GCC-Inline-Assembly-HOWTO

Sandeep.S

v0.1, 01 March 2003.

This HOWTO explains the use and usage of the inline assembly feature provided by GCC. There are only two prerequisites for reading this article, and that’s obviously a basic knowledge of x86 assembly language and C.

1. Introduction.

- 1.1 Copyright and License.
- 1.2 Feedback and Corrections.
- 1.3 Acknowledgments.

2. Overview of the whole thing.

3. GCC Assembler Syntax.

4. Basic Inline.

5. Extended Asm.

- 5.1 Assembler Template.
- 5.2 Operands.
- 5.3 Clobber List.
- 5.4 Volatile ...?

6. More about constraints.

- 6.1 Commonly used constraints.
- 6.2 Constraint Modifiers.

7. Some Useful Recipes.

8. Concluding Remarks.

9. References.

1. Introduction.

1.1 Copyright and License.

1.2 Feedback and Corrections.

Kindly forward feedback and criticism to Sandeep.S. I will be indebted to anybody who points out errors and inaccuracies in this document; I shall rectify them as soon as I am informed.

1.3 Acknowledgments.

I express my sincere appreciation to GNU people for providing such a great feature. Thanks to Mr.Pramode C E for all the helps he did. Thanks to friends at the Govt Engineering College, Trichur for their moral-support and cooperation, especially to Nisha Kurur and Sakeeb S. Thanks to my dear teachers at Govt Engineering College, Trichur for their cooperation.

Additionally, thanks to Phillip, Brennan Underwood and colin@nyx.net; Many things here are shamelessly stolen from their works.

---

2. Overview of the whole thing.

We are here to learn about GCC inline assembly. What this inline stands for?

We can instruct the compiler to insert the code of a function into the code of its callers, to the point where actually the call is to be made. Such functions are inline functions. Sounds similar to a Macro? Indeed there are similarities.

What is the benefit of inline functions?

This method of inlining reduces the function-call overhead. And if any of the actual argument values are constant, their known values may permit simplifications at compile time so that not all of the inline function's code needs to be included. The effect on code size is less predictable, it depends on the particular case. To declare an inline function, we've to use the keyword `inline` in its declaration.

Now we are in a position to guess what is inline assembly. Its just some assembly routines written as inline functions. They are handy, speedy and very much useful in system programming. Our main focus is to study the basic format and usage of (GCC) inline assembly functions. To declare inline assembly functions, we use the keyword `asm`.

Inline assembly is important primarily because of its ability to operate and make its output visible on C variables. Because of this capability, "asm" works as an interface between the assembly instructions and the "C" program that contains it.

---

3. GCC Assembler Syntax.

GCC, the GNU C Compiler for Linux, uses **AT&T/UNIX** assembly syntax. Here we’ll be using AT&T syntax for assembly coding. Don’t worry if you are not familiar with AT&T syntax, I will teach you. This is quite different from Intel syntax. I shall give the major differences.

1. Source-Destination Ordering.

   The direction of the operands in AT&T syntax is opposite to that of Intel. In Intel syntax the first operand is the destination, and the second operand is the source whereas in AT&T syntax the first operand is the source and the second operand is the destination. ie,

   "Op-code dst src" in Intel syntax changes to

   "Op-code src dst" in AT&T syntax.

2. Register Naming.

   Register names are prefixed by % ie, if eax is to be used, write %eax.

3. Immediate Operand.

   AT&T immediate operands are preceded by ’$’. For static "C" variables also prefix a ’$’. In Intel syntax, for hexadecimal constants an ’h’ is suffixed, instead of that, here we prefix ’0x’ to the constant. So, for hexadecimals, we first see a ’$’, then ’0x’ and finally the constants.

4. Operand Size.

   In AT&T syntax the size of memory operands is determined from the last character of the op-code name. Op-code suffixes of ’b’, ’w’, and ’l’ specify byte(8-bit), word(16-bit), and long(32-bit) memory references. Intel syntax accomplishes this by prefixing memory operands (not the op-codes) with ’byte ptr’, ’word ptr’, and ’dword ptr’.

   Thus, Intel "mov al, byte ptr foo" is "movb foo, %al" in AT&T syntax.

