are part of the standard or that are provided as helpers to write more portable or cleaner code. These headers are not related to a specific library.

11.1 device/p18xxxxx.h

The `device` directory contains an header file for each flavor of PIC18 device. `#`include one of these files when you need to access a register of the the target device (eg: `#include <device/p18f2525.h>`). Since version V0.7.0, the device headers contain various definitions that allow to access each bit of registers using a symbolic identifier. These identifiers can be found in Microchip's datasheets.

Each bit (or bit field) can be accessed using the standard syntax (based on structures) or using the explicit bit field syntax (that is specific to `cpik`). The following example illustrates how the devices' registers are declared.

```
#ifndef DEVICE
#define DEVICE p18f2525
#define p18f2525

// =====================================
//         PROCESSOR : p18f2525
// =====================================

// This file has been automatically generated from Microchip's "p18f2525.inc" file.
// with the inc2h-v2 utility.             Please do not edit by hand.
// Do not use with cpik versions prior V0.7, report problems to author.
// (C) Alain Gibaud  2012    (alain.gibaud@free.fr)

#pragma firstsfr 0xf80

...

// ------------------------------
//        T3CON
// ------------------------------
unsigned int T3CON@0xfb1 ;
union
{

struct
{
  unsigned int
  TMR3ON : 1 ,
  TMR3CS : 1 ,
  T3SYNC : 1 ,
  T3CCP1 : 1 ,
  T3CKPS0 : 1 ,
  T3CKPS1 : 1 ,
  T3CCP2 : 1 ,
  RD16 : 1 ;
} ;

struct
{
  unsigned int
```

```
  : 2,
  NOT_T3SYNC : 1 ;
} ;

// The following are aliases ..
struct
{
  unsigned int
  : 4,
  T3CKPS : 2 ;
} ;

} T3CONbits@0xfb1  ;

#define _TMR3ON 0
#define _TMR3CS 1
#define _T3SYNC 2
#define _T3CCP1 3
#define _T3CKPS0 4
#define _T3CKPS1 5
#define _T3CCP2 6
#define _RD16 7

#define _NOT_T3SYNC 2

// The following are aliases ..
#define _T3CKPS 4:2
...
```

In this example, the `T3CON` register contains both individual bits, and a group of 2 bits (T3CKPS0 and T3CKPS1), which can be manipulated as a bit field. Thus, the following codes are equivalent:

```
T3CONbits.T3CKPS0 = 0 ;  T3CONbits.T3CKPS1 = 1 ; // method 1 : individual bits
T3CONbits.T3CKPS = 0b10 ;                        // method 2 : bit field
```

Moreover, macros are also provided to use the explicit bit field syntax (see section 9.12.8), so one can write:

```
T3CON._T3CKPS0 = 0 ;  T3CON._T3CKPS1 = 1 ; // method 1 : individual bits
T3CON._T3CKPS = 0b10 ;                     // method 2 : bit field
```

Notice that the names `T3CKPS0`, `T3CKPS1`, etc have been choosen to be compatible with the member names used by Microchip, but obviously, they sound like macro names, despite the fact they are not. On the other hand, the identifiers such as `_T3CKPS0`, etc. correspond to genuine macros.

11.2 sys/types.h

This header helps you to improve the portability of your code. It defines a set of ANSI-compatible integeral types as described in section 9.2.2, but is not part of the ANSI standard.

11.3 macro.h

This header provides a set of general purpose handy macro. In the following, `reg` denotes a 8 bit signed or unsigned integer, and `bit` a bit number ($0 \leq$ `bit` ≤ 7). `macro.h` is not part of the ANSI

standard.

Macro	Role
BIT_1(reg, bit)	Set bit bit of reg variable
BIT_0(reg, bit)	Reset bit bit of reg variable
BIT_TST(reg,bit)	Return true if bit bit of reg is set.
BIT_TOGGLE(reg, bit)	Toggle bit bit of reg
BIT_COPY(treg, tbit, sreg, sbit)	Copy bit sbit of sreg to bit tbit of treg
BIT_NCOPY(treg, tbit, sreg, sbit)	Copy inverse of bit sbit of sreg to bit tbit of treg
BIT_WRITE(treg, tbit, flag)	Copy the boolean value flag to bit tbit of treg
NOP	Generate a nop asm instruction
CLEAR_WATCHDOG	Generate a clrwdt asm instruction

Notice that the bit-oriented macros are made obsolete since V0.7.0 due to the availability of bit fields.

11.4 pin.h

pin.h provides a set of very handy macros that allow to use the bit of each port thru symbolic names. This header has been written by Josef Pavlik.

