BlueRay 105 System Programming Guide

C

Jianzhong Fang

DEC 2007
OCT 2018
JUN 2024

Background

The C language compiler is created to support embedded system development using BR105 microprocessor. The compiler utility is CC105. It implements a subset of ANSI C grammar. It supports most features that are required by user programming. The details about the language extension and particulars are described in later sections. Along with BR105 assembler, simulator, and down-loader, this compiler provides a powerful tool chain for customers.

As this compiler outputs assembly program, it is very useful to understand how a C language compiler works. This development tool set is an ideal stuff for both engineers and amateurs and high school or university students.

Hello World!

As a part of tutorial section, for the purpose to present an understandable knowledge, we begin with this very traditional way to start this embedded C language for development.

```
/*
Hello World !
Simple code, big step
BR105 single board computer system.
Author: Jianzhong Fang @JiangSu China, JUN 2007
*/

#include <minilib.h>
char s[16]="Hello World !\n";

void main(void)
{
  char line[256];

  while (1) {
    mprintf("%s", s);
    gets(line);
  }
}
```

The above C language source code is named “hello.c”. When this program runs, it outputs a line of message “Hello World !”, and repeats to read chars from keyboard until a carriage return is received.
It is an infinite loop, and the program will not exit until external force used to terminate it.

#include <minilib.h>
is the header file, in which defines the prototypes of two functions, “mprintf” and “gets”.

mprintf(char *, …)
outputs an integer value or a string to UART at a given baud rate. It is a minimum implementation of “printf”, which is a standard C library. The source code of mprintf() is listed in library section.

gets(char *)
reads a string from UART, once the carriage return is encountered, this routine completes and exits. In this example, a line of chars are stored in “line”, a local buffer within the main() function.

How to get this piece of program running on BR105 microprocessor?

Users need to have two steps to set up this working environment properly to get the system running.

Step 1. Set up the development kit software, connecting the board to the PC through UART. To do this, one can follow the procedures below.

[a] Connect the BR105 board to the PC through a serial cable provided, connect the power supply to the BR105 board

[b] On PC, make sure which COM port is now connected to the BR105 system. Create a hyper terminal from windows accessories menu. Set the baud rate to 9600, data bits: 8, no parity check, stop bit: 1, no flow control.

[c] On BR105 system board press button 1 and hold it, press button 2 and hold it; then release button 1, then release button 2. The operating sequence leads the BR105 to download mode.

[d] On PC keyboard, press carriage return keys a few time, you can see a response “@” displayed on the hyper terminal. This response suggests that the BR105 system is in download mode, waiting for downloading program.

Step 2. Compile, assemble, and download this program. To do this, one can follow the following procedures.

[a] Do compilation.

Suppose the header file “minilib.h” is located in “include” directory, and this C file “hello.c” is located in “example” directory. “include” directory and “example” directory are on the same level. So the following compilation command is used to do this in DOS windows command line.

```
C:\example> cc105 –I ..\include <hello.c >hello.asm
```

This command searches the header file from “include” directory and inputs “hello.c” source C file, and outputs an assembly file named “hello.asm”, saved on the disk. If no output filename is provided, the result is displayed on the screen.

```
.CODE

FUNC A0_main:
L$hello000:
MOV.w  R4, 0x0001
AND  R4, R4
JZ  L$hello001
MOV  R4, R1
ADD.l  R4, @A0_s
ST.l  R4, 0x100(R0)
MOV.l  R4, @X$hello001
ADD  R4, R1
ST.l  R4, 0x104(R0)
LD.l  R4, 0x100(R0)
```

```
ST.l  R4, 0x108(R0)
PUSH  R0
ADD.l R0, 0x104
CALL A0_mprintf
POP  R0
MOV  R4, R3
MOV  R4, R0
ADD.l  R4, 0x0
ST.l  R4, 0x104(R0)
LD.l  R4, 0x104(R0)
ST.l  R4, 0x108(R0)
PUSH  R0
ADD.l R0, 0x108
CALL A0_gets
POP  R0
MOV  R4, R3
JUMP  L$hello000
L$hello001:
RET

.DATA
A0_s       DT "Hello World !\n"
X$hello001  DT "%s"
```

[b] Merge the assembly files
As the "hello.asm" does not include the library functions (mprintf and gets), and also the system needs a start up file, "init.asm", to boot the system every time when reset button is applied or the power supply is applied. Therefore, we need to put the three assembly files together.

```
C:\example> copy  ..\lib\init.asm + ..\lib\minilib.asm + hello.asm  prog.asm
```

Now the new assembly file "prog.asm" is just the collection of above three assembly files. Where "minilib.asm" is the library code, whose interfaces are prototyped through "minilib.h" header file.

