

cpik

C compiler for PIC_©-18 devices

Alain Gibaud
alain.gibaud@free.fr
Version 0.5.3 (Doc rev e)

October 2, 2011

Contents

1	Introduction	3
2	What is new ?	3
3	The «philosophy» behind cpik	3
4	A very special feature	4
5	Command syntax	5
5.1	Compilation	5
5.2	Link	6
5.3	Final assembly and jump optimizer	6
6	Implementation details	7
6.1	Stacks	7
6.2	Memory layout	7
6.3	Register usage	8
6.4	Computation model	8
6.5	Function calling conventions	9
6.6	Optimizations	10
6.7	How to place data in ROM	12
6.7.1	Creating a block of data in ROM	12
6.7.2	Passing immediate ROM data to a subroutine	12
6.7.3	Passing ROM data to a subroutine with a pointer to ROM	14
6.7.4	Accessing data in ROM with a ROM accessor	15
7	Implemented features	16
7.1	Preprocessor	16

7.2	Data types	16
7.3	Data structuration	17
7.4	Symbolic constants	17
7.5	Storage classes	17
7.6	Static initialization	17
7.7	Scope control	18
7.8	Address allocation	18
7.9	Instructions	19
7.10	Operators	19
7.11	Extensions	19
7.11.1	Binary constants	19
7.11.2	Digit separator	19
7.11.3	Assembler code	19
7.11.4	Interrupt service routines	20
7.11.5	Why and how to write interruptible code	21
7.11.6	Disabling and enabling interrupts	21
7.11.7	Pragmas	21
8	Hints and tips	22
8.1	Access to 16 bit SFR	22
8.2	Access to 16 bit SFR - second part of the story	22
8.3	How to initialize EEPROM data	22
8.4	Avoiding global namespace pollution increases modularity	23
8.5	Do not use uppercase only symbols	23
8.6	How to write efficient code	24
9	Headers	24
9.1	types.h	24
9.2	macro.h	24
9.3	pin.h	25
10	Libraries	25
10.1	standard IO library	26
10.1.1	IO redirection	26
10.1.2	output functions	26
10.1.3	input	27
10.2	rs232	28
10.3	LCD	28
10.4	AD conversion	30
10.5	EEPROM read/write	30

10.6 Timer 0	31
11 Source library structure	32
12 Needed software	34
13 Contributors	34
14 How to contribute to the cpik project ?	34
14.1 Feedbacks and suggestions	34
14.2 Bug reports	34
14.3 Documentation	34
14.4 Libraries	35

1 Introduction

cpik is a C compiler for Microchip PIC 18 microcontrollers. The language recognized by this compiler is a very large subset of ANSI C language, and contains extensions specific to the microcontroller domain. It is a personal project developped on my spare time. Because this spare time tends to zero, the project evolves rather slowly, with small jumps. Anyway, the compiler exists and works well.

2 What is new ?

The current version (0.5.3) fixes bugs in initialization of static data structure and removes a minor deviation (from ANSI C) in the use of pointer to functions. Several libraries (LCD, standard IO) have been improved.

An important improvement is a support for data located in program space (ROM). This feature allows to save RAM space, so it is very useful for small devices. This support is not really part of the compiler: it is implemented with macros and several run-time library routine.

But the most important new is the following: since the current version, this project has received the contribution of Josef Pavlik who has developped the support for the switch instruction, and other features detailed in the section 13. I hope this information will encourage people to join the project.

3 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 single developer (with many other activities), so I had to decide what is important and what is not. My underlying idea is the following: it is better to drop a feature than to incompletely or inexactly implement it.

For example, I choosed 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 `structs` or `unions`. `typedef` has been implemented in V0.3, and structures/unions in V0.4. 32 bit integer arithmetic is supported since V0.5.

A major issue remains in current version (0.5.3): floating point support is not implemented yet, but all the parts of the puzzle are in my head.

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

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

This compiler is written in C++. Any feedbacks concerning bugs, feature requests or criticisms can be addressed to Alain Gibaud (alain.gibaud@free.fr).

4 A very special feature

`cpik` works in a unusual 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 report feedbacks about 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 program¹ compiled with `cpik` ever executed (see figure 1).
- 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` compiler/linker.
- This design has proven its flexibility for the implementation of support for data located in ROM, or jumps optimisations

¹Believe it or not, this program (a simple `for` loop) worked successfully 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.