The first step to use them is to specify the logical name of each bit to be used. The general form of this specification is:

```
#define <logical name> <port name><bit number>
```

for example, the following declare LED to be the logical name corresponding to the bit 3 of port C.

```
#define LED PORTC3
```

Such a logical name can be defined for ports A,B,C,D or E and must be used with the following macros:

Macro	Role
PIN_SET_INPUT(name)	declare name as input bit
PIN_SET_OUTPUT(name)	declare name as output bit
PIN_SET_OUTPUT0(name)	declare name as output bit, and force it to 0
PIN_SET_OUTPUT1(name)	declare name as output bit, and force it to 1
PIN_SET(name)	force name to 1
PIN_1(name)	synonym for PIN_SET
PIN_CLR(name)	force name to 0
PIN_0(name)	synonym for PIN_CLR
PIN_TOGGLE(name)	toggle name
PIN_READ(name)	return 0 when name is clear, else return a non-nul value
PIN_TST(name)	synonym for PIN_READ
PIN_WRITE(name,value)	clear name when value is 0, else set it

11.5 stdarg.h

stdarg.h is the standard header that defines the macros va_start, va_end and va_arg. You must include this header when you plan to define a function accepting a variable argument list.

11.6 `float.h`

`float.h` is the standard header that defines various constants relative to the floating point support. For example, `FLT_MAX` (the greatest representable floating point number) is defined in this file, together with many other constants.

11.7 `assert.h`

`assert.h` is the standard header that defines the `assert()` macro. `assert()` is generally used during tests to check that a given condition is always verified. The run-time support for this functionality is implemented in `stdlib.c`.

12 Libraries

Until 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`, `float.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 file (compiled version of `x.c`

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`.

However, source libraries generated from C files are not distributed because they depends on compiler and run time code version. You need to recompile the source code yourself.

As the compilation is really very fast, I generally include the source code of the libraries I use in my `pikdev` projects, so they are compiled with the rest of the application.

12.1 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 (see below).

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

12.1.1 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. The 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. The intput vector is not initialized, so this function must be used prior any input.
   `input_hook` is defined by `typedef char (*input_hook)() ;`

12.1.2 output functions

1. `void putchar(char c)`

   Writes character `c`.

2. `int puts(char *s)`

   Writes string `s`. Always returns 0.

3. `int RFputs(ROMF_i8_t p)`

   Writes the string corresponding to the rom accessor `p`. Please see section 8.7 for details about data located in ROM. Always returns 0.

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

   Writes the 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. Any other value leads to unpredictable result.

   Due to automatic type conversion and leading zeros suppression, this function can be also used for 8 bit numbers

5. `void outhex32(unsigned long long n, char up)`

   Writes the unsigned long 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.

   Due to automatic type conversion and leading zeros suppression, this function can be also used for 8 bit or 16 bit numbers. However I do not recommend this option if ressources are limited because it leads to import unnecessary 32 bit code in your application.

6. `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.

7. `void outdecu32(unsigned long long n)`

   Writes unsigned long long `n` in decimal. Leading 0 are suppressed, so this function can be also used for 8 bit or 16 bit numbers, which are promoted to `unsigned long long` before call. However I do not recommend this option if ressources are limited because it leads to import unnecessary 32 bit code in your application.

8. `void outdec(long n)`

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

9. `void outdec32(long long n)`

   Writes long long `n` in decimal. Leading 0 are suppressed, so this function can be also used for 8 bit or 16 bit numbers. However I do not recommend this option if ressources are limited because it leads to import unnecessary 32 bit code in your application.

10. `int putfloat (float x, int prec, int format)`

    Writes the `float` number `x` on the standard output. The `prec` parameter specify the number of digit that must be printed after the decimal point. `prec` can range from `0` to `7`. A negative value ask the function to print with the maximum precision. The `format` parameter can be either `'f'` (classic format like in «`1.0022`»), `'e'` or `'E'` (scientific format like in «`31.4158e-1`» or «`-1.0E4`»).

11. `int printf(const char *fmt, ...)`

    Mini implementation of the standard `printf()` function. Recognized conversion specifications are :
    `%c %s %d %u %x %ld %lu %lx %lld %llu %llx %f %e %E`

    See more information about conversion specifiers in section 12.1.3

    This function return the number of characters printed.

12. `int RFprintf(ROMF_i8_t fmt)`

    Version of `printf()` receiving its format string thru a rom accessor. Please see section 8.7 for details about what is a rom accessor.
    See more information about conversion specifiers in section 12.1.3

    This function return the number of characters printed.