[c] Assemble prog.asm
The assembler AS105 will translate all the assembly statements into binary instructions for BR105. Eventually two sections in the memory are created, code section and data section. The code section consists of all the routines and functions of the program and the data section consists of global variables and string constants.

```
C:\example> as105 <prog.asm >prog.hex
```

Its output "prog.hex" is a readable text file. It is a memory map, where the binary code and data are actually stored. If no HEX filename is provided, the output will be displayed on the screen.

[d] download prog.hex
Before downloading the program, one makes sure that the BR105 system is in downloading mode. The downloading command is as follows. Suppose the currently connected hyper terminal is COM1 on PC.

```
C:\example> fsh105 prog.hex
```

The downloading utility reads the program file “prog.hex” and saves every record in the file to a serial flash memory through Serial Peripheral Interface. During the course of downloading, the blue lights flash, suggesting that a normal downloading activity is underway.

So far, all the steps have been competed in order to watch the program running. One can press carriage return key on the keyboard from serial hyper-terminal and see what happens to the hyper terminal. It continues to display this message,
“Hello World !”

Congratulations! You have made it happen. One can change this program and do more. At this stage, the embedded program stays and runs on the small board. The programs stored in flash memory won’t be lost even when the power is turned off. Various programs can be created in this way and applied to various embedded applications. As we will get more knowledge about BR105 system, more sophisticated programs and more complicated tasks will be done with BR105 microprocessor.

CC105 Convention

Registers

Br105 architecture defines 16 32-bit registers. They are denoted by a capital letter "R" followed by one hexadecimal number. R0 – R9, Ra – Rf. Apart from its common usage, Rf is defined by the BR105 system architecture as a system stack pointer. When call and return instructions are executed, the Rf will be updated according to the stack growth. The rest 15 registers are not affected and exactly the same in their function. However, CC105 has indeed configured some of them for special purposes as far as some basic conventions are required by the compiler.

R0 – local stack pointer for user local variables. A caller will create a local stack for callee's local variables and function pass-in parameters are recommended to store in user stack.

R1 – base address for global variables. When parallel flash memories are used to store the code and data for run-time purpose (as a part of memory space that many embedded systems employ this solution), R1 is necessary to provide a base address for global data in RAM. The simple reason for doing this is that the flash memory cannot just be read/written as RAM can. This problem can be solved by relocation technique. So the data movement is required to move the global data from parallel flash memory to RAM before the program is executed. In BR105 system, things are slightly different. BR105 uses serial flash memory to store embedded program and data, before running embedded program, the whole image is loaded in to RAM. Therefore, there is no need to keep R1 in such situation. However, we can still keep R1 as the global data base address in RAM to harness all the convenience for system extension.

R2 – used by CC105 compiler as an intermediate address register. When resolving some complicated address calculation, such as array, record field, and pointer access, R2 may be used to calculate the final target address for some variables. This is done automatically by CC105 compiler algorithm.

R3 – used by CC105 compiler as a function return value. As R3 is a 32-bit register, for all the basic data types in C it can return directly, for record type and other combined data types, users should make sure the communication of the return value should be a pointer or reference to a memory structure.

Rf – system stack pointer. System stack saves the PSW and return address for interrupt, and return address for function call. System stack is essential for all the programs.

R4 – Re are general purpose registers for users. If users write assembly language for BR105, users are encouraged to use and configure these registers for their own preference. The register usage conventions are important when mixed language programming is considered.

Caller and Callee

The C program is actually consists of functions and global declarations. For users, the very beginning function is main(). For the compiler, no matter what the function (or routine) name is, the processing compilation for a function is quite the same. Functions can call a new function inside the function body, and even calls itself recursively. The function that calls another function is the caller and the function that is called by other functions is the callee. The communication established between caller and callee is through the function name and parameters that have been passed from caller to callee. From the viewpoint of compiler the process of calling and being called can be summarized as follows using three-addressed pseudo code (TAC).

