PhrozenC user guide

written by Arnaud Storq

March 26, 2010

https://sourceforge.net/apps/trac/phobia/wiki/PhrozenC

Contents

1 Introduction

PhrozenC is a C compiler based on original SmallC compiler. SmallC has been initially created in 1980 by Ron Cain, targetting CP/M machines at the time (which also includes the Amstrad CPC). In the beginning of the year 2010, I found out back the compiler's source-code on the net and tried to compile it for the PC platform. It worked pretty well, and I personally found its implementation relatively small in terms of memory footprint. "For the fun" (like many projects started I guess :) I tried making a real Amstrad CPC port; at first, I was not able to get it working as expected but finally, after some tweaks and many headaches I succedeed in the task !

PhrozenC is available in several versions :

- **Windows PC** : the compiler is available as a stand-alone command-line application.
- **Amstrad CPC 464/464+** : the compiler is available as a stand-alone application that use the whole memory of the machine. After execution, all memory has been plainly used (and destroyed previous content). This version is released for compatibility purposes and is definitively not the way to go!
- **Amstrad CPC 6128/6128+** : the compiler is executed through RSX usage. It makes use of the full extra 64Kb memory (to keep the compiler in its compressed state, and also to preserve memory state in the first 64Kb memory area).
- **Amstrad CPC 6128/6128+ (ROM version)** : the compiler is available as a ROM application that can be executed through RSX usage. It makes use of the upper extra 48Kb memory (`C5`/`C6`/`C7` banks) but let intact resident memory and bank &C4. It's recommended to use the ROM version of PhrozenC conjointly with the popular *Arnor* ROM-based development products : *Protext*, *Maxam* and *Promerge*.

PhrozenC's usage is really easy. Basically, it converts a C file (and its dependencies) directly to a single ASM source file, ready to be compiled with Arnor's Maxam or Richard Wilson's WinAPE.

PhrozenC is a single pass compiler, meaning that it does not have to keep the whole source-code to be compiled in memory. It directly outputs ASM opcodes of C code being read.

Due to internal usage of SmallC, PhrozenC is not an ANSI-C compiler, but use K&R style C instead. Differences between ANSI-C and K&R style C are relatively small, and explained in a later section. That said, K&R style C is standard, and all compilers on PC platforms are able to compile K&R style C code (GCC, VisualStudio,..).

Finally, PhrozenC does not introduce specific optimizations. Developer is able to directly embeed inlined-ASM, meaning it's possible to include in the earth of a C source-code a pure bunch of ASM source-code.

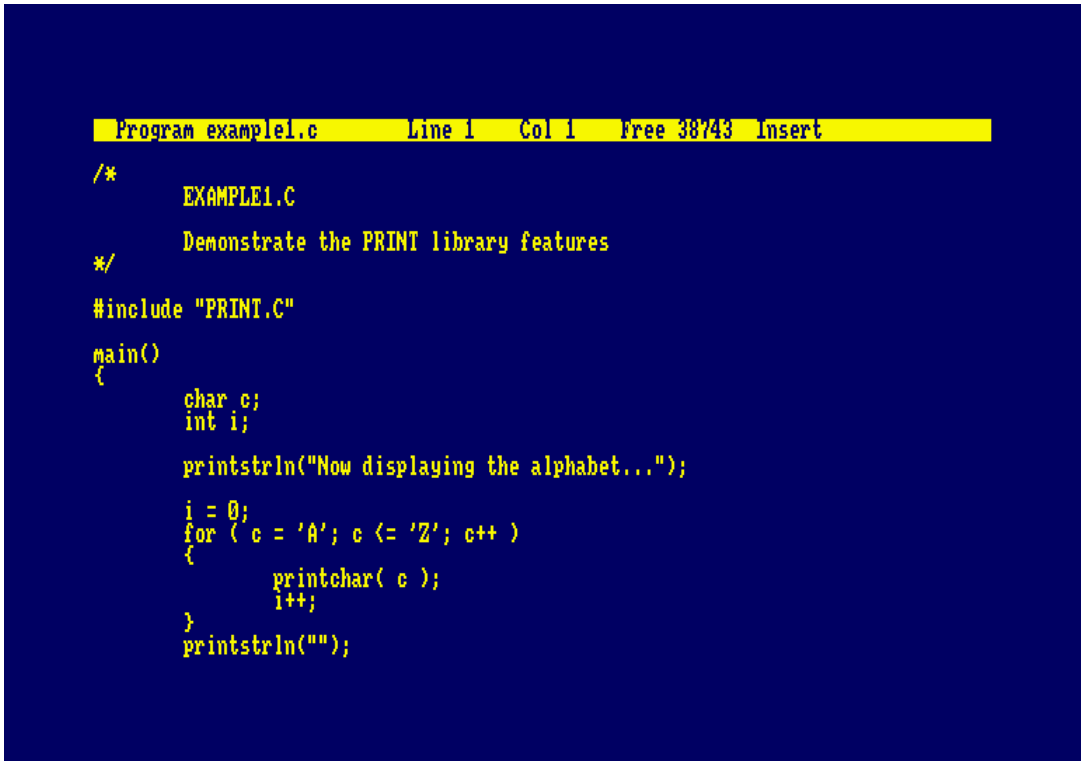

Figure 1: A C source (shown under Arnor's Protext word processor).

