cpik: a C compiler for pic 18Fxxx devices

preliminary documentation

Alain Gibaud

Version 0.1

21st June 2005

Contents

1 Introduction

cpik is a C compiler which generates code for Microchip pic 18 microcontrollers. The language recognized by this compiler is (for the moment) a subset of ANSI C language, but contains extensions specific to microcontroller domain.

This compiler is in alpha stage, so any feedbacks concerning bugs, feature requests or criticisms can be addressed to Alain Gibaud (`alain.gibaud@free.fr`).

2 Specific features

cpik works on an unsusual way: unlike other compilers, it does not produce object code but *source libraries* (`.slb`) files.

A source library is simply a source file containing assembly language. This file can be processed by an assembler (such as `mpasm` or `gpasm`) but contains special comments which are used as *directives* by an ad-hoc linker. This linker is included in cpik executable, so the same command can be used for compilation and link tasks.

This approach have several advantages

- Any source library is simply a text file so it can be manually builded using assembly language and your favourite text editor,
- source libraries do not depend on any object/library format, and are really very simple to build. Please report to the section *"source library structure"* for details about `.slb` files,
- final executable code can be generated by a very simple assembler without any advanced feature (in fact, the current target assembler is currently `gpasm` running in absolute mode)
- any output from the compiler is potentially a library, so there is no more differences between *object files* and *libraries*,
- linking process is globally very simple.

I suppose there is drawbacks too, and I will make them public as soon as you will report them.

The important point is that the linker *works at assembly source code level*: it picks needed program sections from source libraries and copies them in a single output file. This file is easy to manually verify, and I suppose (and hope) that advanced users will examine it and will report feedbacks about generated code.

3 Command syntax

3.1 Compilation

```
cpik -c [-o output_file] [-I path] [-p device] input_file
```

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

`-I path` : specify the path to include (`.h`) files. This option follows the traditionnal behaviour of Unix C compilers. You can specify any number of include path, and they will be searched in the order of `-I` options. The default include path is `/usr/share/cpik/include/` and is alway searched in last position.

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

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

3.2 Link

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

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

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

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

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

4 Processor model

cpik generates code for pic-18 processors running in legacy (ie: non-ehanced) mode. Pic-18 is fundamentally a 8 bit processor with 16 bit pointers and distinct program/data address spaces. From the C programmer point of view, up to 64K bytes of program space and 64K bytes of data space are potentially available. Pointer generally points to data space, but pointer to function points to program space. Please note that *constant data* located in program space is not implemented yet.

4.1 Memory usage

cpik-generated code uses two stack:

- The *hardware return stack* (32 levels):

  This stack might be insufficient for recursive intensive applications, but is quite suffcient for "normal" ones.

- A *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 register `FSRx`, and indirect, auto-incremented, and indexed addressing mode, the stack usage is quite efficient. `FSR0` is used as the software stack pointer.

  cpik reserves memory from 0x0 to 0xF for run time support.

  Bytes above this little region are used by static variables, and data stack is located above them.

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

4.2 Register usage

cpik potentially uses all registers located in SFR region, plus 4 16 bit pseudo registers named R0,R1,R2,R3. Each Rx register can be splitted in RxH and RxL. These registers are located in page 0, and are accessed via address bank (a=0).

- `W` is a general purpose register
- `R0` is the 16 bit equivalent of W,
- `R1` to `R3` are used by Run Time Library,
- `FSR0` is the software stack pointer,
- `FSR1` is a general purpose address register,
- `FSR2` is not used yet,

4.3 Computing model

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

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

Due to hardware limitation, the total amount of local non static data used by a function can't exceed 127 bytes. This point covers parameters, local variables, and temporaries. Excepted for very complex expressions, temporaries never exceed a few bytes, so, as a rule of thumb, 110 bytes are always available.

4.4 Function calling conventions

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

These two characteristics are common for C code and needed by variable length function parameters list feature of C language.

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

5 Implemented features

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

5.1 Preprocessor

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

5.2 Data types

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

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

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

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

The `typedef` keyword is not supported yet.

5.3 Data structure

cpik currently supports arrays, with any number of subscripts.