When a function starts to call another function, the compiler creates the following sequence.
[1] TAC_PREARG: do alignment for the new frame to be created in user stack and spill temporary variables. These temporary variables are stored in caller's user stack.
[2] For All the pass-in parameters, from left to right, do TAC_ARG, to store the necessary arguments to the new frame based on their data size.
[3] TAC_FRAME: Change pointer of user stack R0 and save the caller's frame base (R0) in system stack (PUSH R0), and create a new user's stack for callee: R0 ← R0 + tos. Where "tos" is the user stack space claimed by caller's local variables.
[4] TAC_CALL: Purge all entries of register description table and emit the call instruction.
[5] TAC_POSTCALL: Restore R0 and accept the return value in R3 if required. Once R0 is recovered, the program continues as if nothing had happened to the user's stack.

The TAC code of the called sequence for the callee's function body, is
[1] TAC_FUNC: define a new function. Purge the register description table.
[2] TAC_BEGINFUN: set local stack top to zero. tos = 0.
[3] TAC_VAR: enter the parameters to symbol table, allocate space for each parameter. Please take attention: the SYM_3DOT("…") type parameter has a possible variable parameter list, whose local storage size cannot be decided during compilation. In order to cope with this problem with a reasonable solution, CC105 compiler reserves 64 bytes storage in local stack for this parameter list. Such case can happen to a very common routine, printf(char *, …).
[4] Translate C language statements to TAC pseudo codes
[5] TAC_ENDFUN: Conclude the function definition. Add RET instruction at the end of function body in case no return ending in C code.

In order to illustrate this calling and being called convention, we use an example to show they relationship linked by the user stack.

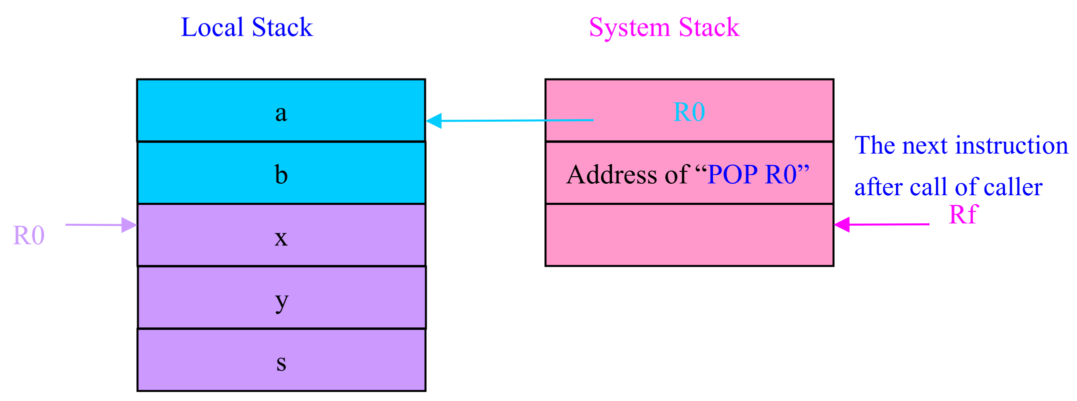
Suppose we define two functions, the caller "foo" and the callee "koo".
The C code of the functions are listed as below.

```
long koo(long x, long y)
{
  long s;
  s = x * y;
  return s;
}

void foo(void)
{
  long a, b;
  a = 10;
  b = 20;
  koo(a, b);
}
```

After compilation by CC105, the following piece of code is generated.

```
.CODE
FUNC A0_koo:
LD.l  R4, 0x0(R0)
LD.l  R5, 0x4(R0)
MUL  R4, R5
ST.l  R4, 0x8(R0)
MOV  R3, R4
RET
ENDF
FUNC A0_foo:
MOV.w  R4, 0x000a
ST.l  R4, 0x0(R0)
MOV.w  R5, 0x0014
ST.l  R5, 0x4(R0)
LD.l  R4, 0x0(R0)
ST.l  R4, 0x8(R0)
LD.l  R4, 0x4(R0)
ST.l  R4, 0xc(R0)
PUSH  R0
ADD.l R0, 0x8
CALL A0_koo
POP  R0
MOV  R4, R3
RET
ENDF

.DATA
```

