

cpik
C compiler for PIC[®]-18 devices
Version 0.7.4,

Alain Gibaud
alain.gibaud@free.fr
(Documentation: rev C)

July 9, 2015

Contents

1	Introduction	5
2	What is new in version 0.7.4 ?	5
3	What is new in version 0.7.3 ?	6
4	What is new in version 0.7.2 ?	6
5	What is new in version 0.7.1 ?	7
6	What is new in version 0.6.0 ?	7
7	The «philosophy» behind cpik	9
8	A very special feature	9
9	Installation of cpik	10
9.1	Manual build	11
9.2	Build using qmake	11
9.3	cpik under Windows	11
10	Command syntax	12
10.1	Compilation	12
10.2	Link	12
10.3	Final assembly and jump optimizer	13
10.4	Pragmas	14
10.4.1	#pragma processor device_name	14
10.4.2	#pragma _CONFIGxy value (Deprecated, but still supported) . .	14
10.4.3	#pragma config keyword=symbol,keyword=symbol,...	14

10.4.4	#pragma _IDL0Cx value	15
10.4.5	#pragma saved_regs register,register,register,	15
10.4.6	#pragma fast_stack flag	15
10.4.7	#pragma noreturn	16
11	Support of C language on PIC-18 architecture	17
11.1	Stacks	17
11.2	Memory layout	17
11.3	Register usage	18
11.4	Computation model	18
11.5	Function calling conventions	19
11.6	Stack frame allocation and interrupts	20
11.7	Optimizations	20
11.8	Data in ROM	22
11.8.1	Creating a block of data in ROM	22
11.8.2	Passing immediate ROM data to a subroutine	22
11.8.3	Passing ROM data to a subroutine with a pointer to ROM	24
11.8.4	Accessing data in ROM with a ROM accessor	25
12	Features	26
12.1	Preprocessor	26
12.2	Data types	26
12.2.1	Numeric data types	26
12.2.2	ANSI types	26
12.2.3	void type	27
12.2.4	Pointers	27
12.2.5	Type safety	27
12.2.6	Cast and type promotion	27
12.2.7	const qualifier	27
12.3	Data structuration	28
12.3.1	Array	28
12.3.2	Struct and Union	28
12.4	Symbolic constants	28
12.5	Storage classes	29
12.6	Static data initialization	29
12.7	Non static data initialization	29
12.8	Scope control	30
12.9	Address allocation	30
12.10	Instructions	32

12.11	Operators	32
12.12	Extensions	32
12.12.1	Binary constants	32
12.12.2	Digit separator	32
12.12.3	Assembler code	32
12.12.4	Numeric constants in ROM (<code>__data8__</code> , <code>__data16__</code> , etc.)	33
12.12.5	Interrupt service routines	33
12.12.6	Why and how to write interruptible code	34
12.12.7	Disabling and enabling interrupts	34
12.12.8	Explicit bit fields	35
13	Hints and tips	37
13.1	Access to 16 bit SFR	37
13.2	Access to 16 bit SFR - second part of the story	37
13.3	How to initialize EEPROM data	37
13.4	Use <code>struct</code> to increase modularity	38
13.5	Do not use uppercase only symbols	38
13.6	How to write efficient code	39
14	Headers	40
14.1	<code>device/p18xxxxx.h</code>	40
14.2	<code>sys/types.h</code>	41
14.3	<code>macro.h</code>	41
14.4	<code>pin.h</code>	41
14.5	<code>stdarg.h</code>	42
14.6	<code>float.h</code>	42
14.7	<code>assert.h</code>	42
15	Libraries	43
15.1	standard IO library	43
15.1.1	IO redirection	43
15.1.2	output functions	43
15.1.3	Conversion specifiers supported by the <code>printf()</code> family	45
15.1.4	input	45
15.1.5	Conversion specifiers supported by the <code>scanf()</code> family	46
15.2	Standard math library	47
15.2.1	Trigonometric functions	47
15.2.2	Hyperbolic functions	47
15.2.3	Exponential, logarithmic and power functions	47
15.2.4	Nearest integer, absolute value, and remainder functions	48

15.3	Standard string library	48
15.4	sort library	48
15.5	Standard stdlib library	48
15.5.1	System	48
15.5.2	Character processing	49
15.5.3	Conversions to/from strings	49
15.6	rs232	49
15.7	LCD	49
15.8	AD conversion	51
15.9	EEPROM read/write	51
15.10	Timer 0	52
16	Source library structure	54
17	Needed software	56
18	Contributors	56
19	Credits	56
20	How to contribute to the cpik project ?	56
20.1	Feedbacks and suggestions	57
20.2	Bug reports	57
20.3	Documentation	57
20.4	Libraries	57
21	inc2h-v3	58
21.1	What is inc2h-v3 ?	58
21.2	How to build inc2h-v3 ?	58
21.3	Command summary	58

1 Introduction

`cpik` is an ANSI-C compiler for Microchip PIC 18 microcontrollers. It is a personal project developed on my spare time.

2 What is new in version 0.7.4 ?

1. Fix incompatibilities with Windows 7.

The `popen` POSIX standard function does not work properly under windows 7. This version contains a workaround that fixes this issue.