5. Memory Operands.

   In Intel syntax the base register is enclosed in ’[’ and ’]’ where as in AT&T they change to ’(’ and ’)’. Additionally, in Intel syntax an indirect memory reference is like

   section:[base + index*scale + disp], which changes to

   section:disp(base, index, scale) in AT&T.

   One point to bear in mind is that, when a constant is used for disp/scale, ’$’ shouldn’t be prefixed.

Now we saw some of the major differences between Intel syntax and AT&T syntax. I’ve wrote only a few of them. For a complete information, refer to GNU Assembler documentations. Now we’ll look at some examples for better understanding.

```
+------------------------------+------------------------------------+
|       Intel Code             |      AT&T Code                     |
+------------------------------+------------------------------------+
| mov     eax,1                |  movl    $1,%eax                   |
| mov     ebx,0ffh             |  movl    $0xff,%ebx                |
| int     80h                  |  int     $0x80                     |
| mov     ebx, eax             |  movl    %eax, %ebx                |
| mov     eax,[ecx]             |  movl    (%ecx),%eax             |
| mov     eax,[ebx+3]           |  movl    3(%ebx),%eax            |
| mov     eax,[ebx+20h]         |  movl    0x20(%ebx),%eax         |
| add     eax,[ebx+ecx*2h]      |  addl    (%ebx,%ecx,0x2),%eax    |
| lea     eax,[ebx+ecx]         |  leal    (%ebx,%ecx),%eax        |
| sub     eax,[ebx+ecx*4h-20h]  |  subl    -0x20(%ebx,%ecx,0x4),%eax |
+------------------------------+------------------------------------+
```

4. Basic Inline.

The format of basic inline assembly is very much straight forward. Its basic form is

`asm("assembly code");`

Example.

```
asm("movl %ecx %eax"); /* moves the contents of ecx to eax */
__asm__("movb %bh (%eax)"); /*moves the byte from bh to the memory pointed by eax */
```

You might have noticed that here I've used `asm` and `__asm__`. Both are valid. We can use `__asm__` if the keyword `asm` conflicts with something in our program. If we have more than one instructions, we write one per line in double quotes, and also suffix a '\n' and '\t' to the instruction. This is because gcc sends each instruction as a string to **as**(GAS) and by using the newline/tab we send correctly formatted lines to the assembler.

Example.

```
 __asm__ ("movl %eax, %ebx\n\t"
          "movl $56, %esi\n\t"
          "movl %ecx, $label(%edx,%ebx,$4)\n\t"
          "movb %ah, (%ebx)");
```

If in our code we touch (ie, change the contents) some registers and return from asm without fixing those changes, something bad is going to happen. This is because GCC have no idea about the changes in the register contents and this leads us to trouble, especially when compiler makes some optimizations. It will suppose that some register contains the value of some variable that we might have changed without informing GCC, and it continues like nothing happened. What we can do is either use those instructions having no side effects or fix things when we quit or wait for something to crash. This is where we want some extended functionality. Extended asm provides us with that functionality.

5. Extended Asm.

In basic inline assembly, we had only instructions. In extended assembly, we can also specify the operands. It allows us to specify the input registers, output registers and a list of clobbered registers. It is not mandatory to specify the registers to use, we can leave that head ache to GCC and that probably fit into GCC's optimization scheme better. Anyway the basic format is:

```
       asm ( assembler template
           : output operands                  /* optional */
           : input operands                   /* optional */
```

```
             : list of clobbered registers      /* optional */
             );
```

The assembler template consists of assembly instructions. Each operand is described by an operand-constraint string followed by the C expression in parentheses. A colon separates the assembler template from the first output operand and another separates the last output operand from the first input, if any. Commas separate the operands within each group. The total number of operands is limited to ten or to the maximum number of operands in any instruction pattern in the machine description, whichever is greater.

If there are no output operands but there are input operands, you must place two consecutive colons surrounding the place where the output operands would go.