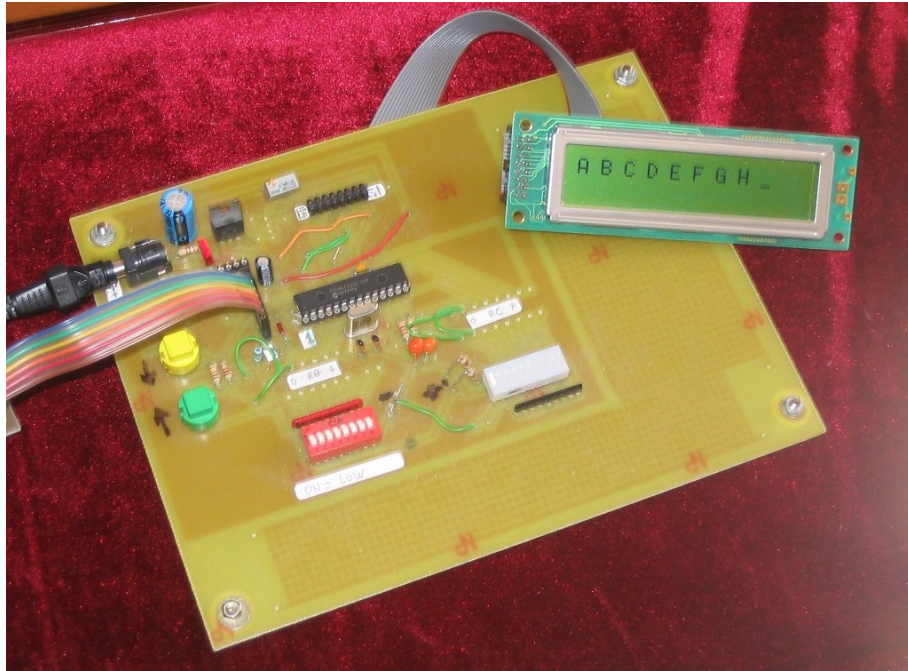


Figure 1: Result of the very first program compiled by `cpik` ever executed

- 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.

5 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 optimizes jumps after the linking process.

5.1 Compilation

```
cpik -c [-v] [-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.

`-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.

`-p device` : specifies the target pic device name. `device` must be a valid pic 18 name like `p18xxxx`. The exact name is not verified, except 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
<code>-d1</code>	print unoptimized intermediate code as comment in .slb file
<code>-d2</code>	print peep hole optimized intermediate code as comment in .slb file
<code>-d4</code>	print symbol tables with entities names and types
<code>-d8</code>	print internal expression trees before optimisations, without type annotation
<code>-d16</code>	print internal expression trees before optimisations, with type annotations
<code>-d32</code>	print internal expression trees after optimisations, without type annotation
<code>-d64</code>	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.

5.2 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.

`-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.

5.3 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_path` : specify the path to the `gpasm` tool. This option is generally not needed.

`-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`.

6 Implementation details

`cpik` generates code for PIC-18 processors running in legacy (ie: non-enhanced) 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. In fact, programs can be larger than 64K bytes (depending on your device flavour), but only the lower 64Kbytes can be reached from pointer to functions. This is not an issue because it is easy to force the addresses of such functions to be less than `0xFFFF`.

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

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

6.1 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 during subroutines execution. 31 levels of nested calls are generally largely sufficient for most applications.

However, recursive applications may provoke overflows of the return stack, this point being under the responsibility 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 back the data.

6.2 Memory layout

The current memory layout used by `cpik` is the following:

Name	Addresses	Usage
Soft Stack	[GG+1->TT]	software stack, grows upward to top of memory
Globals	[PP+1->GG]	global variables
Scratch	[22->PP]	0 to 118 bytes zone for structures
Registers	[12->21]	R0,R1,R2,R3,R4 pseudo-registers
Arithmetic	[2->11]	reserved for 32 bit float and integer arithmetic
IT mask	[0->1]	reserved for cpik code

Addresses from 0 to 21 are reserved for the run-time library. The *Register* zone can be extended by a *Scratch* zone ranging from 0 to 118 bytes. This *Scratch* zone is used when your program uses functions returning a structure. In this case, the maximum size of the returned structure is the *Scratch* zone size+10.

The *Scratch* size can be adjusted by editing the prolog file `/usr/share/cpik/<version>/lib/cpik.prolog`. You just have to edit the `SCRATCH_SIZE EQU xx` line to change the *Scratch* zone size.

However, I plan to implement a more flexible way to specify scratch zone size in the future.

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 (hi addresses) to bottom.

6.3 Register usage

cpik uses four 16 bit pseudo registers named R0,R1,R2,R3,R4. Each Rx register can be splitted in RxH and RxL. These registers are located in page 0, and are efficiently accessed via Access Bank (a=0).

- W is used as a general purpose scratch register
- R0 is the 16 bit equivalent of W,
- R1 to R4 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 addressing registers such as `INDFx`, `PREINCx`, `POSTDECx`, and `PLUSWx` are intensively used and also accessed in Access Bank for efficiency reasons.

6.4 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 location.

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,PREINCO ; 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
```