The local stack layout is



The caller's (foo's) local stack pointer R0 is stored in system stack after the callee (koo) is invoked. Before koo is called, the caller (foo) constructs a local stack frame for callee (koo),. This task includes calculate the local stack space for its own variables, push the parameters (x, y) to the local stack and calculate the beginning address of the new local stack for the callee (koo). The caller's R0 is then pushed to the system stack, and the R0 is adjusted to a new pointer to the callee's local stack. Once CALL instruction is executed, the next address of CALL (the address of POP R0) is automatically pushed to the system stack. And then the CPU starts to execute koo program. When this program returns from callee (koo), the return address (address of POP R0) is automatically popped out and restored, then the POP R0 instruction is executed and local stack frame then restored back to foo's local space again, as if nothing happened to the local stack and system stack. This is the communication mechanism between the caller and callee. It is worth pointing out that the callee (koo) has a returned value which is the production of parameter "a" and "b". This value is stored in R3, and immediately taken by the caller on koo's return.

Symbol name decoration

As a C source may include several functions, different functions may have the same named local variables. CC105 compiler maintains a single symbol table, in the meanwhile, it tells apart the same named variables from different functions. Internally, it created prefixed symbol name decoration, although the eventual output in the assembly language does not show any local variable names. The similar techniques are applied to internal temporary variables, internal labels, string literals, global variables, and global function names.

Among them, three kinds of the symbols eventually turn up in the output assembly file. They are global function names, global variables, and internal labels.

The function names and global variables are decorated by headed with "A0_", this rule applies to global entity names. Under mixed language programming environment, to understand this decoration convention is very important, as it establishes communication protocol between the C program and Assembly program through this naming space. We will have a multi-task example to code the interrupt handler using this way in later section.

The internal labels are generated widely during compilation. The “while” statement, “case” statement, “for” statement, and so on, will eventually generated a number of internal labels for the program to change its course wherever needed. Besides the internal labels, external labels should be kept properly in the assembly file. Both internal labels and external labels should be unique among the whole naming space when several assembly files are put together. So the following symbol decoration scheme is applied to the labels.
L$&filename&serial_number

For example, the abc.c file looks like the following code.

```
void main(void)
{
  for (; ; ) ;
}
```

“abc.c” is compiled by CC105 using the following command.

```
C:> cc105 abc.c –o abc.asm
```

The following assembly code is generated.

```
.CODE

FUNC A0_main:
L$abc000:
NOP
L$abc002:
JUMP  L$abc000
L$abc001:
RET

.DATA
```

L$abc000 is the one really controls the program execution. L$abc001 and L$abc002 are also the examples of the internal labels, although they have no effect in this circumstance.

Code section and Data section

After compilation, every assembly file will have at least one section, the code section. If the C file module has no global data or string constant, there will be no data section. Otherwise, the global data and string literals are organized in a data section. As a part of software the data section keeps as important information as the code section. Both code section and data section should be stored in non-volatile memory. Most embedded solution stores both sections in parallel flash memory as a part of memory space where is run-time accessed by the CPU. However, such a solution has some drawbacks. One obvious disadvantage is that the global variable is read only, cannot be written as RAM can be done. The other disadvantage is that the speed of flash memory is usually much slower than RAM, so there is speed mismatch for this solution. To solve the first problem is by moving global data to be written to the RAM area in memory space. The data relocation caused the memory access failure because of the offset between the original data location and the new data location. BR105 assembler assumes the data section start from a zero offset based on R1, so the new location in the RAM can be assigned to R1 before the program starting. This is a proper way to solve this data relocation problem.

Although BR105 supports this solution for parallel flash memory implementation, a new technique has already been employed in this board module. BR105 stores all the code section and data section in a serial flash memory, and treat this flash memory just as an IO device. When the system starts, the whole image of the program is uploaded into RAM. So the code section and data section both reside in RAM. The base register R1 of global variables can be directly assigned with Data_start macro, or one can use the conventional method to move the data section and assign it with relocation base address. Both methods render the same result. However, the conventional method consumes more memory space. It is user’s up to configure the memory space for embedded system.