Example:

```
        asm ("cld\n\t"
             "rep\n\t"
             "stosl"
             : /* no output registers */
             : "c" (count), "a" (fill_value), "D" (dest)
             : "%ecx", "%edi"
             );
```

Now, what does this code do? The above inline fills the `fill_value` `count` times to the location pointed to by the register `edi`. It also says to gcc that, the contents of registers `eax` and `edi` are no longer valid. Let us see one more example to make things more clearer.

```

        int a=10, b;
        asm ("movl %1, %%eax;
              movl %%eax, %0;"
             :"=r"(b)        /* output */
             :"r"(a)         /* input */
             :"%eax"         /* clobbered register */
             );
```

Here what we did is we made the value of ’b’ equal to that of ’a’ using assembly instructions. Some points of interest are:

- "b" is the output operand, referred to by %0 and "a" is the input operand, referred to by %1.
- "r" is a constraint on the operands. We’ll see constraints in detail later. For the time being, "r" says to GCC to use any register for storing the operands. output operand constraint should have a constraint modifier "=". And this modifier says that it is the output operand and is write-only.
- There are two %’s prefixed to the register name. This helps GCC to distinguish between the operands and registers. operands have a single % as prefix.
- The clobbered register %eax after the third colon tells GCC that the value of %eax is to be modified inside "asm", so GCC won’t use this register to store any other value.

When the execution of "asm" is complete, "b" will reflect the updated value, as it is specified as an output operand. In other words, the change made to "b" inside "asm" is supposed to be reflected outside the "asm".

Now we may look each field in detail.

5.1 Assembler Template.

The assembler template contains the set of assembly instructions that gets inserted inside the C program. The format is like: either each instruction should be enclosed within double quotes, or the entire group of instructions should be within double quotes. Each instruction should also end with a delimiter. The valid delimiters are newline(\n) and semicolon(;). ’\n’ may be followed by a tab(\t). We know the reason of newline/tab, right?. Operands corresponding to the C expressions are represented by %0, %1 ... etc.

5.2 Operands.

C expressions serve as operands for the assembly instructions inside "asm". Each operand is written as first an operand constraint in double quotes. For output operands, there’ll be a constraint modifier also within the quotes and then follows the C expression which stands for the operand. ie,

"constraint" (C expression) is the general form. For output operands an additional modifier will be there. Constraints are primarily used to decide the addressing modes for operands. They are also used in specifying the registers to be used.

If we use more than one operand, they are separated by comma.

In the assembler template, each operand is referenced by numbers. Numbering is done as follows. If there are a total of n operands (both input and output inclusive), then the first output operand is numbered 0, continuing in increasing order, and the last input operand is numbered n-1. The maximum number of operands is as we saw in the previous section.

Output operand expressions must be lvalues. The input operands are not restricted like this. They may be expressions. The extended asm feature is most often used for machine instructions the compiler itself does not know as existing ;-). If the output expression cannot be directly addressed (for example, it is a bit-field), our constraint must allow a register. In that case, GCC will use the register as the output of the asm, and then store that register contents into the output.

As stated above, ordinary output operands must be write-only; GCC will assume that the values in these operands before the instruction are dead and need not be generated. Extended asm also supports input-output or read-write operands.

So now we concentrate on some examples. We want to multiply a number by 5. For that we use the instruction `lea`.

```
        asm ("leal (%1,%1,4), %0"
             : "=r" (five_times_x)
             : "r" (x)
             );
```

Here our input is in ’x’. We didn’t specify the register to be used. GCC will choose some register for input, one for output and does what we desired. If we want the input and output to reside in the same register, we can instruct GCC to do so. Here we use those types of read-write operands. By specifying proper constraints, here we do it.

```
        asm ("leal (%0,%0,4), %0"
             : "=r" (five_times_x)
             : "0" (x)
             );
```

Now the input and output operands are in the same register. But we don’t know which register. Now if we want to specify that also, there is a way.

```
asm ("leal (%%ecx,%%ecx,4), %%ecx"
     : "=c" (x)
     : "c" (x)
     );
```

In all the three examples above, we didn’t put any register to the clobber list. why? In the first two examples, GCC decides the registers and it knows what changes happen. In the last one, we don’t have to put `ecx` on the clobberlist, gcc knows it goes into x. Therefore, since it can know the value of `ecx`, it isn’t considered clobbered.

5.3 Clobber List.

Some instructions clobber some hardware registers. We have to list those registers in the clobber-list, ie the field after the third ’**:**’ in the asm function. This is to inform gcc that we will use and modify them ourselves. So gcc will not assume that the values it loads into these registers will be valid. We shoudn’t list the input and output registers in this list. Because, gcc knows that "asm" uses them (because they are specified explicitly as constraints). If the instructions use any other registers, implicitly or explicitly (and the registers are not present either in input or in the output constraint list), then those registers have to be specified in the clobbered list.

