

Kernel Debugging

Introduction

I am developing my hobby OS (Ace - <http://samueldotj.com/Ace/index.html>) using GCC in Windows environment. In July 2006, I decided to rewrite my OS from scratch (once again); Symbolic debugger is one of the features I wanted to add, in that rewrite. I thought GDB is huge and can't be ported to an entry level hobby OS. So I skipped the idea of porting GDB or any other debugger to my OS and started working on my own debugger. However after some posts in alt.os.development newsgroup and from GDB manual I found that it is not difficult to port GDB to my OS. So I started working on that and completed it within one day and struggled one month to make it work because of tools and a small bug. Although it is easy work, while doing that I find no detailed document for that in the net. I have a habit of documenting the work that I have done which does not have documentation (or not documented properly). So I started writing (this) one.

In this article I tried to document some of the debugging techniques I have been using during my OS development. Since Linux has most of kernel development tools and documentation, I want cover kernel debugging in Windows development environment. Although most of the GNU tools are available for Windows, they are not up to date as Linux tools and they lack documentation for windows specific issues. The following tools are I am using for my OS development are GCC (MINGW), NASM, GRUB, Bochs and QEmu. So the little techniques I know are related to these tools only.

Although this article covers debugging tools and technique for developing x86 architecture under Windows XP, I believe the techniques can be used across the different OS with the exception of tools.

-Samuel

This article is only for amateur OS developers who have just started crashing their kernel(s).

I am still working on this document – This document is not yet completed.

Static data analysis

kprintf

The easiest technique to debug an amateur kernel is printing messages to the screen on each critical action. (Most OS developer's first debugging session starts with a specialized print() statement, so it is here in the top of the list). Implementing this technique is very simple; however implementing standard C library style printf() is little harder. (Writing/implementing printf() is out of the scope of kernel development). There are lot of open source implementation of printf(); check out the libc comes with open source C compilers like GCC. Ace OS uses DJGPP's printf() implementation after some modification.

The following is the modification needed to be done to use the standard printf(). PUTC is internally used by the _DoPrint function to print a character in the screen. So the kernel must provide a function which will print the given character in the screen. In X86 architecture it is easy to write a function to use the memory mapped VGA to print the characters. You can refer the Ace OS VGA function implementation in Kernel/VGAText.c. It is not optimized however it is not dead slow.

```
void (*PUTC)(BYTE ch);
int printf(const char *fmt, ...)
{
    PUTC = VGA_PutCh;
    _DoPrint( fmt, (va_list) ((&fmt)+1));
    return 1;
}
```

src: /kernel/printf.c and VGAText.c

Advantages

- Prints critical errors in the screen
- Kernel trace

Disadvantage

- Inserting and removing debugging messages in the kernel source is not easy. However it is possible to make it easy by using macros.
- You can't modify the behavior at run time
- Limited number of messages can be displayed in the screen at a time

It is not advisable to rely on this technique for kernel debugging; if no other method is working then try this.

Exception handlers

Analyzing exceptions and taking appropriate action is the purpose of exception handlers. However most of amateur kernel which is based on the assumption all codes and design aspects are correct, the exception handler is nothing but just an iret or (cli and hlt) instruction. But halting/crashing the system on each exception without any information won't help in debugging. It is very critical to get exception details after each exception, so it also a tool in kernel debugging. This page deals with how to get maximum details on an exception.

All the exception handler and interrupt handler should be filled with default exception handler, initially. In this way all the exceptions and interrupts which don't have specific handler can be tracked. The default exception handler is responsible for displaying/logging the system state while the exception occurs. Two tasks should be accomplished by the default exception handler - catching exception details and displaying/storing them.

1) Capturing Exception Details

This part is actually the exception handler. It may be an assembly routine which should be set in the Interrupt Descriptor Table. This routine when called pushes all the register contents in the stack and calls a C routine to print the pushed values and halts the system. In Ace OS this method is implemented as assembly macro.