2. Suppress name clash between `cpik`-specific and device-specific symbols.

Some uppercase symbols defined in `.inc` device files were the same than symbols used by `cpik` and the `cpik` run-time library. This problem was very rare, but a few devices were affected. Because of new naming conventions now used by `cpik`, the name clashes are made impossible

Warning: the new naming convention affects the code generator, the `cpik` prolog code (`cpik.prolog`), the floating point library (`float.slb`), the run-time library (`rtl.slb`) and many headers. It makes any previously compiled code incompatible with new one, thus *the existing applications must be totally recompiled* (including the IO library) to become compatible with this version of `cpik`.

3. Several bug fixes concerning issues reported by Pascal Niklaus (Thanks Pascal !)

- (a) Incorrect parsing of character constants containing an hexadecimal value
- (b) Spurious error message when an expression was split by a line-feed at a "wrong" place.
- (c) Undetected programming errors concerning structures.
- (d) Missing W register value tracking causing a wrong code generation

4. Several instructions have been added in order to allow a better support of data tables located in ROM.

The non-standard instructions `__data8__`, `__data16__`, `__data32__`, `__dataf__` and `__datat__` allow to insert integer, floating point or text data in ROM. These new instructions are used by new macros in the `rom.h` header. The run-time support of these macros has been improved.

5. In case of error during a compilation, the `.slb` output file is not generated anymore.

This simple feature suggested by Pascal Niklaus avoids to build an application from a broken code. Moreover, the output of the preprocessor (`.c.c` file) is now removed after each compilation.

6. New `-k` option.

This option, related to the previous item asks the compiler to *keep* the output file generated by the preprocessor (`.c.c` file), and the object file (`.slb` file), even when the compilation fails. The default behavior of the compiler is to remove these files.

7. New string and sort libraries,

Directly written in assembly language by P. Niklaus, these libraries are provided as two `.slb` files. The first one contains 9 frequently used functions implemented in a compact and fast way. The second library contains 3 different sort algorithms.

8. Modified devices files,

The `#defined` symbols that were provided in the device files (ie: `p18xxxx.h`) to access register's field using the nonstandard bit-field extension of `cpik` have been removed. This decision was necessary because of name clashes in certain header files. For this reason, use the `XXXbits.MEMBER` syntax instead of the `XXX.MEMBER` one to refer to the `MEMBER` member

of the `XXX` register. Note that the former is the Microchip's standard way to use register. Because this decision may break existing code, the old definitions are still available in the `<device_obsolete>` directory in order to allow a smooth transition to the unique way to access register bit-fields.

3 What is new in version 0.7.3 ?

1. The `__asm__()` instruction has been improved.

In previous versions, assembler instructions generated from `__asm__()` were not able to change the stack pointer `FSR0`, because it prevented the compiler to access the local variables properly. This limitation is removed, allowing the programmer to insert a more sophisticated assembler code. See section 12.12.3 for details.

2. New pragma to specify the registers to save in an Interrupt Service Routine

This pragma makes the `SAVE_REGISTERS` and `RESTORE_REGISTERS` macros obsolete. Advantages of this new way to save registers are numerous. Full explanations in section 12.12.5.

3. Reservation of stack space

The code used for stack space allocation (used for function parameters or local variables) has been improved. This improvement was necessary for ISR that use more than 8 bytes of local variables. This topic is very important to understand if you use ISR routines. Please refer to the section 11.6 for details.

4. Bug fixes

Two bugs, discovered and analyzed by Jon Hilt (thanks!) have been fixed. One of them affected the compiler itself, and the second one the Run Time Library.

4 What is new in version 0.7.2 ?

1. Support for `CONFIG` directive

The old `<<_CONFIGxy value>>` assembler directive was deprecated for several years, but was still supported by `gpasm` with a warning message. Unfortunately, the `.inc` files coming with new PIC18 devices don't define the `CONFIGxy` symbols anymore, so the `_CONFIG` directive cannot be used for these devices.

This is why `cpik` now offers a new pragma that allows to use the new `CONFIG` directive :

```
#pragma config key=symbol,...
```

Please refer to the section 10.4.3 for details.

Note that the new pragma is also supported by `pikdev` 1.4, the new version of my IDE.

2. New supported devices

`cpik` now supports all the PIC18 devices that are supported by the new `gputils` 1.0.0 suite. It means that many new device-headers are provided.

3. Support for devices with SFR out of access bank

Some (rare) new devices contain Special Function Registers that cannot be reached in access bank¹. Such devices did not exist when `cpik` has been designed, so the generated code was wrong for them. The fix for this issue has been to change the `firstsfr` pragma in device-header files. A new version of the `inc2h` utility has been written for this purpose. See section 21 for details.

4. Bug fix in structure support

The code generated for some access to structure members was wrong, due to a missing pair of parentheses.

¹for example: 18F25K22

5. Other minor bug fixes

Thank you to Jon Hilt who has reported the two previous problems, and fixed one of them.

5 What is new in version 0.7.1 ?

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 slightly modified.

6. Minor bug fixes

6 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 NAN and INF are not supported. This design choice has been made to reduce the size of code. The core floating-point library² has been carefully 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 specifications (`%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.

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.

²Basic operators and conversion routines.