Please note 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 ;
}
```

6.5 Function calling conventions

All parameters are passed to functions on the software stack. They are stacked in reverse order (1st parameter pushed last)². 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`.

²No alignment is done during parameter passing, so a 16 bit local data can be located at odd or even address.

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 74 bytes. This pool can be decreased for devices with little memory size, or increased when the program uses large structures. See section 6.2 for details.

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

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

and the resulting code

```
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
```

6.6 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*

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

2. *Register value tracking*

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

3. *Constant folding*

Most constant subexpressions are computed by 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 simplified 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, instructions 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 optional³ 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%.

³See section 5.3

6.7 How to place 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 space⁴.

As a consequence, initialized data are 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 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 `__roms` keyword. However, it is perfectly usable and very efficient from both time and memory points of view.

6.7.1 Creating a block of data in ROM

The following macros allow to insert data in program space:

1. `ROM.TXT(text)`

Insert the specified 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, enclosed by double quotes (eg: `ROM.BYTES("0,1,2,0x33,5") ;`). Since the parameter of `ROM.BYTES` is processed by the assembler, you 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") ;`).

6.7.2 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

⁴The loader is not included if your program does not use statically initialized data.

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 understanding 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 `prodh1` 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 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 RInputs_()
{
    PREPARE_ROM_ACCESS ; READ_ROMBYTE ;
    while( prod1 )
    {
        putchar( prod1 ) ; 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 a set of 16 bit data from a ROM table, and displays them. In this example, the data are preceded by a word indicating the size of the table.

```
#define RInputwords(list) { RInputwords_() ; ROM_WORDS(list) ; }

void RInputwords_()
{
    int k ;
    PREPARE_ROM_ACCESS ; READ_ROMWORD ;
    for( k = prod1 ; k ; --k)
    {
        READ_ROMWORD ;
        outdec(prodh1) ; putchar(' ' ) ;
    }
    FINISH_ROM_ACCESS ;
}
```

This function is very simple to use:

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

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

6.7.3 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 receiving 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 guy !»
}

```

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( prod1 )
    {
        putchar( prod1 ) ; READ_ROMBYTE ;
    }
}

```

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

6.7.4 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
<code>ROMF_TXT</code>	<code>ROMF_i8_t</code>	<code>int8_t</code>	<code>ROMF_TXT(a,"hello\0")</code>
<code>ROMF_DATA8</code>	<code>ROMF_i8_t</code>	<code>int8_t</code>	<code>ROMF_DATA8(b,"-1,2,0xFF")</code>
<code>ROMF_DATA8U</code>	<code>ROMF_ui8_t</code>	<code>uint8_t</code>	<code>ROMF_DATA8U(c,"1,2,0xFF")</code>
<code>ROMF_DATA16</code>	<code>ROMF_i16_t</code>	<code>int16_t</code>	<code>ROMF_DATA16(d,"-1,0xFF34")</code>
<code>ROMF_DATA16U</code>	<code>ROMF_ui16_t</code>	<code>uint16_t</code>	<code>ROMF_DATA16U(e,"1,12300")</code>

7 Implemented features

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

7.1 Preprocessor

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

7.2 Data types

cpik V0.5x only allows integer data types, which can be either 8, 16 or 32 bit long:

- **char** (8 bit)
- **unsigned char** (8 bit)
- **int** (8 bit)
- **unsigned int** (8 bit)
- **long** (16 bit)
- **unsigned long** (16 bit)
- **long long** (32 bit)
- **unsigned long long** (32 bit)

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

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

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

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 guarantees 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 ints, 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	representable values	constant suffix
char	char_t	[-128..+127]	none
unsigned char	uchar_t	[0..255]	none
int	int8_t	[-128..+127]	none
unsigned int	uint8_t	[0..255]	none
long	int16_t	[-32768..32767]	L or l
unsigned long	uint16_t	[0..65535]	UL or ul
unsigned long	size_t	[0..65535]	UL or ul
long long	int32_t	[-2147483648..2147483647]	LL or ll
unsigned long long	uint32_t	[0..4294967295]	ULL or ull

7.3 Data structuration

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

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

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

structs 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.

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

7.4 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 ;
```