2 First compilation of a C program

First, let's create a small C program that will display 10 times the sentence "Hello World" (see how original it is ! :).

```
#include "PRINT.C"

void main()
{
        int i;

        for ( i =0; i < 10; i++ )
        {
                printstrln("Hello World");
        }
}
```

Edit this C source-code using your favorite text-editor and save it as `HELLO.C`.

2.1 Windows PC

In command-line mode, type `CC HELLO.C`. If no errors were found during the compilation, `HELLO.ASM` will be created in the same folder than the C source-file. Now you can open WinAPE emulator

and compile that generated ASM file, exactly like you would do with regular ASM source-code.

2.2 ROM version

Execute the following RSX command type `|CC,"HELLO.C"`. If no errors were found during the compilation, `HELLO.ASM` will be created. Now you can open Maxam and compile that generated ASM file, exactly like you would do with regular ASM source-code.

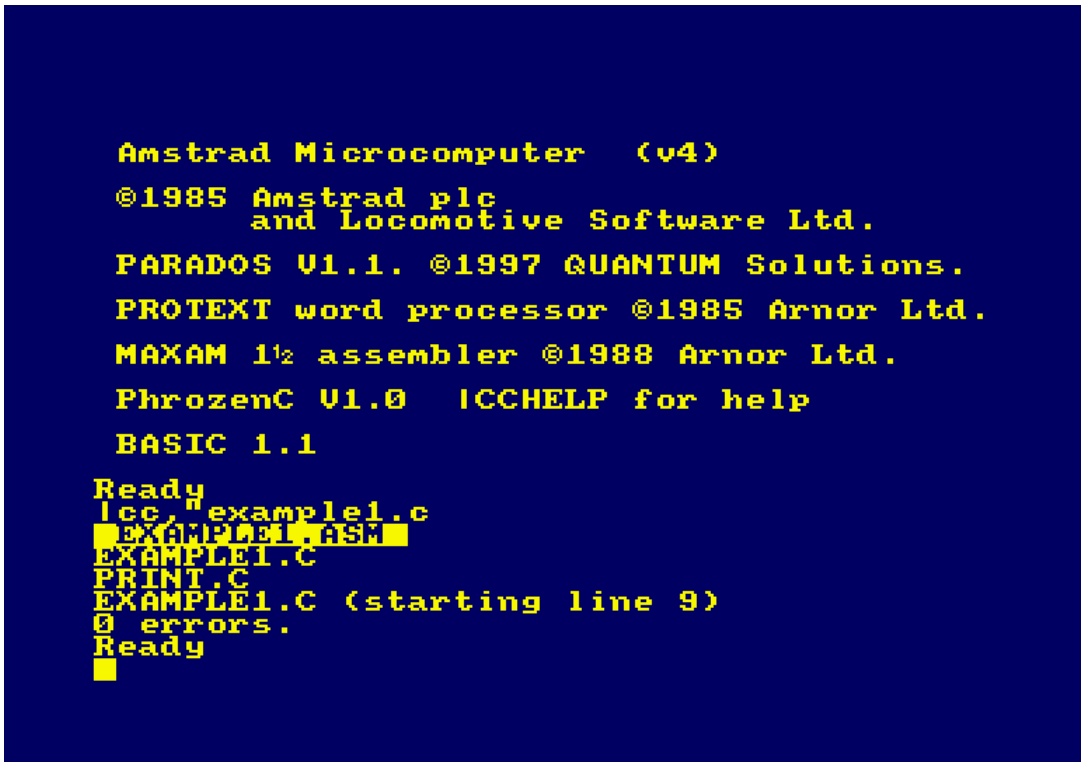

Figure 2: Compilation of a C program using ROM version.

2.3 RAM version

2.3.1 64Kb-compatible standalone version

Execute the following program : `RUN "-CC"`. Enter "HELLO.C" when the program asks for a filename (an empty string entered will display disc catalog). If no errors were found during the compilation, `HELLO.ASM` will be created, and you will be asked to press a key to reset the computer. Now you can open Maxam and compile that generated ASM file, exactly like you would do with regular ASM source-code.

2.3.2 128Kb-only RSX version

First, make sure to initialize PhrozenC in memory. To accomplish this, execute the following BASIC program : `RUN "-CC-RSX"`. Now, the RSX commands are installed. Execute the following RSX command type `|CC,"HELLO.C"`. If no errors were found during the compilation, `HELLO.ASM`

will be created. Now you can open Maxam and compile that generated ASM file, exactly like you would do with regular ASM source-code.