12.1.3 Conversion specifiers supported by the `printf()` family

The `printf` function is very handy, but very memory consuming when all the data types are simultaneously supported. For this reason, the support for 32 bit integers and floating point numbers must be explicitely enabled to become active. If you need this support, just `#`define the macros `INT32_IO` and/or `FLOAT_IO` when `stdio.c` is compiled[14].

Specifier	**Data types** of parameter	**Support**
`%c`	`int, char`	always
`%s`	`char *, int *, etc.`	always
`%d`	`int, char`	always
`%u`	`unsigned int, unsigned char`	always
`%x`	`int, char, unsigned int, unsigned char`	always
`%ld`	`long`	always
`%lu`	`unsigned long`	always
`%lx`	`long, unsigned long`	always
`%lld`	`long long`	`INT32_IO` defined
`%llu`	`unsigned long long`	`INT32_IO` defined
`%llx`	`long long, unsigned long long`	`INT32_IO` defined
`%f`	`float` (standard notation)	`FLOAT_IO` defined
`%e`	`float` (scientific `e` notation)	`FLOAT_IO` defined
`%E`	`float` (scientific `E` notation)	`FLOAT_IO` defined

In order to save memory, `printf()` do not support minimal width specifiers, such as `%3ld`.

However, a precision specifier is supported for `%f` `%e` and `%E`. For example, `%3e` ask `printf()` to print 3 decimal digits after the decimal point. The specified precision can range from `0` to `7` (that is the default value).

Remember that the number of significant digits is 6 or 7 for the IEEE-754 floating point numbers. For example, printing $\frac{1.0E4}{3}$ with more than 3 digits after the decimal point doesn't make sense because the least significant written digits do not represent anything in the result.

12.1.4 input

1. `long getch()`

   Returns the next available character, or EOF if no character is available. This character is not echoed thru the output vector. This input is not buffered.

2. `long getche()`

   Returns the next available character, or EOF if no character is available. This character is echoed thru output vector. For this reason, `set_putchar_vector()` must be used prior any use of `getche()`. This input is not buffered.

3. `long getchar()`

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

   All «high level» functions such as `scanf()` or `gets()` use `getchar()` as low level input function.

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

[14] You can either edit the source code, or just use `-DINT32_IO -DFLOAT_IO` at `cpik` invocation.

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

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

   `fillbuf` returns the number of character stored in the input buffer, including the `'\n'` terminator, but excluding the `'\0'` trailer.

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

   `fillbuf` interprets the `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 getlong(long *pn,int base);`

   Reads a signed long integer in base `base`, and store it at location pointed to by `pn`. Returns 1 if successful, else 0.

7. `int getlong32(long long *pn,int base);`

   Reads a signed long long integer in base `base`, and store it at location pointed to by `pn`. Returns 1 if successful, else 0.

8. `int getfloat(float *pf);`

   Reads a `float` number, and store it in the variable pointed to by `pf`.

   This function returns the number of digits of the number (ie: returning `0` means that the input has failed).

9. `int scanf(const char *fmt, ... ) ;`

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

   This function returns the number of conversion specifiers successfully processed.

10. `int RFscanf(ROMF_i8_t fmt, ... ) ;`

    Version of `scanf()` receiving its format string thru a rom accessor. Please see section 8.7 for details about data located in ROM.

    This function returns the number of conversion specifiers successfully processed.

12.1.5 Conversion specifiers supported by the `scanf()` family

Like `printf()`, `scanf()` doesn't support all the conversion specifiers by default. If you need this support, just #define the macros `INT32_IO` and/or `FLOAT_IO` when `stdio.c` is compiled[15].

[15] You can either edit the source code, or just use `-DINT32_IO -DFLOAT_IO` at `cpik` invocation.

Specifier	**Data types** of parameter	**Support**
%c	*pointer to* int, char	always
%s	*pointer to* char, int, etc.	always
%d	*pointer to* int, char	always
%u	*pointer to* unsigned int, unsigned char	always
%x	*pointer to* int, char, unsigned int, unsigned char	always
%ld	*pointer to* long	always
%lu	*pointer to* unsigned long	always
%lx	*pointer to* long, unsigned long	always
%lld	*pointer to* long long	INT32_IO defined
%llu	*pointer to* unsigned long long	INT32_IO defined
%llx	*pointer to* long long, unsigned long long	INT32_IO defined
%f	*pointer to* float (standard or scientific notation)	FLOAT_IO defined

12.2 Standard math library

Since V0.6.0, cpik comes with a near complete standard math library. Here is a list of the well known math functions that are supported. These functions are prototyped in math.h, that also #defines numerous standard macros such as PI, TWO_PI, HALF_PI, QUART_PI, etc.