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.

7.5 Storage classes

Variables can be either automatically or statically allocated.

Local variables and function parameters are truly **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 algorithms⁵, and can be used to produce re-entrant code.

7.6 Static initialization

All statically allocated 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, initializer data type must match the type of the first member of the union.

```
/* the following is supported */
int k = 4 ; /* simple initialization */
char str[] = "hi !" ; /* array size inferred from initializer */
```

⁵However, remember that the hardware stack is limited to 31 levels.

```

int array[2][3] = {{1,2,3},{4,5,6}} ; /* initialization of array of arrays */
long z[10] = {0};      /* partial initialization, missing data replaced by 0s */
void f(char t)
{
    static char t='Z'-26 ; /* compile-time constant folding */
    /* ... */
}
int *pk = &k +1;        /* address calculation deferred to assembly stage */
void (*pf)(char) = &f ; /* idem */
struct xxx a = { 1, 55 } ; /* structure initialization */
struct xxx b = { 5 } ; /* partial structure initialization */

```

7.7 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).

7.8 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 located, using the `@address` extension. For example:

```

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

```

This is very handy to access Special Function Registers. `cpik` provides a set of header files containing the declaration of each SFR register, for each processor. These headers are automatically generated by a program that takes the informations from Microchip's `«.inc»` files. For example, here is the beginning of the `p18f242.h` header file:

```

#ifndef PROCESSOR
#define PROCESSOR
#define p18f242
//
// SFR and bits definitions for p18f242 device
// generated from p18f242.inc by inc2h utility
// Alain GIBAUD 2005, (alain.gibaud@free.fr)
//
// Register Files
//

```

```

unsigned int TOSU@0xffff ;
unsigned int TOSH@0xffe ;
unsigned int TOSL@0xffd ;
unsigned int STKPTR@0xffc ;
unsigned int PCLATU@0xffb ;
unsigned int PCLATH@0xffa ;
unsigned int PCL@0xff9 ;
...

```

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

7.9 Instructions

All C instructions are implemented.

The support for the **switch** instruction is recent and 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.

7.10 Operators

All C operators are implemented.

7.11 Extensions

7.11.1 Binary constants

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

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

7.11.2 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
```

7.11.3 Assembler code

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

```

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

```

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.

7.11.4 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 */
}
```

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

The keyword `__interrupt__` insure that

- `W`, `BSR`, `FSR1`, `FSR2`, `STATUS`, registers are properly saved and restored on the data stack. `FSR0` is not saved because it is the stack pointer itself.
- `retfie 0` is used as return instruction instead of `return 0`⁶.

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 exits⁷.

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

```
__interrupt__ void hi_pri_ISR()
{
    SAVE_REGS ;

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

    RESTORE_REGS ;
}
```

The `SAVE_REGS` and `RESTORE_REGS` macro are defined in the `<interrupt.h>` header, that should be included.

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.

⁶The `reftie 1` instruction is not used because it is explicetely mentionned as bogus by errata documents from Microchip.

⁷Caution: Never alter the `FSR0` stack pointer in a low level ISR if high level interrupts are enabled.

7.11.5 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 (`FSRO`) 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.

7.11.6 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.

All these macros are defined in `<interrupt.h>` header.

7.11.7 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.

8 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.