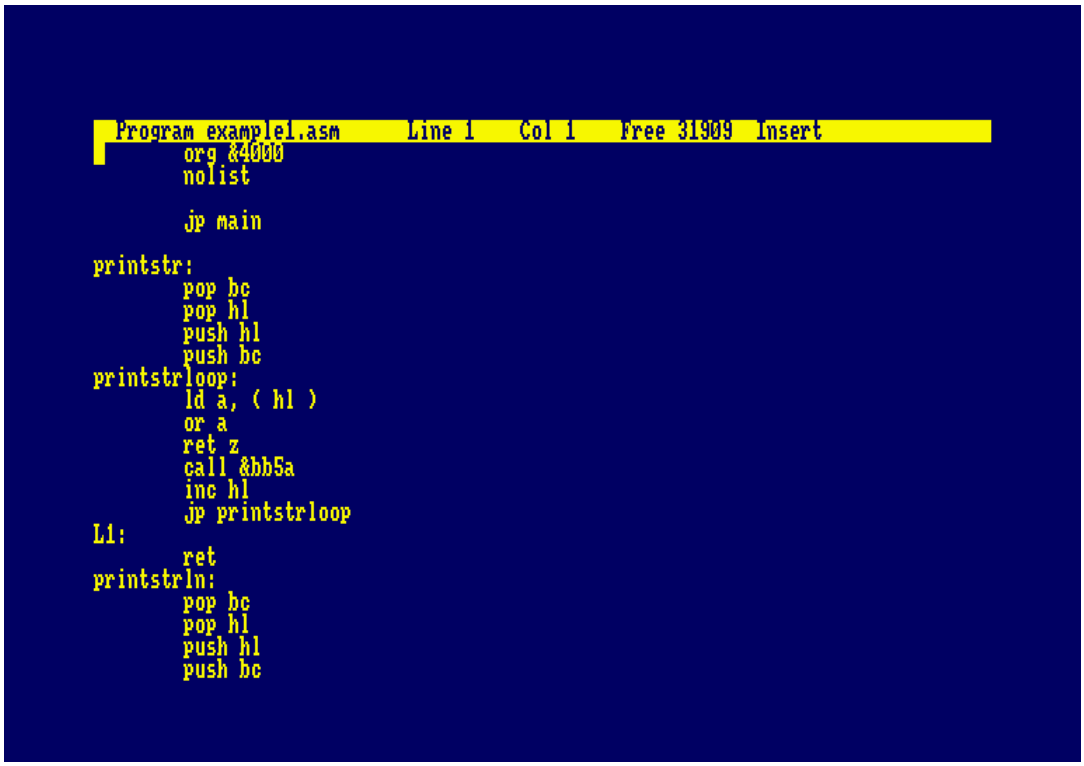

Figure 3: Generated ASM output (shown under Arnor's Protext word processor).

3 Available libraries

PhrozenC is a standalone C compiler. To help developer in making programs for the Amstrad CPC, a set of external C source-code is provided. These helpers, called librairies, are uniquely based on Amstrad CPC's firmware. It's eventually planned in a later version to extend those libraries with management of CRTC, Gate Array, interrupts,.. feel free to enhance current versions and send them to me, I will be happy to integrate your changes in future PhrozenC releases...

3.1 PRINT

This library features functions for displaying text to the user.

3.1.1 Declaration

```
/* print text to screen */
printstr(text) char *text;

/* print text to screen with line return appended to the end */
printstrln(text) char *text;

/* print ASCII char to screen */
printchar(c) char c;

/* print integer to screen (decimal version) */
printdec(i) int i;

/* print integer to screen (hexadecimal version) */
printhex(i) int i;
```

3.1.2 Example

```
/*
        EXAMPLE1.C

        Demonstrate the PRINT library features
*/

#include "PRINT.C"

main()
{
        char c;
        int i;

        printstrln("Now␣displaying␣the␣alphabet...");

        i = 0;
        for ( c = 'A'; c <= 'Z'; c++ )
```

```
        {
                printchar( c );
                i++;
        }
        printstrln("");

        printstr("Letter␣count␣:␣");
        printdec(i);
        printstrln("");
}
```

3.2 STRING

This library features functions for all string related operations.

3.2.1 Declaration

```
/* get length of string (outputs an integer) */
strlen(src) char *src;

/* copy string content from src to dst */
strcpy(src, dst) char *src; char *dst;

/* append stc string to dst string */
strcat(src, dst) char *src; char *dst;

/* converts integer to dest string, base being equals to 10 (decimal)
   or 16 (hexadecimal) */
itoa(intvalue, dest, base) int intvalue; char base; char *dest;
```

3.2.2 Example

```
/*
        EXAMPLE4.C

        Demonstrate the STRING library features
*/

#include "PRINT.C"
#include "STRING.C"

static char bigString[ 64 ];

main()
{
        char *ptr1;
        char *ptr2;
```

```
        int len;
        int aNumber;

        /* strlen */
        ptr1 = "ABCD1234";
        len = strlen( ptr1 );
        printstr(ptr1);
        printstr("␣=␣");
        printint(len);
        printstrln("␣characters");

        /* strcpy */
        strcpy( bigString, "I'm␣a␣copy␣!");
        printstrln( bigString );

        /* strcat */
        strcpy( bigString, "This␣is␣a␣");
        strcat( bigString, "string␣concatenation");
        printstrln( bigString );

        /* itoa */
        aNumber = 32;
        itoa(aNumber, bigString, 10);
        printstr(bigString);
        printstr("␣=␣&");
        itoa(aNumber, bigString, 16);
        printstrln(bigString);
}
```

3.3 GRAPHIC

This library features functions for all screen-related operations.