7. New command line option

The traditional «-D» switch allows to pass macro definitions to the preprocessor.

8. Minor bug fixes

7 The «philosophy» behind cpik

My idea was to develop a compiler based on the ANSI (C89) specification. This is a huge work for a developer that have many other activities, so I had to decide what is important and what is not. My main idea was 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 `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.

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).

8 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 `.s1b` extension. This file can be processed by an assembler (such as `mpasm` or `gpasm`) but contains special comments that are intended to be used as *directives* by the 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 (In contrast, most linkers work from the object code generated by an assembler).

The file generated by the linker is easy to verify manually, 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 has been used to support the very first program³ 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.

³Believe it or not, this program (a simple `for` loop) worked successfully at the first execution. To be honest, this execution has been preceded by many manual check of the generated code.

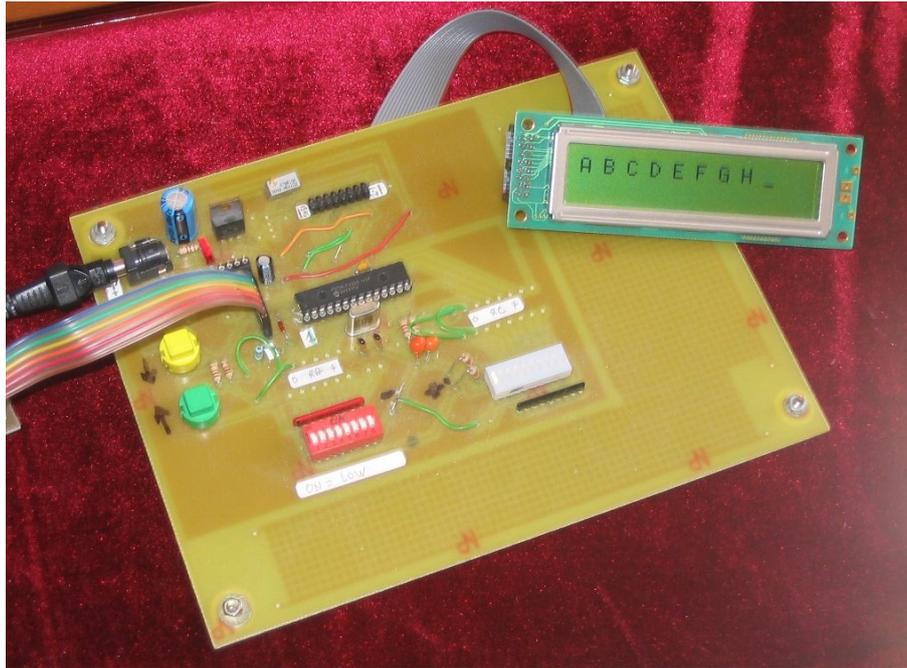


Figure 1: Result of the very first program compiled by `cpik` ever executed

- the final executable code (ie: `<.hex>` file) can be generated by a very simple assembler without any advanced feature (The target assembler is currently `gasm` 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*.
- the linking process is globally very simple and does not increase significantly 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 that depend on the location of entities in memory can be deferred to assembly stage.

In fact, the `<source 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.

9 Installation of `cpik`

Notes:

- In the following, `<VERSION>` is the version number of `cpik` (eg: 0.6.0), and `<version>` is a 3-digit string corresponding to this version number (eg: 060 for V0.6.0).
- The `cpik` archive is supposed to be extracted.

9.1 Manual build

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

1. Compile each `.cpp` file separately, 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 `<VERSION>` (use here the proper version number) to `/usr/share/cpik`

9.2 Build using qmake

`qmake` is the build tool that comes with Qt. You can download the complete Qt toolkit at the address <http://qt-project.org/>, 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`. Note 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 when it is not absolutely needed.

9.3 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 `x.y.z` to `C:\cpik` (*replace here x.y.z by the current version number*)
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.

10 Command syntax

The `cpik` command can be used as a compiler or a linker, exactly like the `gcc` front end. 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.

10.1 Compilation

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

`-v` : prints the version number, then exits immediately, regardless the other options.

`-k` : This option asks the compiler to keep the output file generated by the preprocessor (`.c.c` file), and the object file (`.slb` file), even when the compilation fails. The default behavior of the compiler is to remove these files. This option is for debugging purpose only and do not need to be used in standard condition. Note that the object file generated by `cpik` is likely to be corrupted in case of compilation error.

`-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 extension-less 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 traditional behavior 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 checked, 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
<code>-d1</code>	print unoptimized intermediate code as comment in <code>.slb</code> file
<code>-d2</code>	print peep hole optimized intermediate code as comment in <code>.slb</code> file
<code>-d4</code>	print symbol tables with entities names and types
<code>-d8</code>	print internal expression trees before optimizations, without type annotation
<code>-d16</code>	print internal expression trees before optimizations, with type annotations
<code>-d32</code>	print internal expression trees after optimizations, without type annotation
<code>-d64</code>	print internal expression trees after optimizations, with type annotations

Notice that the `-d` option is never useful for normal operations with `cpik`. This option generates outputs that 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.

10.2 Link

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

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

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

`-L path` : specifies the path to libraries (`.slb`) files. This option follows the traditional 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.