Src: /ace/kernel/processor.asm (DumpCPU assembly macro)

2) Printing Exception Details

This can be implemented as a c function which prints the given arguments. The following is the C code used for displaying the dumped register and exception name. This function is called by the exception handler (assembly routine).

```
char szExceptionType[TOTAL_EXCEPTIONS][50]=
{
    "Divide by zero",
    "...",
    "...",
    "SIMD Floating point",
    "Unknown Exception",
};
void DumpCPU
(
    UINT32 ExceptionType, UINT32 EBX, UINT32 ECX, UINT32 EDX,
    UINT16 ES,  UINT16 SS,  UINT16 GS,  UINT16 FS,
    UINT32 EBP,  UINT32 ESP,  UINT32 ESI,  UINT32 EDI,  UINT32 EFlags,
    UINT32 CR0,  UINT32 CR1,  UINT32 CR2,  UINT32 CR3,  UINT32 CR4,
    UINT16 DS,  UINT32 EAX,  UINT32 ErrorCode,  UINT32 EIP,  UINT32 CS
)
{
    if ( ExceptionType >= TOTAL_EXCEPTIONS )
        ExceptionType = TOTAL_EXCEPTIONS - 1;

    printf
    (
        DumpCPUFormat, EAX, EBX, ECX, EDX,
        CS, DS, ES, SS, GS, FS,
        EBP, ESP, ESI, EDI, EIP,
        CR0, CR1, CR2, CR3, CR4,
        EFlags, ErrorCode, szExceptionType[ExceptionType],
        ExceptionType
    );
}
```

Kernel Log

The above two techniques are useful to take snapshot of a particular object at a particular time/interval. But since the above two method uses the screen for printing the messages, it is impossible to read the message trace over a period of time. For this the output must be stored in permanent storage (file) rather than the screen. But the problem is most of the amateur kernel don't have a file system or the file system itself needs an analysis. So log files can't be used in the kernel debugging.

Another option is redirecting the messages to printer so that the messages can be printed on a paper. It is not a cost effective method for a hobby OS project and also difficult to trace.

The other option is sending the output to serial/parallel port, and an external program must read the port and save it in a file. Since it is easy to code serial/parallel ports and it is also easy to read a serial port and store the results in a file in Windows, Ace OS uses this method.

Concept

- printf should write the output into serial/parallel port
- A program should read the serial/parallel port and writes it to a file.

For real machine, a null modem cable is required to connect the two system and a external program needed to read the serial port.

Bochs and QEmu have options to redirect the parallel/serial port writes to a file. The following option will enable parallel port redirection in Bochs

```
parport1: enabled=1, file="file.txt"
```

For qemu, a special version is needed to support logging and other debugging options. The windows download is available in <http://www.h6.dion.ne.jp/~kazuw/qemu-win/qemu-0.8.2-windbg.zip>. Add the following command line argument while starting the qemu.exe -parallel file:file.txt. In qemu it is also possible to redirect the output to virtual console using the following option -parallel vc.

The following code is used for debug printing, in this code instead of the normal VGA text print routine, printer print routine is passed to _DoPrint so all the print will be redirected to printer port.

```
void DebugPutCh(BYTE Character)
{
    Printer_Print(0, Character);
}

int dbgprintf(const char *fmt, ...)
{
    PUTC = DebugPutCh;
    _DoPrint( fmt, (va_list) ((&fmt)+1));
    return 1;
}

src : /kernel/Printer.c
```

MACROS

It is possible to end up with thousand lines of debug output and don't know which code fragment generated which line; this will happen especially in the case of memory management routine debugging. This can be avoided by using special macros in the debug print statements. The following are some of those macros `__FILE__`, `__LINE__` and `__PRETTY_FUNCTION__`. These macros will transform into file name, line number and function name during the compilation time. These macros are supported by GCC; for other compilers there may be similar macros.