3.3.1 Declaration

```
/* set graphic mode to 0, 1 or 2 */
mode(mode) char mode;

/* clear screen */
cls();

/* set current for drawing operations */
pen(pen) char pen;

/* set color for specified pen */
ink(pen, color) char pen; char color;

/* set color for border */
border(color) char color;
```

```
/* move current position to x,y */
move(x, y) int x, int y;

/* get pen color at specified position x,y */
test(x, y) int x, int y;

/* plot a pixel at x,y coordinates */
plot(x, y) int x, int y;

/* draw a line to x,y coordinates */
draw(x, y) int x, y;
```

3.3.2 Example

```
/*
        EXAMPLE3.C

        Demonstrate the GRAPHIC library features
*/

#include "GRAPHIC.C"

#define STEP 10

main()
{
        int x;
        int y;

        mode(1);

        border(7);
        ink(2,26);
        ink(3,15);

        for ( x = 0; x < 200; x += STEP )
        {
                for ( y = 0; y < 200; y += STEP )
                {
                        pen(2);

                        plot(x, y);
                        draw(x+STEP, y+STEP);

                        pen(3);

                        plot(x+STEP, y);
```

```
                        draw(x, y+STEP);
                }
        }
}
```

3.4 INPUT

This library features functions for all input-related operations.

3.4.1 Declaration

```
/* wait for a pressed key. ASCII character is returned as output. */
waitKey();

/* fill destination buffer with string entered by user. */
getstr(dest) char *dest;
```

3.4.2 Example

```
/*
        EXAMPLE5.C

        Demonstrate the INPUT library features
*/

#include "PRINT.C"
#include "INPUT.C"

#define MAXNAMESIZE 10
static char name[ MAXNAMESIZE ];

main()
{
        printstr("Enter␣your␣name␣:␣");
        getstr(name);

        printstrln("");
        printstr("Your␣name␣is␣");
        printstrln(name);
}
```

3.5 MEMORY

This library features functions for all memory-related operations.

3.5.1 Declaration

```
/* copy a block of memory from src to dst with specified length */
memcpy(src, dest, length) char *src, char *dst, int length;

/* set bank, value being 0xC0, 0xC4, 0xC5, 0xC6, 0xC7. Previous bank is
    returned as output. */
bankSwitch(bank) char bank;
```

3.5.2 Example

```
/*
        EXAMPLE5.C

        Demonstrate the MEMORY library features
*/

/* after compilation, don't forget to override ORG position
        to a value different from 0x4000 to 0x7FFF ! */

#include "PRINT.C"
#include "MEMORY.C"
#include "GRAPHIC.C"
#include "INPUT.C"

main()
{
        int i;
        int bankList[ 5 ];

        bankList[ 0 ] = 0xc0;
        bankList[ 1 ] = 0xc4;
        bankList[ 2 ] = 0xc5;
        bankList[ 3 ] = 0xc6;
        bankList[ 4 ] = 0xc7;

        for ( i = 0; i < 5; i++ )
        {
                mode( 2 );

                bankSwitch( bankList[ i ] );
                memcpy( 0x4000, 0xc000, 0x4000 );

                printstr("this␣is␣bank␣&");
                printhex(bankList[ i ]);
                printstr("␣..");

                waitKey();
```

```
        }

        bankSwitch( 0xc0 );
}
```

3.6 FILE

This library features functions for all file-related operations.

3.6.1 Declaration

```
/* open a file for reading. Returns 0 if a problem occurs, 1 if OK. */
openread(filename,readbuffer) char *filename; char *readbuffer;

/* open a file for writing. Returns 0 if a problems occurs, 1 if OK. */
openwrite(filename,writebuffer) char *filename; char *writebuffer;

/* close file opened in read mode */
closeread();

/* close file opened in write mode */
closewrite();

/* read a char from reading file */
getchar();

/* returns 1 if end of current reading file has been reached. */
iseof();

/* read the whole content of the file to ptr */
getbinary(ptr) char *ptr;

/* write a char to writing file */
putchar(c) char c;

/* write a string to writing file */
putstr( text ) char *text;

/* write a string with carriage return to writing file */
putstrln( text ) char *text;

/* write content to file */
putbinary(ptr,length,execPtr) char *ptr; int length; char *execPtr;
```