`struct` is not supported, but will be in the future. For the moment, `struct` can be emulated by ad-hoc macros.

```
struct X
{
  char a ;
  long b ;
} ;
```

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

```
struct X x ;
x.a = 0 ;
x.b = 55 ;
```

Could by replaced by:

```
#define a_(p) *((char *)(p))
#define b_(p) *((long *)((p)+1))

char x[3] ;
a_(x) = 0 ;
b_(x) = 55 ;
```

Due to constant folding optimization, the resulting code is very close to the code that could have been generated from a program using `struct`.

5.4 Storage classes

Variables can be either automatically or statically allocated.

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

All statically allocated variables or arrays can be statically initialized, as usual in C. *Scalar* `auto` entities can also be initialized.

```
int i = 23 ; // compile-time initialization
void f()
{
  int j = 0 ; // run-time initalization
  // ..
}
```

5.5 Scope control

The keyword `static` is implemented for data local to function, but not for data local to files.

```
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).

5.6 Address allocation

Real entity addresses are generated during final assembly and depends on link process, so you cannot make any assumption about normal variable/function locations.

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

[2]However, remember that the hardware stack is limited to 32 levels.

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

This is useful for referencing hardware registers. In the future, **cpik** will generate optimized code for accessing global variables located in page 0, so it will be interesting to manually locate critical data.

Please note that **cpik** *does not perform any verification of specified addresses.*

5.7 Instructions

All C instructions are implemented, excepted `switch`, which will be implemented later.

5.8 Operators

All C operators are implemented, excepted `"->"` and `"."`, which are related to structures.

5.9 Extensions

5.9.1 Binary constants

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

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

5.9.2 Assembler code

Assembly code can be included in C code using the (nonstandard) `__asm__` directive.

```
void f()
{
  __asm__("\tmovlw 0xFA\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.

5.9.3 Interrupt service routines

Two empty interruput service routines are provided in the RT library.

- `_hi_pri_ISR` for high priority interrupts,
- `_lo_pri_ISR` for low priority interrupts.

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

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

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

Routines tagged as `__interrupt__`

- Save and restore registers `W, FSR0L, FSR0H, FSR1L, FSR1H, STATUS, BSR` on the data stack.
- uses `retfie 0` instead of `return 0` . The `reftie 1` instruction is not used because it is explicitely mentionned as bogus by errata documents from Microchip.

ISR are currently not tested.

5.9.4 Pragmas

1. `#pragma processor device_name`

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

2. `#pragma _CONFIGxy value`

   This pragma allows to specify config bits.

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

   `value` is directly passed to assembler, so you can use here constants not defined at C level, but defined in the `<processor>.inc` file.

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

3. `#pragma _IDLOCx value`

   This pragma allows to specify ID data.

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

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

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

4. `#pragma firstsfr value`

   This pragma allows to specify the address of the first SFR. The default value is 0xF80, but some devices have more SFRs. This information is used by compiler to establish if a manually located variable may be accessed tru access bank. This pragma is used in each device specific header (ie: `p18fxxx.h`) so there is usually no need to use it again.

   `value` is an integer hex (`0xXXX`) or decimal (`dddd`) number.

6 Libraries

For the moment, only 4 libraries are available, and they have been developed as a support for compiler development itself. Excepted `rtl.slb` and `lcd.slb` (which are directly written in assembly language) each library `x` is written in C, and related to 3 files:

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

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

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

6.1 rs232

Minumum support for pic rs232 interface.

1. `void rs232_init()`

   Configure rs232 interface without interruption. Current speed is 9600 bauds with 16Mhz device clock. Source code must be modified for other speeds

2. `void rs232_putchar(char c)`

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

3. `char rs232_getchar()`

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

6.2 lcd

Support for classic LCD display, with 2 lines of 20 characters, in 4 bit mode. This display must be connected to port B. See source code for details.

1. `void lcd_init()`

   Initialize LCD subsystem. Must be used prior any LCD routine.

2. `void lcd_putchar(char c)`

   Display char `c` at current cursor position.

3. `void lcd_move(int pos)`

   Move cursor to device-specific position `pos`. For 20x2 displays, first line begins at position `0`, and second line at position `0x40`.

4. `void lcd_clear()`

   Erase display.

5. `void lcd_hex4(unsigned int c)` Display low nibble of `c` in hexadecimal. High nibble must be 0.

6. `void lcd_hex8(unsigned int c)` Display byte `c` in hexadecimal. Leading 0 is not suppressed.

7. `void lcd_hex16(unsigned long w)`

   Display 16 bit integer `w` in hexadecimal. Leading 0 are not suppressed.