10.3 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 contain* 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`.

10.4 Pragmas

Using pragmas is a standard way to do non-standard things in C language. Pragmas are often used to switch on/off a special feature of the compiler, in a more handy way than the command line. For this reason, a pragma can be seen as an extension of the command line.

10.4.1 `#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.

10.4.2 `#pragma _CONFIGxy value` (Deprecated, but still supported)

Note: This pragma is deprecated.

Please use instead the `<config>` pragma, as described in the next section.

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.

10.4.3 `#pragma config keyword=symbol,keyword=symbol,..`

This pragma allows to generate a standard `CONFIG` directive in the output code. It replaces the previous `_CONFIGxy` directive. I recommend to switch to the new pragma, because some devices cannot be configured with the old `_CONFIGxy` directive. Moreover, the old directive provokes a warning from `gpasm`.

for example,

```
#pragma _CONFIG2L 0x0A
#pragma _CONFIG2H 0x10
```

should be replaced with

```
#pragma config WDTPS=256,WDT=OFF,BORV=27,BOR=ON,PWRT=ON
```

You can use any number of `config` pragmas. For example, the previous pragma could be replaced with

```
#pragma config WDTPS=256,WDT=OFF
#pragma config BORV=27,BOR=ON,PWRT=ON
```

This pragma can be automatically generated by the config word generator of `PiKdev V1.4`. However, you can write the `config` parameters by yourself. In this case, you will have to refer to the device documentation. Another option is to read the `.inc` file specific to your device to know the available keywords and constants names.

10.4.4 #pragma _IDL0Cx 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 `_IDL0Cx` pragma with different values will have an unpredictable behavior.

10.4.5 #pragma saved_regs register,register,register, ..

Since version 0.7.3, the `SAVE_REGS` and `RESTORE_REGS` macro are suppressed. These macros were used to save and restore the data modified by an Interrupt Service Routine on to the stack. However, this kind of context saving had two disadvantages:

1. Data was saved in the stack frame of the function, so the available space on the stack was reduced,
2. because data was saved *after* the local variables, these variables were not properly managed by the compiler. It was indeed a very bad feature, because the code of ISRs was obliged to use only global variables.

With the new pragma, registers are saved before the stack frame, so the two previous defaults disappear.

The following `saved_regs` pragma is provided in the `<interrupt.h>` header:

```
#pragma saved_regs _r0,_r0+1,_r1,_r1+1,_r2,_r2+1,_r3,_r3+1,PRODL,PRODH
```

You can redefine this pragma just before an ISR routine if you are not happy with the standard registers list. Note that doing such a redefinition does not expand the previous list of registers, but creates a new list. It means that you can easily turn off all the registers savings with:

```
#pragma saved_regs
```

Also remember that the registers used by the floating point routines are not saved if you use the standard registers list, so if an ISR performs floating point maths, you will have to use this pragma⁴.

The *register* specification can be any expression or address that refers to a valid location in memory because data is pushed with a `movff` instruction, that can access any page. However, never use this pragma to save any interrupt control registers (`INTCONx` or `PIRx`) because, as stated in the documentation, they cannot be manipulated with the `movff` instruction.

10.4.6 #pragma fast_stack flag

This pragma allows to control the technique used for stack allocation in functions. The default technique is to use a fast stack allocation. However, this technique cannot be used *when the function is called from an Interrupt Service Routine (ISR)*, so it must be manually disabled in this specific case⁵ (ie: `#pragma fast_stack 0`). The standard technique must be re-enabled after the concerned function (ie: `#pragma fast_stack 1`).

Note that:

⁴However, it is generally not a good idea to perform complex calculations in an interrupt service routine.

⁵This feature is due to the fact that it is impossible for the compiler to anticipate whether a function is invoked from an ISR or not.

- The standard stack allocation technique corresponds to `#pragma fast_stack 1`
- this pragma has not effect (and is not needed) for functions that use less than 8 bytes of local data
- this pragma has not effect (and is not needed) for ISRs

10.4.7 `#pragma noreturn`

This pragma allows to suppress the generation of the `return 0` assembly instruction that normally terminates any function. It is only useful when a function contains assembly language code inserted via the `__asm__` nonstandard instruction that make the standard `return` useless. The same effect can be obtained with the `__noreturn__`; instruction.

Note that the effect of this pragma is limited to the next function to be compiled.

11 Support of C language on PIC-18 architecture

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. Pointers generally point to data space, but pointers to function point 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 addresses 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 section 12.12.5 for details). Thanks to auto-incremented and indirect addressing modes, this design leads to efficient code.

The memory model is flat and covers the totality of program/data spaces. There is no banks, "small" stacks, "far" pointers or other tricky ways to save memory but to confuse everybody.

11.1 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 an overflow 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,PREINCO` instruction. Symetrically, a `movxx POSTDECO,dest` is used to pop the data back.

11.2 Memory layout

The current memory layout used by **cpik** is the following⁶:

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	<code>_r0, _r1, _r2, _r3, _r4, _r5</code> pseudo-registers
<code>C18_errno</code>	[1->1]	1	reserved by libraries
IT mask	[0->0]	1	reserved by RTL

⁶This layout has changed in 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 than 20 bytes, the *Register* zone can be extended by the *Scratch* zone. The size of this area can be adjusted 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 that `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 (ie: `malloc()`, `free()` and so on). However such a zone could be obviously implemented at the end of physical memory, and must expand from top (high addresses) to bottom.