3.6.2 Example

```
/*
```

```
        EXAMPLE2.C

        Demonstrate the FILE library features
*/

#include "PRINT.C"
#include "FILE.C"

#define TXTFILENAME "TESTTEXT.BIN"
#define BINFILENAME "TESTDATA.BIN"

static char readbuffer[ 2048 ];
static char writebuffer[ 2048 ];

main()
{
        /* write an ASCII file containing the alphabet */
        printstrln("** TEST 1/4 : WRITE ASCII FILE");
        writeText();

        /* read the ASCII file and display it to screen */
        printstrln("");
        printstrln("** TEST 2/4 : READ ASCII FILE");
        readText();

        initBinData();

        /* write a BINARY file containing the alphabet */
        printstrln("");
        printstrln("** TEST 3/4 : WRITE BINARY FILE");
        writeBinary();

        /* read the BINARY file and check that result is OK */
        printstrln("");
        printstrln("** TEST 4/4 : READ BINARY FILE");
        readBinary();
}

writeText()
{
        int i;

        if ( openwrite( TXTFILENAME, writebuffer ) == 0 )
        {
                return;
        }

        putstrln("This is the alphabet :");
```

```
        for ( i = 'A'; i <= 'Z'; i++ )
        {
                putchar(i);
        }
        putstrln("");

        closewrite();
}

readText()
{
        if ( openread( TXTFILENAME, readbuffer ) == 0 )
        {
                return;
        }

        while ( iseof() == 0 )
        {
                printchar( getchar() );
        }

        closeread();
}

#define BINARYSIZE 100
static char binWriteData[ BINARYSIZE ];
static char binReadData[ BINARYSIZE ];
static int checksum;

initBinData()
{
        int i;

        checksum = 0;

        for ( i = 0; i < BINARYSIZE; i++ )
        {
                binWriteData[ i ] = i;

                checksum += binWriteData[ i ];
        }
}

writeBinary()
{
        if ( openwrite( BINFILENAME, writebuffer ) == 0 )
        {
                return;
        }
```

```
        putbinary( binWriteData, BINARYSIZE, 0 );

        closewrite();
}

readBinary()
{
        int i;
        int readcs;

        if ( openread( BINFILENAME, readbuffer ) == 0 )
        {
                return;
        }

        getbinary( binReadData );

        readcs = 0;
        for ( i = 0; i < BINARYSIZE; i++ )
        {
                readcs += binReadData[ i ];
        }

        closeread();

        if ( readcs == checksum )
        {
                printstrln("Checksum␣OK␣!");
        }
        else
        {
                printstrln("Checksum␣FAILED␣!");
        }
}
```

4 Compiler internal

4.1 Supported C features

Because PhrozenC is based on SmallC compiler, developer has to use the K&R style C syntax. Modern compilers are ANSI-C, so expect many differences between PhrozenC and a full-featured ANSI C compiler.

- **all common operators are supported** :

```
-, +, *, /, %, <<, >>, &, ^, |, &&, ||, ==, -=, +=, *=, /-, %=,
   >>=, <<=, &=, ^=, |=
```

- **no float** : with Zilog Z80 it's recommended to use fixed math anyway.
- **no struct** : try reorganizing memory with indices instead.
- **no bool** : use char instead.
- **no macros** : macros/multi-lined `#DEFINE` are not supported.
- **no constants** : use `#DEFINE` instead.
- **no function pointers** : so calls by reference are impossible. But you can easily get over this using inlined asm.
- **all functions are assumed to return an int-sized output parameter** : that's why you don't need to specify the return types. You can return a char, an int, a pointer. Or nothing.
- **you can't assign a default value to a variable at declaration-time**
- **different function headers** : the types of the parameters are given separately from their names, in the space before the functions first open curly brace.
- **one dimensional arrays only** : multi-dimensional arrays are not supported.

4.2 Inlined assembly

4.2.1 Declaration

PhrozenC allows inlined-assembly. This means that everything between `#asm` and `#endasm` statements won't be treated by the C compiler, but present in ASM output.

```
function()
{
#asm
        ld a, 65
        call &bb5a
#endasm
}
```

4.2.2 Registers constraints

Good news for you : at any time, you can use ALL registers, even the stack (if your code does not corrupt it, of course).

4.2.3 Accessing C variables from an assembly block

Only global variables can be easily accessed. If you declare a global variable like this :

```
static int myGlobalInt;
```

..it will be inserted like this in the ASM output file :

```
myGlobalInt
        dw 0
```

As a consequence, you can easily write the following :

```
static int myGlobalInt;

function()
{
#asm
        ld hl, ( myGlobalInt )
        inc hl
        ld ( myGlobalInt ), hl
#endasm
}
```

4.2.4 Creating native functions

You can write functions that will be entirely implemented in assembly. In the following example, input parameters are available through the stack ; and output parameter is always the `HL` register's content.

```
strlen( text ) char *text;
{
#asm
        pop bc
        pop hl ; HL = text
        push hl
        push bc
        ld e, 0
strlenloop:
        ld a, (hl)
        or a
        jp z, strlenend
        inc hl
        inc e
        jp strlenloop
strlenend:
        ld h, 0 ; HL = output value
        ld l, e
#endasm
}
```

5 Small introduction to C

Like PhrozenC features a very small subset of C, it's easy to make some kind of tutorial about programming in C.