As R1 is saved for global data base address, there for all the reference to global variable and string literals should be based on R1. This should be careful when user write an assembly language module and try to communication with global variables defined in other modules.

AS105 assembler automatically incorporates R1 as the base address into the assembly statements whenever a global variable is being referred.

CC105 Extension

CC105 compiler supports a set of unique features to make the development much easier. CC105 command line syntax is simple but it can generate different output format and support to debug source code error. C language is also extended by inline assembly code, which makes it powerful to handle the program in great convenience. This inline feature also helps developers to understand computer technology greatly. The following topics will describe such unique features.

CC105 command line syntax

Under the DOS windows, CC105 help message is displayed by the following command

```
C> cc105 -?
```

It will display some message lines which look like:

```
cc105 input -o output
cc105 [-E] [-D<symbol>] [-I<dir>] input –o output
cc105 –debug input
cc105 –tac input
```

Where “input” refers to the input source C file to be compiled; “output” refers to the output assembly file to be generated. Option “-debug” displays the error location exactly when there is error in C source code. in this case, the complier will exit immediately once an error is encountered. Option “-tac” outputs a three address code list instead of assembly code list. User can use this option to study the compiler itself. Option “-E” just does preprocess C source code only. It will not generate any assembly code. “-D” defines a macro if such a macro is to be used in source code by the users. The macro can also be set a value by using “-D<symbol=value>”. “-I” tells the compiler where to search the included header files.

Suppose here is a piece of C source code named “mycode.c”

```c
int mprintf(char *s, ...);
int main(int agc, char **agv)
{
  int b;
  int a;
  a=1;
  b=2;
  mprintf("end: %i\n", a+b);
  return(a+b);
}
```

Demonstrating “–tac” option

After the following command is issued.

```
C> cc105 –tac <mycode.c
```

The following TAC code is generated. For more about TAC code, please refer to the source code of cc105 compiler. If it's no use for you at the moment, please ignore this.

```
.CODE
TAC_NIL
FUNC A0_main:
TAC_BEGINFUN
TAC_VAR A1_agc
TAC_VAR A1_agv
TAC_NIL
TAC_VAR A1_b
TAC_NIL
TAC_VAR A1_a
TAC_NIL
TAC_NIL
TAC_ASSIGN A1_a <= 0x1
TAC_CLEAR
TAC_NIL
TAC_NIL
TAC_ASSIGN A1_b <= 0x2
TAC_CLEAR
TAC_NIL
TAC_TXT X$000
TAC_NIL
TAC_NIL
TAC_opADD T$000 <=  A1_a,  A1_b
TAC_PREARG
TAC_ARG X$000
TAC_ARG T$000
TAC_FRAME
TAC_CALL A0_printf
TAC_POSTCALL T$001
TAC_CLEAR
TAC_NIL
TAC_NIL
TAC_opADD T$002 <=  A1_a,  A1_b
TAC_RET T$002
TAC_ENDFUN

.DATA
X$000     DT "end: %i\n"
```

Demonstrating "-debug" option

If we slightly change this source program to make it syntax wrong, then we can use "-debug" option to locate the error. For example, we omit the comer "," from the "mprintf" parameter delimiter, then this statement looks like
mprinft("end: %i\n"a+b); and the following command is applied

```
C> cc105 –debug mycode.c
```

The following message will display:

```
int mprintf(char *s, ...);
int main(int agc, char **agv)
{
int b;
int a;
a=1;
b=2;
mprintf("end: %d\n"a line: 11  column: 20 => syntax error
```

Demonstrating "-I" option

The function prototype "int mpritf(char *, …)" is defined in "minilib.h" stored in "include" directory. The header file "minilib.h" defines a few basic IO interfaces, and it is listed as:

```
int getchar(void);
void putchar(char c);
char *gets(char *st);
void PutString(char *s);
char Nibble2Hex(int c);
void PutInt(int n);
void PutLong(long a);
int mprintf(char *fmt, ...);
long GetInt(char *s, int n);
void SetBaudrate(long b);
void LightLed(int c);
```