8. `void lcd_puts(char *s)`

   Display nul terminated string pointed to by `s`.

`lcd_hex8`, `lcd_hex16` and `lcd_puts` are currently no more useful because they can be replaced by more general `stdio` routines.

6.3 stdio

Basic support for standard (and non-standard) buffered IO.

6.3.1 IO redirection

1. `void (*set_putchar_vector(void (*v)(char)))(char)`

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

2. `char (*set_getchar_vector( char (*v)()) )()`

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

Character I/O routine must cope with local/standard end-of-line translations: During inputs, EOL condition must be seen as a single `'\n'`. During outputs single `'\n'` must be expanded according to local EOL representation.

6.3.2 output

1. `void putchar(char c)`

   Write character `c`.

2. `int puts(char *s)`

   Write string `s`. Return the number of characters written.

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

   Writes unsigned long `n` in hexadecimal. If `up` is `'A'` uppercase letters are used for digits >= 10, else `up` must be equals to `'a'`, and lowercases are used. Leading 0 are suppressed, so this function can be also used for 8 bit numbers. Return the number of characters written.

4. `int 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. Return the number of characters written.

5. `int outdec(long n)`

   Writes long `n` in decimal. Leading 0 are suppressed, so this function can be also used for 8 bit numbers. Return the number of characters written.

6. `int printf(char *fmt)`

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

   Despite the fact cpik does not recognize the ANSI ≪ . . . ≫ syntax, this function uses a variable lengh argument list.

   Due to reduced memory size available, width specifiers (ie: `%4d`) are not supported.

6.3.3 input

1. `long getch()`

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

2. `long getche()`

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

3. `long getchar()`

   Returns the next available character, or EOF if no character is available. This character is echoed thru output vector. This function uses buffered IO. (See `ungetchar()`). The input buffer is a 80 bytes buffer.

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

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

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

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

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

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

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

   Please note that `fillbuf` is not intended to be used by end users and is just an implementation tool for buffered inputs.

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. `long ungetchar(long c)`

   Put back character `c` to input buffer. Putting back EOF is not an error and has no effect. Putting back more than one character leads to indefinite result. Return `c` if success, else `EOF`.

7. `void skipwchar()`

   Skip any number of ≪white≫ char from input. White char is either `' '`, `'\n'` or `'\t'`.

8. `int getlong(long *pn,int base)`

   This very versatile function allows to read signed or unsigned 16 bit number, in base `base`. Any base from `2` to `36` is valid. Leading minus sign is valid in any base (`-FFFF` in base `16` is interpreted as `1`)

   Return `1` is success, else `0`.

9. `int scanf(char *format_string)`

   General purpose classic input routine. Supported formats are:
   `%c %s %d %u %x %ld %lx %lu`

   Return the number of successfully exploited formats.

7 Source library structure

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

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

Here is the list of recognized special comments:

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

   ```
   ;<+module_name>
   ```

2. **End of module definition**

   ```
   ;<->
   ```

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

   ```
   ;<?module_name>
   ```

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

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

Example:

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

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

will generate:

```
;<+C18_table>
  CBLOCK
  C18_table:6
  ENDC
;<= 1 2 0 >
;<->

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

8 Standard headers

A standard header is provided for each `18FXXXX` device. For example, the header for `18F2330` is `p18f2330.h`, and is located in `/usr/share/cpik/include/`.

These headers contains SFR declarations as global manually located `unsigned int`s. It also contains bit definitions as `#define` preprocessor directives.

Headers are automatically generated from standard `.inc` files which come from `gpasm` (or `mpasm`), so they use uppercase standard names.

Here is an exerpt from `p18f2515.h`

```
// ...
unsigned int PCLATH@0xffa ;
unsigned int PCLATU@0xffb ;
unsigned int STKPTR@0xffc ;
unsigned int TOS@0xffd ;
unsigned int TOSL@0xffd ;
unsigned int TOSH@0xffe ;
unsigned int TOSU@0xfff ;

//
// start of SFRs block
//
#pragma firstsfr 0xf80

//
// Bits definitions
//

//----- PORTA Bits
#define RA0 0
#define RA1 1
#define RA2 2
#define RA3 3
#define RA4 4
#define RA5 5
#define RA6 6
#define RA7 7
// ...
```

9 Needed additionnal softwares

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.