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

11.4 Computation model

- *operators with 2 operands* are executed with the first operand on the top of stack, and the second one in `W` (8 bit) or `_r0` (16 bit) or `_r0-_r1` (32 bit). The result replaces the first 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,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
```

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 ;
}
```

11.5 Function calling conventions

All parameters are passed to functions on the software stack. They are stacked in reverse order (1st parameter pushed last)⁷. 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`⁸.

8 bit results are returned in `_r0L` register, 16 bit results are returned in `_r0` register and 32 bit results 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 adjusted to fit you needs or the hardware requirements. See section 11.2 for details.

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

```
void caller()
{
    int res, k ;
```

⁷No alignment is done during parameter passing, so data can be located at odd or even address.

⁸This feature will change in future versions.

```

    res = h(k, 25) ;
}

```

and the resulting code

```

C18_caller
    movf PREINCO,F,0      ; reserve stack space
    movf PREINCO,F,0      ; for k and res
    movlw 25
    movwf PREINCO,0      ; push param 25 onto the stack
    movlw -1
    movff PLUSW0,PREINCO ; push parameter k
    ICALL C18_h           ; call h()
    movf POSTDECO,F,0    ; (partially) clean stack
    movff _r0,INDFO      ; move result to temporary
    movlw -1             ; pop result to res and
    movff POSTDECO,PLUSW0 ; finish to clean stack
    movf POSTDECO,F,0
    movf POSTDECO,F,0    ; (discard local variables)
    return 0

```

11.6 Stack frame allocation and interrupts

A *stack frame* is a memory block that is reserved on the stack by any function that uses local variables. This allocation is typically done at the very beginning of the user code of the function by simply pushing any value on to the stack. For example, a simple `<<clrf PREINCO,0>>` instruction can be used to reserve space for one `char` variable.

When more data has to be pushed, the code for the stack frame allocation can become very long and slow because one byte is reserved at a time. This is why `cpik` generates in this case a code that just alter the stack pointer (`FSR0`) by adding to it the number of bytes needed. Unfortunately, the stack pointer is composed of 2 separate 8 bit registers, so doing the operation needs 4 instructions. Here, the annoying things begin. Lets suppose that an interruption occurs just during this calculation: an ISR is triggered, but is executed with a `<<half-cooked>>` stack pointer that do not point to a valid memory place.

In order to avoid such a situation, interruptions must be masked prior the calculation, then restored to their previous state after it. By this way, the change of the stack pointer is made atomic (ie: non interruptible) and no problem can occur.

However, this technique cannot be used in an Interrupt Service Routine for very technical reasons, so ISRs are automatically compiled with a specific stack frame allocation technique.

This restriction also applies to routines called by ISR. Because the compiler cannot anticipate whether such a routine will be called by an ISR or not, this routine cannot be automatically compiled with the proper stack management technique. This situation is very rare, but you can avoid it by disabling the standard stack allocation for a specific routine. See section 10.4.6 for details about the `fast_stack` pragma.

11.7 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 is compared against a library of possible simplifications or transformations. Then, the simplified code is simplified again, and so on. This kind of optimization may lead to 25% to 30% 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, consider 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 optional⁹ 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%.

⁹See section 10.3

11.8 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 keep initialized data¹⁰ because it can only be stored in program space.

In order to provide access to initialized data, `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 space¹¹. 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 (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. This support is experimental because it prepares a definitive implementation based on a `__rom` keyword. However, it is perfectly usable and very efficient from both time and memory points of view.

11.8.1 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") ;`).

11.8.2 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") ;
```

¹⁰such as literals like `"hello !"`.

¹¹The loader is not included if your program does not use statically initialized data.

A better way to do that is to use a 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.s1b`, 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 reads from ROM 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 the processor to execute data instead of machine code and will crash the program.

Here is another example: this routine fetches 16 bit data from a ROM table, and displays it. In this example, the data is preceded 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(prodh1) ; 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 is a sequence of words (not a string of chars). Obviously, it is forbidden to nest functions that access ROM memory.

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

In the previous section, each block of data in ROM was anonymous and the return address of the subroutine was used as a pointer to this block.

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. The `void lcd_Rputs(ROMptr) ;` (from the LCD library) is an example of such a function.