5.1 Hello, World

This is what looks a basic "Hello World" program using PhrozenC :

```
#include "PRINT.C"

void main()
{
        printstrln("Hello World");
}
```

5.2 Preprocessor DEFINE

In all C programs there is a `main` function which is followed by a and closed by a . `printStrln` function is used to print a string onto the screen. The first line tells to the compiler to include the whole content of the file `PRINT.C`, which contains the declaration of the `printstrln` function.

```
#include "PRINT.C"

#define HELLO "Hello World"

void main()
{
        printstrln(HELLO);
}
```

In the above example, we replaced the string `Hello World` by HELLO (defined by `#define` statement). Once declared, when the compiler meet HELLO next times it will replace text with the content of the define. `#define` also works with integers.
It's possible to undefine previously defined declarations using `#undef` ("undefine"). Also, it's possible to include blocks of code by testing defined declarations existence through `#ifdef` ("if defined"), `#ifndef` ("if NOT defined"), `#else` and `#endif` commands :

```
#define OPTIMIZE 1

void main()
{
#ifdef OPTIMIZE
        /* the uber optimization is here */
#else
        /* slow code, probably a version which is less buggy? */
#endif

#ifndef OPTIMIZE
        /* special treatment for non-optimized code.. */
```

```
#endif

#undef OPTIMIZE
        /* from here, the compiler don't know anything about the
           previous OPTIMIZE declaration */
}
```

5.3 Get string from user and displays it back to user

Now a new example, where user is asked to enter his age and displays it back to the user :

```
#include "PRINT.C"

void main()
{
        char ageText[ 30 ];

        printstr("Enter␣your␣age␣:␣");
        getstr(ageText);
        printstrln("");

        printstr("Your␣age␣is␣");
        printstrln(ageText);
}
```

In the above example, we actually declared a table of characters called `ageText` of size 30. The `getstr` function is used here to let the user type a string value and output it into our `ageText` table. Then we display it back to the user through the `printstrln` function. `printstr` is used here to display text without carriage return at the end, where `printstrln` adds a carriage return at the end of the string when displaying it.

5.4 Variables declaration

Basically, 2 types are available : `char` (which is 8 bits, values from -127 to 127) and `int` (which is 16 bits, values from -32767 to 32767).
You can also prefix types with `unsigned` statement, which would give `unsigned char` (which is 8 bits, values from 0 to 256) and `unsigned int` (which is 16 bits, values from 0 to 65535).

```
void main()
{
        unsigned int memorySize;
        unsigned char age;

        int angle;
        char temperature;
}
```

5.5 Loop (FOR statement)

Now let's have fun with loops. Let's say we want to display 10 times the message "Loop" :

```
#include "PRINT.C"

void main()
{
        int i;

        for ( i = 0; i < 10; i++ )
        {
                printstrln("Loop");
        }
}
```

The above example initialize the `i` variable to 0 (`i=0`), then increments it by 1 (`i++`) till it reachs 10 (`i<10`). So basically, the format for a FOR statement is as follows: `for(initial=value;condition;increment) instruction;`.

5.6 Incrementation in Loop (FOR statement)

But we could also get it incrementing by 2 instead of 1 with the following code :

```
#include "PRINT.C"

void main()
{
        int i;

        for ( i = 0; i < 10; i+=2 )
        {
                printstrln("Loop");
        }
}
```

We just changed `i++` by `i+=2`.

5.7 Display integer

Now let's say we want to display iterator value :

```
#include "PRINT.C"

void main()
{
        int i;
        char text[30];
        char textAsInt[30];

        for ( i = 0; i < 10; i+=2 )
        {
```

```
                printstr("Loop ");
                printdec(i);
                printstrln("");
        }
}
```

It displays "Loop " followed by integer value.

5.8 Loop (WHILE statement)

Now let's introduce a new way of doing loop, with `while` keyword :

```
#include "PRINT.C"

void main()
{
        int i;

        i = 10;
        while ( i != 0 )
        {
                printstrln("Hello");

                i--;
        }
}
```

This above example initialize the `i` integer to 10, displays the "Hello" string, decrements the `i` value tills it reachs 0.

5.9 Loop (DO/WHILE statement)

Also available, the structure `do/while` :

```
#include "PRINT.C"

void main()
{
        int i;

        i = 10;
        do
        {
                printstrln("Hello");

                i--;
        } while ( i != 0 );
}
```

This above example initialize the `i` integer to 10, displays the "Hello" string, decrements the `i` value tills it reachs 0. The only difference with the previous example is that the evaluation is done at the end of the code of block, instead of the beginning.

5.10 Arrays declaration and usage

Now let's play with arrays :

```
#include "PRINT.C"

void main()
{
        int i;
        int table[5];

        table[0] = 12;
        table[1] = -7565;
        table[2] = 98;
        table[3] = -5;
        table[4] = 267;

        for ( i = 0; i < 5; i++ )
        {
                printdec(i);
        }
}
```

This above example initialize the `table` array with 5 items, that are later initialized with values. Please note that the first element in an array is designed by index 0, not 1. Finally, we use the `io_printIntln` function which allows to display an integer to screen with carriage return.