If our instruction can alter the condition code register, we have to add "cc" to the list of clobbered registers.

If our instruction modifies memory in an unpredictable fashion, add "memory" to the list of clobbered registers. This will cause GCC to not keep memory values cached in registers across the assembler instruction. We also have to add the **volatile** keyword if the memory affected is not listed in the inputs or outputs of the asm.

We can read and write the clobbered registers as many times as we like. Consider the example of multiple instructions in a template; it assumes the subroutine _foo accepts arguments in registers `eax` and `ecx`.

```
asm ("movl %0,%%eax;
      movl %1,%%ecx;
      call _foo"
     : /* no outputs */
     : "g" (from), "g" (to)
     : "eax", "ecx"
     );
```

5.4 Volatile ...?

If you are familiar with kernel sources or some beautiful code like that, you must have seen many functions declared as `volatile` or `__volatile__` which follows an `asm` or `__asm__`. I mentioned earlier about the keywords `asm` and `__asm__`. So what is this `volatile`?

If our assembly statement must execute where we put it, (i.e. must not be moved out of a loop as an optimization), put the keyword `volatile` after asm and before the ()’s. So to keep it from moving, deleting and all, we declare it as

```
asm volatile ( ... : ... : ... : ...);
```

Use `__volatile__` when we have to be verymuch careful.

If our assembly is just for doing some calculations and doesn’t have any side effects, it’s better not to use the keyword `volatile`. Avoiding it helps gcc in optimizing the code and making it more beautiful.

In the section `Some Useful Recipes`, I have provided many examples for inline asm functions. There we can see the clobber-list in detail.

---

6. More about constraints.

By this time, you might have understood that constraints have got a lot to do with inline assembly. But we’ve said little about constraints. Constraints can say whether an operand may be in a register, and which kinds of register; whether the operand can be a memory reference, and which kinds of address; whether the operand may be an immediate constant, and which possible values (ie range of values) it may have.... etc.

6.1 Commonly used constraints.

There are a number of constraints of which only a few are used frequently. We’ll have a look at those constraints.

1. **Register operand constraint(r)**

   When operands are specified using this constraint, they get stored in General Purpose Registers(GPR). Take the following example:

   ```
   asm ("movl %%eax, %0\n" :"=r"(myval));
   ```

   Here the variable myval is kept in a register, the value in register `eax` is copied onto that register, and the value of `myval` is updated into the memory from this register. When the "r" constraint is specified, gcc may keep the variable in any of the available GPRs. To specify the register, you must directly specify the register names by using specific register constraints. They are:

   ```
           +---+--------------------+
           | r |    Register(s)     |
           +---+--------------------+
           | a |   %eax, %ax, %al   |
           | b |   %ebx, %bx, %bl   |
           | c |   %ecx, %cx, %cl   |
           | d |   %edx, %dx, %dl   |
           | S |   %esi, %si        |
           | D |   %edi, %di        |
           +---+--------------------+
   ```

2. **Memory operand constraint(m)**

   When the operands are in the memory, any operations performed on them will occur directly in the memory location, as opposed to register constraints, which first store the value in a register to be modified and then write it back to the memory location. But register constraints are usually used only when they are absolutely necessary for an instruction or they significantly speed up the process. Memory constraints can be used most efficiently in cases where a C variable needs to be updated inside "asm" and you really don’t want to use a register to hold its value. For example, the value of idtr is stored in the memory location loc:

   ```
   asm("sidt %0\n" : :"m"(loc));
   ```

3. **Matching(Digit) constraints**

   In some cases, a single variable may serve as both the input and the output operand. Such cases may be specified in "asm" by using matching constraints.

```
asm ("incl %0" :"=a"(var):"0"(var));
```

We saw similar examples in operands subsection also. In this example for matching constraints, the register %eax is used as both the input and the output variable. var input is read to %eax and updated %eax is stored in var again after increment. "0" here specifies the same constraint as the 0th output variable. That is, it specifies that the output instance of var should be stored in %eax only. This constraint can be used:

- In cases where input is read from a variable or the variable is modified and modification is written back to the same variable.
- In cases where separate instances of input and output operands are not necessary.