```

void f()
{
    lcd_Rputs(hello) ; // displays «hello guys !»
}

```

As previously, it is easy to write code that receives a `ROMptr` 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.

11.8.4 Accessing data in ROM with a ROM accessor

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

Eight macros are available to declare a ROM accessor. For example, the `ROMF_TXT` macro both 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_txt`. The `f` stands for function, because accessors are technically functions receiving an `unsigned int`.

Here is another flavor of the usual `puts()` function that receive a pointer to a `romf_txt` :

```
void RFputs(romf_txt *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_txt</code>	<code>int8_t</code>	<code>ROMF_TXT(a, "hello\0")</code>
<code>ROMF_DATA8</code>	<code>romf_data8</code>	<code>int8_t</code>	<code>ROMF_DATA8(b, -1,2,0xFF)</code>
<code>ROMF_DATA8U</code>	<code>romf_data8u</code>	<code>uint8_t</code>	<code>ROMF_DATA8U(c, 1,2,0xFF)</code>
<code>ROMF_DATA16</code>	<code>romf_data16</code>	<code>int16_t</code>	<code>ROMF_DATA16(d, -1,0xFF34)</code>
<code>ROMF_DATA16U</code>	<code>romf_data16u</code>	<code>uint16_t</code>	<code>ROMF_DATA16U(e, 1,12300)</code>
<code>ROMF_DATA32</code>	<code>romf_data32</code>	<code>int32_t</code>	<code>ROMF_DATA32(f, -1,0xdeadbeef)</code>
<code>ROMF_DATA32U</code>	<code>romf_data32u</code>	<code>uint32_t</code>	<code>ROMF_DATA32U(g, 22, 34)</code>
<code>ROMF_DATAF</code>	<code>romf_dataflt</code>	<code>float</code>	<code>ROMF_DATAF(h, 3.14,-22.0/9)</code>

Since V0.7.4, these macros use the `__data....` instructions (see section 12.12.4), so they can receive any valid C constant expression.

12 Features

12.1 Preprocessor

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

12.2 Data types

12.2.1 Numeric data types

cpik supports the following numeric data types:

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

12.2.2 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 performances from the processor when an `int` is used. PIC-18 devices are based on 8 bit data registers, so I suppose that an `int` should be coded on 8 bits 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
<code>char</code>	<code>char_t</code>
<code>unsigned char</code>	<code>uchar_t</code>
<code>int</code>	<code>int8_t</code>
<code>unsigned int</code>	<code>uint8_t</code>
<code>long</code>	<code>int16_t</code>
<code>unsigned long</code>	<code>uint16_t</code>
<code>unsigned long</code>	<code>size_t</code>
<code>long long</code>	<code>int32_t</code>
<code>unsigned long long</code>	<code>uint32_t</code>

12.2.3 void type

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

12.2.4 Pointers

Any valid pointer can be declared.

12.2.5 Type safety

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

12.2.6 Cast and type promotion

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

12.2.7 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 consistent 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
}
```

12.3 Data structuration

12.3.1 Array

`cpik` currently supports arrays in any valid way.

12.3.2 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: `structs` can be passed to functions as parameters and can be returned by functions.

`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 members¹².

`structs` cannot be larger than 128 bytes¹³, but I suppose it is not a terrible limitation for a 8 bit microcontroller.

Members of `structs` 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 too long to be inserted in the first byte.

Bits are specified from low to high (so, in the `struct XXX`, `a` is the low nibble)

Please note that `unsigned` bit fields are much more efficient than their signed counterpart because they don't need to be sign extended when they are converted to `int`.

12.4 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 ;
```

¹²However, this feature does not apply to structures composed of bit-fields, as explained below.

¹³This 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 ints, so the specified values must range from -128 to 255 (255 and -1 being in fact the same constant).

12.5 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 algorithms¹⁴, and can be used to produce re-entrant code.

12.6 Static data initialization

All *statically allocated*¹⁵ 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 initialization, 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 */
```

12.7 Non static data initialization

Automatic scalar variables can be initialized. However automatic array/structure/union variables cannot be initialized: this behavior corresponds to the old K&R standard, but is a restriction from the ANSI standard.

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

¹⁴However, remember that the hardware stack is limited to 31 levels.

¹⁵Local variable are not statically allocated, unless they are declared with `static`.

```

    /* ... */
}

```

12.8 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).

12.9 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 gets the information 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 responsibility to insure that the specified location really corresponds to usable memory.

12.10 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 both high speed and compact code, so it perfectly fits the spirit of the **switch** statement.

12.11 Operators

All the operators of the C language are implemented.

12.12 Extensions

12.12.1 Binary constants

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

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

12.12.2 Digit separator

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

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

12.12.3 Assembler code

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

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

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

Prior version 0.7.3, the inserted code had to leave the stack pointer (FSR0) unchanged. The violation of this rule made the compiler unable to access the local variables properly.

However, this limitation is now removed: it is possible to specify that the inserted code changes the stack pointer, so the compiler is not fooled anymore. The following code shows how to use `__asm__()` with an extra parameter that gives the number of byte pushed on to the stack.

```
int g()
{
    int a,b,c ;
    // ...
    __asm__("tmovff PRODL,PREINCO", 1) ; // one byte pushed on to the stack
    __asm__("tmovff PRODH,PREINCO", 1) ; // one more byte pushed on to the stack
    a = b+c ; // a, b and c are properly accessed
    __asm__("tmovff POSTDECO,PRODH" ) ;
    __asm__("tmovff POSTDECO,PRODL", -2)) ; // two bytes popped
    return a ;
}
```

This feature allows the user to insert a more sophisticated assembler code in C code.

12.12.4 Numeric constants in ROM (`__data8__`, `__data16__`, etc.)

The instructions `__data8__`, `__data16__`, `__data32__`, `__dataf__` and `__datat__` allow to insert 8 bit, 16 bit, 32 bit, floating point or text data in ROM. Inserting 8 bit or 16 bit data was previously possible via the `__asm__` instruction, but inserting 32 bit or floating point data was very tricky. Another difference with the previous solution is that the data list is now processed a C level, not at assembly level. As a consequence, any valid C constant expression can be used to specify the data value, as below:

```
__data16__(4) ; // size of the table, as a 16 bit integer
__dataf__(2.718281, 3.141592/180., 1/3.0E7, 55 ) ; // constant values
```

Note 1: The last value is an `int` in the previous example, but it will be automatically converted to float.

12.12.5 Interrupt service routines

The PIC18 devices use two kind of interrupts: low priority interrupts, and high priority interrupts. The corresponding Interrupt Service Routines (ISR) respectively start at addresses `0x18` and `0x8`. `cpik` automatically put a branch instruction at these addresses, so the code of the ISRs can be located anywhere in memory. Two empty interrupt service routines are provided by the run-time library¹⁶.

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

A user who plan to use interrupts must provide a specific interrupt service routine that can be written in C, using the (non-standard) `__interrupt__` keyword as following:

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

¹⁶The run time library is the file `/usr/share/cpic/<version>/lib/rtl.slb`

The used-defined routine will shadow the default one, because user libraries are scanned before `rtl.slb`.

The role of the `__interrupt__` keyword is to insure that

- `W`, `BSR`, `FSR1`, `FSR2`, `STATUS`, registers are properly saved and restored on the data stack¹⁷.
- `retfie 0` is used as return instruction instead of `return 0`¹⁸.

The body of an ISR routine can be written in pure assembly language, using the `__asm__` directive. In this case, all previously mentioned registers can be freely altered, as long as `FSR0` (the software stack pointer) is not altered when the ISR exits.

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 will be saved if a proper pragma `saved_regs` has been seen by the compiler before the source code of the ISR.

A standard `saved_regs` pragma is provided in the `<interrupt.h>` header, so, generally, nothing special must be done if `interrupt.h` is included. However, I recommend to verify that the registers specified by the `saved_regs` pragma match the registers that are actually used in the ISR.

See section 10.4.5 about the `saved_regs` pragma for details.

Note: the `SAVE_REGS` and `RESTORE_REGS` macro that were defined in the `<interrupt.h>` header prior version 0.7.3 *have been suppressed and cannot be used anymore*. Any call of these macros should be removed from your source code and the equivalent `saved_regs` pragma should be inserted before the concerned ISR.

12.12.6 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.

12.12.7 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 consistent 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.

¹⁷`FSR0` is not saved because it is the stack pointer itself.

¹⁸The `retfie 1` instruction is not used because it is explicitly mentioned as bogus by errata documents from Microchip.

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 were enabled or not. Of course, never change the interrupt status in a critical section, and never enter a critical section from a critical section.

Caution: never use one of these macros in an Interrupt Service Routine because it is likely to cause an infinite recursive call of the ISR. This limitation is caused by a flaw in the design of the PIC18 architecture. All these macros are defined in `<interrupt.h>` header.

12.12.8 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.

13 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.

13.1 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 ;
```

13.2 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 forces 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 recommend to perform reads with same care.

13.3 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 explicitly 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.

13.4 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 (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` ;

13.5 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.

13.6 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. 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.

14 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.

14.1 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.

The following example illustrates how the devices' registers are declared.

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

// This file has been automatically generated from Microchip's "p18f2525.inc" file.
// with the inc2h-v3 utility.           Please do not edit.
// Do not use with cpik versions < V0.7. Please report problems to the author.
// (C) Alain Gibaud 2012-2013   (alain.gibaud@free.fr)

// Note; All SFRs are reachable via access bank
#pragma firstsfr 0xf80
// ...
// ...
// -----
//          T3CON
// -----
unsigned int T3CON@0xfb1 ;
union
{

struct
{
    unsigned int
    TMR3ON : 1 ,
    TMR3CS : 1 ,
    NOT_T3SYNC : 1 ,
    T3CCP1 : 1 ,
    T3CKPS0 : 1 ,
    T3CKPS1 : 1 ,
    T3CCP2 : 1 ,
    RD16 : 1 ;
} ;

struct
{
    unsigned int
    : 2,
    T3SYNC : 1 ;
} ;

// The following are aliases ..
struct
```

```

{
    unsigned int
    : 4,
    _T3CKPS : 2 ;
} ;

} T3CONbits@0xfb1 ;

```

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 because the _T3CKPS member name is an alias for the T3CKPS0/T3CKPS1 group of bits.