12.2.1 Trigonometric functions

1. `float sinf( float x) ;`
2. `float cosf( float x) ;`
3. `float tanf( float x) ;`
4. `float cotf( float x) ;`
5. `float asinf( float x) ;`
6. `float acosf( float x) ;`
7. `float atanf( float x) ;`
8. `float atan2f( float x, float y);`

12.2.2 Hyperbolic functions

1. `float sinhf( float x) ;`
2. `float coshf( float x) ;`
3. `float tanhf( float x) ;`

12.2.3 Exponential, logarithmic and power functions

1. `float expf( float x);`
2. `float logf( float x) ;`
3. `float log10f( float x) ;`
4. `float powf( float x, float y);`
5. `float sqrtf( float a) ;`

12.2.4 Nearest integer, absolute value, and remainder functions

1. `float fabsf( float x) ;`
2. `float frexpf( float x, int *pw2);`
3. `float ldexpf( float x,  int pw2);`
4. `float ceilf(float x) ;`
5. `float floorf(float x) ;`
6. `float modff(float x, float * y);`

12.3 Standard stdlib library

This library is not complete yet. It contains the following standard functions :

12.3.1 System

1. `int _assert(long line,char *s) ; // support for the assert() macro`
2. `void exit(int i) ;`

12.3.2 Character processing

1. `int isspace(char c);`
2. `int isdigit(char c);`

12.3.3 Conversions to/from strings

1. `char  *ftoa (float x, char * str, int prec, int format);`
2. `float  strtof(const char *str, char **endptr);`
3. `float  atof(const char *str) ;`
4. `int  atoi(const char *str) ;`
5. `long  atol(const char *str) ;`
6. `long long  atoll(const char *str) ;`

12.4 rs232

Minimum 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.

12.5 LCD

Support for classic HD-44780 based LCD display, in 4 bit unidirectional mode. This kind of interface needs 6 lines (`data4/data5/data6/data7` , `RS` and `E`). Since version `0.5.3` the LCD library is configurable to allow any connection to the target device. It provides for this purpose the following macros:

- `CONFIGURE_LCD_RS(PORTx , pin) ;`
- `CONFIGURE_LCD_E(PORTx , pin) ;`
- `CONFIGURE_LCD_DATA4(PORTx , pin) ;`
- `CONFIGURE_LCD_DATA5(PORTx , pin) ;`
- `CONFIGURE_LCD_DATA6(PORTx , pin) ;`
- `CONFIGURE_LCD_DATA7(PORTx , pin) ;`

Despite the fact this library is written in assembly language, the configuration can be done from the C code: just put the macro invocations somewhere in the `main()` function. For example:

```
// command/data selection pin
CONFIGURE_LCD_RS(PORTB, 5) ;
// enable pin
CONFIGURE_LCD_E(PORTB, 4) ;
// data pins used in 4 bit mode
CONFIGURE_LCD_DATA4(PORTA, 0) ;
CONFIGURE_LCD_DATA5(PORTA, 1) ;
CONFIGURE_LCD_DATA6(PORTA, 2) ;
CONFIGURE_LCD_DATA7(PORTA, 3) ;
```

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

1. `void lcd_init(int delay) ;`

   Initialize the LCD 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_init_Mhz(int Mhz) ;`

   This is a macro, provided for convenience as an alternative to `void lcd_init(int delay)`. The Mhz parameter is simply the clock frequency (in Mhz) of the target processor.

3. `void lcd_putchar(char c);`

   Displays character `c` at current cursor position. This function can be used as the parameter of `set_output()` to redirect outputs to the LCD display.

4. `void lcd_move(int pos) ;`

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

5. `void lcd_clear() ;`

   Erases display.

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

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

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

   Defines a new character with code `c`.

   Definable character 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.

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

   Displays `c` as two hex digits.

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

   Displays `n` as four hex digits.

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

    Displays a nul-terminated character string.

11. `void lcd_Rputs(ROMptr s) ;`

    Displays a nul-terminated character string pointed to by rom pointer p. Please see section 8.7 for details about data in ROM.

12. `void lcd_RIputs(ROM literal) ;`

    Displays a nul-terminated character string defined by an immediate rom literal. Please see section 8.7 for details about data in ROM.