8.1 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 ;
```

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

Viewing two contiguous 8 bits SFRs as a single 16 bit register is 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 needs to write *hi byte first*, then *low byte*. Please see Microchip documentation for details.

Unfortunately, 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 recomand to perform reads with same care.

8.3 How to initialize EEPROM data

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

```
void your_function()
{
    /* Do the job this function is written for */
    return ;
    /*
```

```

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

}

```

Here, I initialize all EEPROM space of a 18F2525 device with nulls (the default value for an «erased» chip is all 0xFF).

8.4 Avoiding global namespace pollution increases modularity

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

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

```

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

```

could be replaced by an (anonymous or not) **struct**

```

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

```

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

8.5 Do not use uppercase only symbols

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

```

struct Z { int a,b ; } ;

```

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

```

struct 2 { int a,b ; } ;

```

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

8.6 How to write efficient code

The following simple rules help 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. Global data usage leads to smaller and faster code. Be careful, it also lead to lack of modularity and memory wasting. See previous section for modularity issue.
9. Implement 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.

9 Headers

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

9.1 `types.h`

This header allows to improve the portability of your code. It defines a set of ANSI-compatible integral types as described in section 7.2.

9.2 `macro.h`

This header provides a set of general purpose handy macros. In the following, `reg` denotes a 8 bit signed or unsigned integer, and `bit` a bit number ($0 \leq \text{bit} \leq 7$).

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

9.3 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

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

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

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

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

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.

10.1 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.

10.1.1 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 input. 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 input vector. The input vector is not initialized, so this function must be used prior any input.

`input_hook` is defined by `typedef char (*input_hook)() ;`

10.1.2 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 6.7 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 resources 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 resources 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 resources are limited because it leads to import unnecessary 32 bit code in your application.

10. `void printf(char *fmt)`

Mini implementation of the standard `printf()` function. Recognized conversion specifications are :

`%c %s %d %u %x %ld %lu %lx %lld %llu %llx`

Despite the fact `cpik` does not recognize the ANSI `<...>` syntax, this function supports a variable length argument list, and its prototype should be `<void printf(char *fmt,...)>`

Due to memory limitation, width specifiers (eg: `%03d`) are not supported.

As `printf()` supports 8, 16 and 32 bit integers, it imports an important part of the arithmetic library in your application, no matter it is used or not.

However, the 32 bit support will be disabled if the `stdio.c` file is compiled when the `NO_INT32_SUPPORT` macro is defined.

11. `void RFprintf(ROMF_i8_t fmt)`

Version of `printf()` receiving its format string thru a rom accessor. Please see section 6.7 for details about what is a rom accessor.

10.1.3 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.

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

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

This function is used to fill 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 scanf(char *fmt, addr [,addr ...])`

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

Like in `printf()`, the 32 bit integer support can be disabled by:
`#define NO_INT32_SUPPORT`

9. `int RFscanf(ROMF_i8_t fmt, addr [,addr ...])`

Version of `scanf()` receiving its format string thru a rom accessor. Please see section 6.7 for details about data located in ROM.

10.2 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 indefinitely block.

10.3 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 that 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

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

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 6.7 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 6.7 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 functionalities of the LCD display.

10.4 AD conversion

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

1. `void AD_init()`

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

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

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

10.5 EEPROM read/write

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

1. `void ee_init()`

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

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

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

3. `unsigned long ee_read16(unsigned long addr)`
Returns the 16 bit data located at address `addr`.
4. `void ee_write(unsigned long addr, unsigned int value)`
Writes 8 bit value at address `addr`.
5. `void ee_write16(unsigned long addr, unsigned long value)`
Writes 16 bit value at address `addr`.
6. `unsigned int ee_inc8(unsigned long addr)`
Increments 8 bit value located at address `addr`. Return the incremented value.
7. `void ee_inc16(unsigned long addr)`
Increment 16 bit value located at address `addr`.
8. `void ee_refresh()`
Performs EEPROM refresh as recommended 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.

10.6 Timer 0

Basic implementation of a slow real-time clock, with 1s and 1/10s ticks. This module can provide up to 8 independent 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 `TO_1S_FLAG` and `TO_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() & TO_1S_FLAG, new_flag ;

for( new_flag = old_flag ; ; )
{
    if( (new_flag = timer0_flags() & TO_1S_FLAG) != old_flag )
    {
        old_flag = new_flag ;
        // do something each second
    }
}
```

11 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 initialisation

(c) **UDATA**

This segment is dedicated to uninitialized data and will be 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>
<->
```

12 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.

13 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!

14 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 interested by bringing a new compiler to free software, you can contribute in many manners :

14.1 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.

14.2 Bug reports

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

14.3 Documentation

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

14.4 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.