5.11 Switch/case statement

Instead of cumulating `if` statements, it's possible to make custom code implementation depending of a condition :

```
#include "PRINT.C"

void main()
{
        int i;

        i = 4;

        switch( i )
        {
                case 4:
                        printstrln("Value␣is␣4");
                        break;

                case 2:
```

```
                        printstrln("Value is 2");
                        break;

                default:
                        printstrln("Value is not 4 nor 2");
                        break;
        }
}
```

Please note the usage of `break` command which indicates where custom implementation for a given result ends.

5.12 Pointers

New thing to cover, let's deal with pointers. Basically, pointers are variables that refer to memory locations of variables. Let's see how it works by taking back with previous example :

```
#include "PRINT.C"

void main()
{
        int i;
        int table[5];
        int *ptr;
        int tempI;

        table[0] = 12;
        table[1] = -7565;
        table[2] = 98;
        table[3] = -5;
        table[4] = 267;

        ptr = table;

        for ( i = 0; i < 5; i++ )
        {
                tempI = ptr[ i ];
                printdec(tempI);
        }
}
```

This produces exactly the same behavior than previous example.

5.13 Functions declaration and usage

Finally, let's introduce function usage, by taking back previous example :

```
#include "PRINT.C"

void main()
{
```

```
        int i;
        int table[5];
        int *ptr;
        int tempI;

        table[0] = 12;
        table[1] = -7565;
        table[2] = 98;
        table[3] = -5;
        table[4] = 267;

        ptr = table;

        showTable(ptr);
}

showTable(p) int *p;
{
        for ( i = 0; i < 5; i++ )
        {
                tempI = ptr[ i ];
                printdec(tempI);
        }
}
```

What it basically does is to declare a function called `showTable` that takes as parameter a pointer of type int called `p`.

5.14 Functions returning values

Another usage of functions :

```
#include "PRINT.C"

void main()
{
        int i;

        i = add( 4, 6 );

        printdec(i);
}

add(a, b) int a, int b;
{
        return a+b;
}
```

The above example will display 10 to the user. It actually calls the `add` functions, which takes 2 int parameters called `a` and `b`. In that custom function, we returned the sum of the 2 integers.
If function has to return something else than an integer, you don't have to specify its type :

```
getStringWithoutFirstchar(ptr) char *ptr;
{
        return ptr+1;
}
```

5.15 Including multiple C files together

It's possible to link several C source-code in a one-and-only file using the `#include` statement.

```
/* MAIN.C */
#include "PRINT.C"

#include "ADD.C"

void main()
{
        int i;

        i = add( 4, 6 );

        printdec(i);
}
```

```
/* ADD.C */
add(a, b) int a, int b;
{
        return a+b;
}
```

Compiling MAIN.C will result in succesful build. The `add` function has been included.

5.16 Comments in source-code

Now, let's see how to add comment in a source :

```
/* include the IO functionalities */
#include "PRINT.C"

/* main function */
void main()
{
        /* declare values on stack */
        char isError;
        char c;

        /* try opening file */
        isError = io_fopen_readStream("README.TXT");
        if ( isError != 0 )
        {
```

```
                printstrln("Error occured while opening file");
        }
        else
        {
                /* display file content */
                do
                {
                        c = io_readStream_char();
                        if ( c != 0 )
                        {
                                io_printChar(c);
                        }
                } while ( c != 0 );

                /* close file */
                io_fclose_readStream();
        }
}
```

5.17 Using global variables

It's possible to declare a variable outside a code block delimited by and . Important note : always use static keyword when the scope is global !

```
static int i;

void main()
{
        for ( i = 0; i < 5; i++ )
        {
#asm
                ld a, 65
                call &BB5A
#endasm
        }
}
```

That way, every functions can directly access the variable without passing it through parameters.

5.18 Using inlined ASM

Now, let's see how to use inlined ASM in a C source :

```
void main()
{
        int i;

        for ( i = 0; i < 5; i++ )
        {
#asm
```

```
                ld a, 65
                call &BB5A
#endasm
        }
}
```

The above example displays 5 characters 'A' to screen directly using firmware method. There is no special constraint in using asm inlined code. You can even kill interrupts if you want, move stack pointer everywhere you want.. and also access global variable :

```
static char i;
void main()
{
        for ( i = 0; i < 5; i++ )
        {
#asm
                ld a, (i)
                add a, 65
                call &BB5A
#endasm
        }
}
```

That will display ABCDEF to the user.