The most important effect of using matching restraints is that they lead to the efficient use of available registers.

Some other constraints used are:

1. "m" : A memory operand is allowed, with any kind of address that the machine supports in general.
2. "o" : A memory operand is allowed, but only if the address is offsettable. ie, adding a small offset to the address gives a valid address.
3. "V" : A memory operand that is not offsettable. In other words, anything that would fit the `m' constraint but not the `o'constraint.
4. "i" : An immediate integer operand (one with constant value) is allowed. This includes symbolic constants whose values will be known only at assembly time.
5. "n" : An immediate integer operand with a known numeric value is allowed. Many systems cannot support assembly-time constants for operands less than a word wide. Constraints for these operands should use 'n' rather than 'i'.
6. "g" : Any register, memory or immediate integer operand is allowed, except for registers that are not general registers.

Following constraints are x86 specific.

1. "r" : Register operand constraint, look table given above.
2. "q" : Registers a, b, c or d.
3. "I" : Constant in range 0 to 31 (for 32-bit shifts).
4. "J" : Constant in range 0 to 63 (for 64-bit shifts).
5. "K" : 0xff.
6. "L" : 0xffff.
7. "M" : 0, 1, 2, or 3 (shifts for lea instruction).
8. "N" : Constant in range 0 to 255 (for out instruction).
9. "f" : Floating point register
10. "t" : First (top of stack) floating point register
11. "u" : Second floating point register
12. "A" : Specifies the `a' or `d' registers. This is primarily useful for 64-bit integer values intended to be returned with the `d' register holding the most significant bits and the `a' register holding the least significant bits.

6.2 Constraint Modifiers.

While using constraints, for more precise control over the effects of constraints, GCC provides us with constraint modifiers. Mostly used constraint modifiers are

1. "=" : Means that this operand is write-only for this instruction; the previous value is discarded and replaced by output data.

2. "&" : Means that this operand is an earlyclobber operand, which is modified before the instruction is finished using the input operands. Therefore, this operand may not lie in a register that is used as an input operand or as part of any memory address. An input operand can be tied to an earlyclobber operand if its only use as an input occurs before the early result is written.

   The list and explanation of constraints is by no means complete. Examples can give a better understanding of the use and usage of inline asm. In the next section we'll see some examples, there we'll find more about clobber-lists and constraints.

7. Some Useful Recipes.

Now we have covered the basic theory about GCC inline assembly, now we shall concentrate on some simple examples. It is always handy to write inline asm functions as MACRO's. We can see many asm functions in the kernel code. (/usr/src/linux/include/asm/*.h).

1. First we start with a simple example. We'll write a program to add two numbers.

```
int main(void)
{
        int foo = 10, bar = 15;
        __asm__ __volatile__("addl  %%ebx,%%eax"
                             :"=a"(foo)
                             :"a"(foo), "b"(bar)
                             );
        printf("foo+bar=%d\n", foo);
        return 0;
}
```

   Here we insist GCC to store foo in %eax, bar in %ebx and we also want the result in %eax. The '=' sign shows that it is an output register. Now we can add an integer to a variable in some other way.

```
 __asm__ __volatile__(
                      "   lock       ;\n"
                      "   addl %1,%0 ;\n"
                      : "=m"  (my_var)
                      : "ir"  (my_int), "m" (my_var)
                      :                                 /* no clobber-list */
                      );
```

   This is an atomic addition. We can remove the instruction 'lock' to remove the atomicity. In the output field, "=m" says that my_var is an output and it is in memory. Similarly, "ir" says that, my_int is an integer and should reside in some register (recall the table we saw above). No registers are in the clobber list.

2. Now we'll perform some action on some registers/variables and compare the value.

```
 __asm__ __volatile__(  "decl %0; sete %1"
                      : "=m" (my_var), "=q" (cond)
                      : "m" (my_var)
                      : "memory"
                      );
```

Here, the value of my_var is decremented by one and if the resulting value is `0` then, the variable cond is set. We can add atomicity by adding an instruction "lock;\n\t" as the first instruction in assembler template.

In a similar way we can use "incl %0" instead of "decl %0", so as to increment my_var.

Points to note here are that (i) my_var is a variable residing in memory. (ii) cond is in any of the registers eax, ebx, ecx and edx. The constraint "=q" guarantees it. (iii) And we can see that memory is there in the clobber list. ie, the code is changing the contents of memory.