13. `void lcd_putcmd(char cmd) ;`

    Enters command mode, then send command `cmd` to LCD display. This function can be used to access more advanced functionnalities of the LCD display.

12.6 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`).

12.7 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.

12.8 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 come 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
  }
}
```

13 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>
   ```

   The module name can be optionally followed by the specification of a program section, with the following syntax:

   ```
   ;<+module_name|section_type>
   ```

   The `cpik` linker supports 4 types of program sections :

   (a) `CDATA`
   This segment is dedicated to const data. Such data will be located at begin of ROM and will not copied to RAM

   (b) `IDATA`
   This segment is dedicated to initialized data. The module must contain the ≪`;<=`≫ tag (see below) with exact number of bytes to be used for initialization

   (c) `UDATA`
   This segment is dedicated to uninitialized data and is *filled with nul bytes during boot.*

   (d) `CODE`
   This segment contains all other kind of modules (code, symbols, etc.)

   A module with no section specification will be included in the `CODE` program section.

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|IDATA>
  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>
;<->
```

14 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.

15 Contributors

- **Alain Gibaud**

  Original author of the compiler and run-time library.

- **Josef Pavlik**
  - Bug fixes in library and code of compiler,
  - `pin.h` header,
  - Implementation of `enum` declarator and `switch` instruction,
  - Optimisation of the run-time support for shifts,
  - Dead-code elimination in the case of tests or loops with a constant condition,
  - Optimisation of the static data initialization,
  - Optimisation of tests in some special situations,
  - Post-compilation branch optimizer that eliminates far jumps whenever possible.

  Thank you Josef!

16 Credits

Most of the code of the `cpik` project is original. However, some parts of the standard libraries are an adapted version of codes available under GPL license.

- **Math library** by Jesus Calvino-Fraga, jesusc@ieee.org
- `strtod` **function** by Michael Ringgaard (a modified version of this code is used by the stdio library)
- **Core algorithms for floating point support** by Pipeline Associates, phw@motown.com (rewritten in PIC18 assembler).

17 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 one person.

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 pushing a new compiler to free software, you can contribute in many manners :

17.1 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 fit) your needs.

17.2 Bug reports

If you detect a bug, please send me the most simple source code that provokes this bug. I (maybe) will be able to analyse the generated code and fix the problem.

17.3 Documentation

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

17.4 Libraries

`cpik` needs more libraries. All kinds of them. Some libraries are easy to write (`stdlib/string`) but I have no time to do it. Some are really hard to code (USB support).

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

18 inc2h-v2

18.1 what is inc2h-v2 ?

inc2h-v2 is a simple program that allows to make device-specific C headers from ≪.inc≫ files. These files describe the SFR available for each device flavor, together with the bits or bit fields they contain. The ≪.inc≫ files are available in Microchip's tools suite, and also in gputils packages.

Because cpik provides a lot of ready-to-use header files, you will probably not need this utility. However, you may have to use it when a new device appears on the market.

18.2 How to build inc2h-v2 ?

1. Unpack the inc2h-v2.tar.gz archive

   ```
   tar xzvf inc2h-v2.tar.gz
   ```

   This command will create the following files in the current directory: main.cpp, processor.h, processor.cpp, bf.h, bf.cpp.

2. Build the program (you need the g++ compiler from GNU)

   ```
   g++ -Wall -O2 -o inc2h-v2 bf.cpp processor.cpp main.cpp
   ```

18.3 Command summary

You can use inc2h-v2 in three ways.

1. inc2h-v2 -locate

   inc2h-v2 do its best effort to discover where the .inc files are located on your hard disk, and displays this location. This command is useful if you don't know where the gputil tool suite has been installed.

2. inc2h-v2 [options] <file>.inc

   inc2h-v2 processes <file>.inc and creates the corresponding header file (ie: <file>.h). notice that this file is created in the current directory (not the directory where the ≪.inc≫ file is located).

3. inc2h [options] -all <directory>

   inc2h-v2 processes *all* the ≪.inc≫ files located in the specified directory and creates the corresponding header files in the current directory. Notice that this command ignores the ≪.inc≫ files that do not correspond to a pic18 device.

The options allow to control the form of the generated files:

- -struct

  The description of device registers (SFRs) uses structs containing bit fields. This is the standard way to manipulate the registers. The member names of these structs are fully compatible with the identifiers used by Microchip's C18 compiler.

- -define

  The description of device registers is based on macro definitions. This mode is intended to be used with the explicit bit field syntax allowed by cpik.

The default behaviour is to create the two kind of information in the same file. Using one of this options has the avantage of keeping the resulting files relatively small.