6 Examining ASM output

6.1 C source-code

Let's take this basic C source-code as example :

```
static int globalVarA;
static char globalTable[256];

main()
{
        int iOnStack;

        iOnStack = add2(4,5);

        for ( iOnStack = 2; iOnStack < 5; iOnStack++ )
        {
                printA();
        }

        return 27;
}

printA()
{
#asm
        ld a, 65
        call &bb5a
#endasm
}

add2(a,b) int a; int b;
{
        return a+b;
}
```

6.2 Generated ASM output listing

This is the generated ASM output listing :

```
        org &4000
        nolist
        jp main
main:
        push bc
        ld hl,65536
        add hl,sp
        push hl
        ld hl,4
        push hl
        ld hl,5
        push hl
        call add2
        pop bc
        pop bc
        pop de
        call Lpint
        ld hl,65536
        add hl,sp
        push hl
        ld hl,2
        pop de
        call Lpint
L2:
        ld hl,65536
        add hl,sp
        call Lgint
        push hl
        ld hl,5
        pop de
        call Llt
        ld a,h
        or l
        jp nz,L4
        jp L5
L3:
        ld hl,65536
        add hl,sp
        push hl
        call Lgint
        inc hl
        pop de
        call Lpint
        dec hl
        jp L2
L4:
        call printA
```

```
        jp L3
L5:
        ld hl,27
        jp L1
L1:
        pop bc
        ret
printA:
        ld a, 65
        call &bb5a
L6:
        ret
add2:
        ld hl,4
        add hl,sp
        call Lgint
        push hl
        ld hl,4
        add hl,sp
        call Lgint
        pop de
        add hl,de
        jp L7
L7:
        ret
globalVarA:
        dw 0
globalTable:
        ds 256

Lpint:
        ld a,l
        ld (de),a
        inc de
        ld a,h
        ld (de),a
        ret

Lgint:
        ld a,(hl)
        inc hl
        ld h,(hl)
        ld l,a
        ret

Llt:
R_crt5:
        call Lcmp
        ret c
```

```
        dec hl
        ret

Lcmp:
        ld a,e
        sub l
        ld e,a
        ld a,d
        sbc a,h
        ld hl,1
        jp m,Lcmp1
        or e
        ret
Lcmp1:
        or e
        scf
        ret
```

6.3 Observations

- **Generated code - overall** : basically, generated code use intensively the stack to store its local variables. It also makes use of HL and DE 16-bits registers, as also the accumulator. It will never use something else!
- **Generated code - speed** : generated code is absolutely not optimized in terms of speed, so you quickly inderstand where you should use C and where it does not apply to the context.
- **Generated code - size** : generated code is absolutely not optimized in terms of size, but the good news here is that due to its low usage of different ASM instructions, it gets packed pretty well.
- **Global variables** : all global variables are put into the same section, like this :

```
globalVarA:
        dw 0
globalTable:
        ds 256
```

- **Local variables** : all local variables are pushed onto the stack. Warning : if you are planning to use recursive functions, feel free to move the stack pointer at a custom location.
- **Inlined ASM** : Code between `asm`/`endasm` statements did not have been modified. Important note : inlined ASM can use ALL registers, even the accumulator, the second register set (EXX), the stack, DE, HL... this is a great feature regarding other existing cross-platform C compilers.
- **CRT functions** : To get C code working as expected, some additional code are integrated into the generated ASM output. In the example, these are `Lpint`, `Lgint`, `Llt` and `Lcmp`.

7 Frequently asked questions (FAQ)

- **Do you seriously consider using C to make CPC programs ?**
  Of course, yes ! Phat and Pheelone demos have been created using a C compiler and it proved to be a completely viable solution.
  That said, inlined ASM is still mandatory in term of performances. Don't expect bringing a port of DOOM game to CPC, it's just impossible. You should use plain C to make utilies. In the context where performances matters, use C to make your initializations, file loading, structured code, big loops.. but let to the inlined ASM be the slowest part to execute.

- **What did you change from the original SmallC ?**
  The main logic of SmallC is still here. I had to completely rewrote the input/output parts, integratde the CRT functions embedded with the source, understood and tweaked memory management, removed unused parts of code, made modifications to the reading code in order to get only one file in reading mode at one time, added compilation-stop when a first error is found, and lots more..
  That said, I tried to no introduce hacks to original compilation algorythm. I wished to keep a clean and reliable port for the Amstrad CPC platform.

- **Do you have some C optimizing tips to share ?**
  If you have multiple variables to initialize at the same time with a default values, prefer the following :

```
{
        int a;
        int b;
        int c;

        a = b = c = 0;
}
```

.. to this :

```
{
        int a;
        int b;
        int c;

        a = 0;
        b = 0;
        c = 0;
}
```

Also, make sure to understand where goes the variable allocations. As a reminder, local variables lives on stack, and global variables lives at static memory address. For performance reasons, I advise to use global variables if possible when speed matters, so that way variables won't be pushed on stack.

Finally, try to use `char` type instead of `int`, whenever it's possible.

8 Credits

These are the people involved in this production :

- Original SmallC programming by Ron Cain (1980)
- Amstrad CPC port, libraries and documentation by Arnaud Storq (NoRecess, 2010)
- Technical support provided by Grim, Targhan and Offset
  Thank you guys! :)
- CPCWiki, Kevin Thacker for the source for file access info
- Third-party tools used : BitBuster (T&J version), WinAPE, ManageDSK.. many thanks to the author !!

9 Contact & Closing words

I tried to do my best in providing a simple and reliable C compiler to the CPC platform.

For any information, suggestions, enhancements, bugfixes or whatever, feel free to contact me at `arnaud.storq@gmail.com`.

Thank you for using PhrozenC !