3. How to set/clear a bit in a register? As next recipe, we are going to see it.

```
__asm__ __volatile__(   "btsl %1,%0"
                      : "=m" (ADDR)
                      : "Ir" (pos)
                      : "cc"
                      );
```

Here, the bit at the position ’pos’ of variable at ADDR (a memory variable) is set to `1` We can use ’btrl’ for ’btsl’ to clear the bit. The constraint "Ir" of pos says that, pos is in a register, and it’s value ranges from 0-31 (x86 dependant constraint). ie, we can set/clear any bit from 0th to 31st of the variable at ADDR. As the condition codes will be changed, we are adding "cc" to clobberlist.

4. Now we look at some more complicated but useful function. String copy.

```
static inline char * strcpy(char * dest,const char *src)
{
int d0, d1, d2;
__asm__ __volatile__(  "1:\tlodsb\n\t"
                       "stosb\n\t"
                       "testb %%al,%%al\n\t"
                       "jne 1b"
                     : "=&S" (d0), "=&D" (d1), "=&a" (d2)
                     : "0" (src),"1" (dest)
                     : "memory");
return dest;
}
```

The source address is stored in esi, destination in edi, and then starts the copy, when we reach at **0**, copying is complete. Constraints "&S", "&D", "&a" say that the registers esi, edi and eax are early clobber registers, ie, their contents will change before the completion of the function. Here also it’s clear that why memory is in clobberlist.

We can see a similar function which moves a block of double words. Notice that the function is declared as a macro.

```
#define mov_blk(src, dest, numwords) \
__asm__ __volatile__ (                                           \
                       "cld\n\t"                                  \
                       "rep\n\t"                                  \
                       "movsl"                                    \
                       :                                          \
                       : "S" (src), "D" (dest), "c" (numwords)  \
                       : "%ecx", "%esi", "%edi"                   \
                       )
```

Here we have no outputs, so the changes that happen to the contents of the registers ecx, esi and edi are side effects of the block movement. So we have to add them to the clobber list.

5. In Linux, system calls are implemented using GCC inline assembly. Let us look how a system call is implemented. All the system calls are written as macros (linux/unistd.h). For example, a system call with three arguments is defined as a macro as shown below.

```
#define _syscall3(type,name,type1,arg1,type2,arg2,type3,arg3) \
type name(type1 arg1,type2 arg2,type3 arg3) \
{ \
long __res; \
__asm__ volatile (  "int $0x80" \
                  : "=a" (__res) \
                  : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
                    "d" ((long)(arg3))); \
__syscall_return(type,__res); \
}
```

Whenever a system call with three arguments is made, the macro shown above is used to make the call. The syscall number is placed in eax, then each parameters in ebx, ecx, edx. And finally "int 0x80" is the instruction which makes the system call work. The return value can be collected from eax.

Every system calls are implemented in a similar way. Exit is a single parameter syscall and let's see how it's code will look like. It is as shown below.

```
{
        asm("movl $1,%%eax;         /* SYS_exit is 1 */
             xorl %%ebx,%%ebx;      /* Argument is in ebx, it is 0 */
             int  $0x80"            /* Enter kernel mode */
             );
}
```

The number of exit is "1" and here, it's parameter is 0. So we arrange eax to contain 1 and ebx to contain 0 and by `int $0x80`, the `exit(0)` is executed. This is how exit works.

8. Concluding Remarks.

This document has gone through the basics of GCC Inline Assembly. Once you have understood the basic concept it is not difficult to take steps by your own. We saw some examples which are helpful in understanding the frequently used features of GCC Inline Assembly.

GCC Inlining is a vast subject and this article is by no means complete. More details about the syntax's we discussed about is available in the official documentation for GNU Assembler. Similarly, for a complete list of the constraints refer to the official documentation of GCC.

And of-course, the Linux kernel use GCC Inline in a large scale. So we can find many examples of various kinds in the kernel sources. They can help us a lot.

If you have found any glaring typos, or outdated info in this document, please let us know.

9. References.

1. Brennan’s Guide to Inline Assembly
2. Using Assembly Language in Linux
3. Using as, The GNU Assembler
4. Using and Porting the GNU Compiler Collection (GCC)
5. Linux Kernel Source