```

T3CONbits.T3CKPS0 = 0 ; T3CONbits.T3CKPS1 = 1 ; // method 1 : individual bits
T3CONbits._T3CKPS = 0b10 ; // method 2 : bit field

```

Notice that the names T3CKPS0, T3CKPS1, etc have been chosen to be compatible with the member names used by Microchip, but obviously, they sound like macro names, despite the fact they are not.

14.2 sys/types.h

This header helps you to improve the portability of your code. It defines a set of ANSI-compatible integral types as described in section 12.2.2, but is not part of the ANSI standard.

14.3 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 ($0 \leq \text{bit} \leq 7$). **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.

14.4 pin.h

pin.h provides a set of very handy macros that allow to use the bit of each port thru symbolic names. This very ingenious 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

14.5 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.

14.6 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.

14.7 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`.

15 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`, `string.slb`, `sort.slb` and `lcd.slb` (which are directly written in assembly language) each library (for example `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.

15.1 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.

15.1.1 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)() ;`

15.1.2 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 11.8 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 resource 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. `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 15.1.3
This function return the number of characters printed.
12. `int RFprintf(ROMF_i8_t fmt)`
Version of `printf()` receiving its format string through a rom accessor. Please see section 11.8 for details about what is a rom accessor.
See more information about conversion specifiers in section 15.1.3
This function return the number of characters printed.

15.1.3 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 explicitly enabled to become active. If you need this support, just `#define` the macros `INT32_IO` and/or `FLOAT_IO` when `stdio.c` is compiled¹⁹.

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

15.1.4 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.

¹⁹You 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 11.8 for details about data located in ROM.

This function returns the number of conversion specifiers successfully processed.

15.1.5 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 compiled²⁰.

²⁰You can either edit the source code, or just use `-DINT32_IO -DFLOAT_IO` at `cpik` invocation.

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

15.2 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.

15.2.1 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);`

15.2.2 Hyperbolic functions

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

15.2.3 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) ;`

15.2.4 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);`

15.3 Standard string library

This library is written in assembly language by P. Niklaus, and is very fast and compact. It contains the following functions, declared in `string.h`.

1. `strlen`
2. `strcat`
3. `strcpy`
4. `strcmp`
5. `strcasecmp`
6. `strncmp`
7. `strncasecmp`
8. `memcpy`
9. `memmove`
10. `xmemshift`

15.4 sort library

This library is written in assembly language by P. Niklaus, and provides 3 different sort functions, declared in `sort.h`.

1. `bssort`
2. `cssort`
3. `ssort`

15.5 Standard stdlib library

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

15.5.1 System

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

15.5.2 Character processing

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

15.5.3 Conversions to/from strings

1. `char *ftoa (float x, char * str, int prec, int format);`
2. `float strtod(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) ;`

15.6 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.

15.7 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. The following function allows to compute the cursor position for most classic flavors of LCD displays. Note that this function is not part of the library.

```

// number of columns (16, 20 or 24)
#define LCD_COLS 20

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

```

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's element corresponding 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 11.8 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 11.8 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.

15.8 AD conversion

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

15.9 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.

15.10 Timer 0

Basic implementation of a slow real-time clock, with $1s$ and $\frac{1}{10}s$ 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 $\frac{1}{10}s$, 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 $\frac{1}{10}s$ 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()
{
    if (INTCON & (1 << TMR0IF)) // does interrupt come from timer0 ?
    {
        timer0_ISR() ; // yes, call interrupt handling code
    }
}

```
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)`
Explicitly 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
    }
}

```

16 Source library structure

A source library is an assembly language source file, with special comments interpreted by the **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,PREINCO
  movlw 2
  ICALL mul8u
  movff POSTDEC0,_r0
; }
L18_main_x2_0
  return 0
; <?mul8u>
; <->
```

17 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.

18 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,
- Optimization of the run-time support for shifts,
- Dead-code elimination in the case of tests or loops with a constant condition,
- Optimization of the static data initialization,
- Optimization of tests in some special situations,
- Post-compilation branch optimizer that eliminates far jumps whenever possible.

Thank you Josef!

- **Pascal A. Niklaus**

- Bugs report and various suggestions,
- Assembly language `string` and `sort` libraries.

Thank you Pascal!

19 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 (fully rewritten in PIC18 assembler by Alain Gibaud).

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

20.1 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.

20.2 Bug reports

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

20.3 Documentation

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

20.4 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 PIC peripheral (`timer`, `AN conversion`, `USB` etc.) needs a library.

21 inc2h-v3

21.1 What is inc2h-v3 ?

`inc2h-v3` is an utility that allows to make device-specific C headers from `<.inc>` files, and `<.lkr>` files. These files describe the SFRs available for each device flavor, together with the bits or bit fields they contain. The `<.inc>` and `<.lkr>` 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.

21.2 How to build inc2h-v3 ?

1. Unpack the `inc2h-v3.tar.gz` archive

```
tar xzvf inc2h-v3.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-v3 bf.cpp processor.cpp inc2h-v3.cpp
```

21.3 Command summary

You can use `inc2h-v3` in two ways.

1. `inc2h-v3 -locate`

`inc2h-v3` tries to discover where the `<.inc>` files are located on your hard disk, and displays this (or these) location(s). This command is useful if you don't know where the `gputil` tool suite has been installed. Basically, this option just uses the `locate` Unix command, so it may fail.

2. `inc2h-v3 [options] -all <directory>`

`inc2h-v3` 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.

Note:

- This command is intended to be used with the files provided by the `gputils-1.x.y` package.
- It will not work if the `<inc>` and the `<lkr>` directories are not located in the same directory.

The options allow to control what the generated files will contain:

- **-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 behavior is to create the two kind of information in the same file. Using only one of this options has the (small) advantage of keeping the resulting files relatively small.

In order to understand the various ways to access the SFRs with `cpik`, I recommend to read one of the `<.h>` files generated by `inc2h-v3`.