So we can include this header file in source C file instead of declare this prototype explicitly. The file "mycode.c" after modification looks like:

```
#include <minilib.h>
int main(int agc, char **agv)
{
 int b;
 int a;
 a=1;
 b=2;
 mprintf("end: %i\n", a+b);
 return(a+b);
}
```

And the command line compilation is

```
C> cc105 –I..\include mycode.c –o mycode.asm
```

The same results obtained. The generated assembly code is stored in “mycode.asm”

```
.CODE

FUNC A0_main:
MOV.w  R4, 0x0001
ST.w  R4, 0xa(R0)
MOV.w  R5, 0x0002
ST.w  R5, 0x8(R0)
ADD  R4, R5
ST.w  R4, 0xc(R0)
MOV.l  R4, @X$ccc000
ADD  R4, R1
ST.l  R4, 0x10(R0)
LD.w  R4, 0xc(R0)
ST.w  R4, 0x14(R0)
PUSH  R0
ADD.l R0, 0x10
CALL A0_mprintf
POP  R0
MOV  R4, R3
LD.w  R4, 0xa(R0)
LD.w  R5, 0x8(R0)
ADD  R4, R5
MOV  R3, R4
RET

.DATA
X$ccc000    DT "end: %i\n"
```

Inline Assembly Code

BR105 compiler supports inline assembly code embedded in C code. Here is an example.

```
int getchar(void)
{
 _asm {

LIB_Wait_getchar:
 IN    Re, [0x3c]
 AND.w Re, 0x1000
 BZ    LIB_Wait_getchar
 IN    R3, [0x34]
 AND.w R3, 0x00ff
 }
}
```

In this C language function "get_char", the key word "_asm" is used to define a piece of inline assembly code between "{" and "}". When cc105 processes this inline code, it just copies and inserts it in a proper location, as if nothing actually happens to other C statements. This feature is very powerful, and it can do the following tasks.
[1] Access to IO ports. It is the best way to access IO space by using this way. As BR105 compiler does not emit IO instructions, therefore, the C code has no way to access IO space. If users want to access IO space, the only way is through system provided IO library. In this example, "get_char", it reads a char from UART and returns the obtained char to a higher-level programs.
[2] Access to the passed in parameters. As R0 is the frame pointer of the local stack for this routine, one can access to the parameters stored in local stack through the base address R0. Users can handle those parameter directly inside the inline assembly code. Nevertheless, the C code can access the passed in parameters much more straightforward than inline assembly code.
[3] Set a return value. As R3 is configured as a return value by the BR105 compiler, so users can take this advantage that assign a return value to R3 before exit the function.

An example about system debug is introduced in "quick start" manual. Users can see more techniques about this mixed language programming.

Structure and Union

BR105 supports complex data types, such as user-defined structure and union data type. The structure and Union data structure can be defined nested. Users can feel free to use them to create more sophisticated programs.

Recursive

BR105 architecture and compiler support recursive function calls.

Subset of ANSI C

Unsigned

CC105 does not support “unsigned” keyword. That is all integer number declared in the source code should be signed number. The following data type are not supported.

unsigned char
unsigned int
unsigned long

The way to work round this limit is to use “int” to handle 8-bit unsigned char calculation, and use “long” to handle “16-bit” unsigned integer calculation. This will make sure no overflow takes place. However, if users are aware of their program and treat their “char” and “int” type naturally as signed numbers, it will be efficient to declare 8-bit, 16-bit, and 32-bit integers in the following.

char
int
long

For string literals and ASCII chars, their maximum number is 127. It is pretty safe and efficient to use “char” to declare them.

Enumerate

CC105 does not support “enumerate” data type. If this keyword appears in the C source code, the compiler will report an error message.

Multi-dimensional array and multi-pointer

Multi-dimensional arrays are not supported. For example, the compiler will produce an error message when the following declaration is encountered.

int a[12][4];

To work round this problem. It is user’s up to convert multi-dimensional array to single-dimensional array. For example, we can covert the above declaration to the following.

int a[48];

The following control can be applied to achieve user’s purpose for index control.

for (i=0; i<12; i++) for (j=0; j<4; j++) a[i*12 + j] = f(i, j);

The multi-indirect addressing is supported by CC105. For example the following program using the pointers is supported by CC105

```
#include <minilib.h>

int **a;
```

```
void main(void)
{
  int *i, j;

  j = 0x5555;
  i = &j;
  a = &i;
  mprintf("**a=0x%i\n", **a);
}
```

This program will produce the very beginning integer: 0x5555.