Ace OS uses the following macro for this kind of debug logs. This macro will generate debug outputs only if the macros `_DEBUG_` defined in a particular file. So you can enable debugging for a particular module and disable it for another module but just adding and removing the `#define _DEBUG_` in the module files.

```
#ifndef _DEBUG_
#define DEBUG_PRINT_INFO(Message) \
    dbgprintf("\n%s:%d:%s(): %s", __FILE__ , __LINE__, __PRETTY_FUNCTION__, Message );
#define DEBUG_PRINT_OBJECT1(Format, Object) \
    {\
        dbgprintf("\n%s:%d:%s(): ", __FILE__ , __LINE__, __PRETTY_FUNCTION__); \
        dbgprintf(Format, Object ); \
    }
#else
#define DEBUG_PRINT_INFO(Message)
#define DEBUG_PRINT_OBJECT1(Format, Object1)

#endif
```

This will produce output like the following

```
Thread.c:84:CreateThread(): ESP0 = 80009FAF ESP1 = 80007FF3
Thread.c:109:CreateThread(): lpThread->IOBASE 8000A000
```

src : kernel/include/Util.h

Dynamic system analysis

Monitor / Debuggers in Emulator

This method is not applicable to real machines; it is only for emulator sessions and only applicable to bochs and qemu emulators. Bochs and QEmu have built-in support for debugging. The debugger/monitor available in the emulator let OS developers to examine the system state without any extra code. These emulators also support break points, examining memory, modifying memory/registers etc.

Debugging with these emulators is very easy when compare to the previous techniques since it is possible examine all the variables, registers, virtual to physical memory translation without adding any extra code and without system restart. But it requires some Intel assembly syntax to use these debuggers.

These debuggers are not aware of any symbols, so it is not enough to know only the name of the variable/function that should be analyzed but it is required to know the address. The following command will give the relative address of the given variable/function.

```
objdump -d <your kernel file> | find "FunctionName" /i
```

The virtual or physical base address should be added to the above returned address to make it valid.

Bochs has a separate executable for debugging purpose – bochsdbg.exe. This debugger is not included in the default download; it should be downloaded separately. This debugger is slower than the normal emulator (bochs.exe), so it should be used only during debugging sessions. The command line options are same for both the executables. The following are the some important commands available in bochsdbg.

c	continues the execution
s	execute the number instructions specified
vb	sets a virtual address instruction breakpoint
pb	sets a physical address instruction breakpoint
d	deletes a break point
x	examine memory at linear address
xp	examine memory at physical address
disas	disassemble
dump_cpu	displays register details
info gdt	displays GDT entries
info idt	displays IDT entries

Checkout the bochs manual for complete list or type help in the bochsdbg command prompt.

After starting the bochsdbg.exe it will open a command window. This is similar to Qemu monitor virtual console. You can set the break points and continue the execution by using vb/pb and c commands. If you want to stop the emulation in the middle of emulation and examine the state of the machine, press `ctl+c` in the command window (not in the emulation window) it will stop the emulation and give command prompt.

Qemu comes with a monitor which is available as a virtual console (default is 2). To switch between virtual consoles press `alt+ctl+<number>`. The commands are similar to bochs. You can get help by typing `help` or `?`. (Pressing `TAB` key will list the options you have). Unlike bochs you can examine the state of the machine without stopping the emulation. However in most situations it is required to stop the emulation to see the contents of a data structure or register. For that you can use `stop` and `cont` commands in the qemu.

GDB – Gnu Symbolic Debugger

Summary of GDB (Taken from GDB manual)

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another

GDB Commands

There are lot and lot of commands available within the GDB. Also those commands are well documented, so it is not listed here. You can get the GDB documentation from the following link - <http://www.gnu.org/software/gdb/documentation/>.

How to use GDB for Kernel Debugging

GDB is designed for application debugging; however its design allows to debug any program (application, system or embedded). For kernel debugging we are interested only in the remote debugging techniques provided by GDB. Remote debugging in the sense that the actual program which needed to be debugged will run in another (remote) machine and the debugger will run in some other machine. GDB can communicate over the remote program using serial port or network cables (but for kernel debugging we use only null modem cable and not network). So for GDB kernel debugging it is better to have two machines connected using null modem cable. You can get more details about null modem cable from http://en.wikipedia.org/wiki/Null_modem. However we can have only one machine and emulate another machine and null modem cable connection using virtualization software.