Initialization of Local Variables

Initialization of local variables in the declaration section is not supported. All the initialization of the local variables should be explicitly implemented by assignment statement. For example, the flowing initialization of integer "j" is not supported. (note: it's been supported since @ 2018)

```
void main(void)
{
  int *i, j=0x5555;

//  j = 0x5555;
  i = &j;
  a = &i;
  mprintf("**a=0x%i\n", **a);
}
```

Nevertheless, for global variables, their initialization can be implemented in both declaration section and main body section through assignment statements, as we will see in DCT example.

Pointer to Function

CC105 supports function pointer variables. If the pointer to a function is a member of complex data type, whose reference will have to be figured out through address calculation, then the invocation of the functional call should be explicitly presented in a separate statement. For example, the following two statements should be arranged in order to get the function launched.

```
typedef struct
{
 void (* pTask)(void);
 int Delay;
 int Period;
 int RunMe;
}sTask;
```

```
void SCH_Dispatch_Tasks(void)
{
  int Index;
  int Update_again;
  void (*pfn)(void);
…
  pfn = SCH_tasks_G[Index].pTask;
  pfn();
…
}
```

In CC105, the declared variable “pfn” is required to make the function call explicitly. Otherwise, the “SCH_tasks_G[Index].pTask” alone will not make this happen.

Data type conversion and constant convention

CC105 compiler provides data type conversion between integers and floating points. If mixed data types are used in an expression, explicit cast is required in the statement. Otherwise, the compiler won’t do data conversion, instead, it reports an error message. For example, to assign an integer to a floating-point number, the following code is correct.
X = (float)(K + 5);

If a constant appears in an expression, the constant will be flexible with it data type. It appears as a float when operated with a float variable; it appears as an integer when operated with an integer variable, even if itself is a real number. For example

K = K + 1.5 is equivalent to K = K + 1, if K is declared as an integer.
X = X + 1 is equivalent to X = X + 1.0, if X is declared as a float.

Preprocessor

CC105 uses free C preprocessor provided by “Daniel Stenberg”. Although it supports a lot of features, we just use some basic ones of platform independent. Users can use macro definitions and other commonly used techniques in header file and C source file. CC105 will not process directives such as “#progma”. Care should be taken to harness the language preprocessor.

C lib

CC105 provided a set of C interface and it comes with the development kit. Currently, they are “minilib”, “spi”, and “i2c”. There are many more libraries can be created by interested users. They can explore and learn computer system using this way freely.

Boolean

CC105 does not support boolean data type. Instead of dealing with boolean value, the boolean expression is designed as a jump action. In dealing with !(relation op, var, const), this NOT operation is the same with standard syntax; In dealing with logical operation !!exp(&&,||), the NOT operation is written as “!!”. eg. if (!!((a<b) || (c>100))) { printf(“abc”); }

Appendix

A more detailed piece of code for this statement convention is listed below:

```
long err_dir, ocl_dir;

err_dir = 80;

ocl_dir = -17;

if (!!((ocl_dir<0 && err_dir<ocl_dir) || (ocl_dir>0 && err_dir>ocl_dir))) {

  br_outport(0, 0x55aa);  //ok

} else br_outport(0, -1);

if ((ocl_dir<0 && err_dir<ocl_dir) || (ocl_dir>0 && err_dir>ocl_dir)) {

  br_outport(4, 0x55aa);

} else br_outport(4, -1); //ok

if ((ocl_dir<0 && err_dir<ocl_dir)) {

  br_outport(8, 0x55aa);

} else br_outport(8, -1); //ok

if (!!((ocl_dir<0 && err_dir<ocl_dir))) {

  br_outport(0x0c, 0x55aa); //ok

} else br_outport(0x0c, -1);

if (!!(ocl_dir>0 && err_dir>ocl_dir)) {

  br_outport(0x10, 0x55aa); //ok

} else br_outport(0x10, -1);
```