Virtual Serial Ports

The following free windows utilities provide pair(s) of virtual serial port. The byte written in one of virtual serial port will be received in the next virtual serial port. Thus these utilities remove the requirement for the two systems with serial port and a null modem cable for remote debugging.

1. MixW Serial Port Bridge (<http://www.mixw.co.uk/download/download.htm>)
2. N8VB vCOM Virtual Serial Ports (<http://www.philcovington.com/SDR.html>)

I have used N8VB vCOM virtual serial ports to check my OS GDB stub. By default N8VB vCOM enables COM4 and COM5 as a pair of virtual port. If you have any hardware with COM4 and COM5 settings then it won't work, so you have change the N8VB vCOM .inf files before installing.

To enable GDB debugging, you have to include GDB stub in your kernel. GDB stub deals with communication details with the GDB server. (GDB has its own protocol for serial communication). However to make things easy GDB already has stub programs for every platform (for x86 the file is i386-stub.c in the GDB source, however the file won't compile because it misses a ;). You may need not to modify the i386-stub.c file apart from the semi-colon. However the kernel has to provide the following operating system specific functions which are referenced in the i386-stub.c

int getDebugChar() – to read single character from the serial port

void putDebugChar(int) – to write single character to the serial port

void exceptionHandler (int exception_number, void *exception_address) – set the exception handler in the exception table.

The first two routines are deals with reading and writing characters to the serial port. Initializing, Reading and writing serial ports using IO polling is easy and you can find the code for that in anywhere in internet.

```
UINT32 GDBPort = 0x3F8;
void Init_GDBStub()
{
    //set up the serial port
    InitSerialPort(GDBPort, 115200,UART_DATA_BIT_8, UART_PARITY_NONE,UART_STOP_BIT_1);
    //setup exception handlers
    set_debug_traps();
    printf("\n\rWaiting for GDB on serial port 0x%X : ", GDBPort );
    //break point exception to sync with GDB
    __asm__("int3");
    printf("connected");
}
```

```
src : ace\kernel\serial.c ace\kernel\gdb\gdb.c ace\kernel\gdb\gdb-stub.c
```

Since I added GDB support in my OS recently, you may not find the above source code in the older releases. If you will find it only in Ace 3 or later versions. You can contact me if you want the source code through email.

MinGW and Cygwin Problem

Due to some unknown reason(s) MinGW GDB is not working for serial debugging however it is works with remote TCP/IP debugging. Latest version of Cygwin GDB also crashes; however older version of Cygwin GDB works fine so I have used Cygwin for debugging purposes.

Compiling with debugging information

By default gcc will truncate the debugging information from the output file. So you have pass `-g` option to gcc. For more information see gcc manual.

GRUB

If you are using GRUB or any other multiboot compliant boot loader then you have to modify your linker script to include the debug information in appropriate location; otherwise GRUB won't boot your kernel. Ace kernel linker script is located at: `/kernel/kernel.ld`.

Starting Debugging Session

Run cygwin `gdb.exe` you will get output like the following, verify you get the last line as `This GDB was configured as "i686-pc-cygwin"`.

One of the advantage of GDB over the emulator debugger/monitor is it can recognize symbols. However to utilize this option the kernel should contain the GDB specific symbolic information. You can instruct GCC to insert debug symbols in the output using the `-g` option.

To load the symbols into gdb session you have to issue the following command.

```
(gdb) symbol-file <your file location>
```

eg:

```
(gdb) symbol-file e:/projects/ace/kernel/kernel.sys
```

To step through your source code, you can use the directory command. This will instruct gdb to retrieve source files from the specified directories.

Now you have initialize the remote debugging and start remote debugging

```
(gdb) set remoteaddresssize 32
```

```
(gdb) set remotebaud 9600
```

```
(gdb) target remote COM4
```

The `remoteaddresssize` option means the address size of your processor can address. If you are developing your OS for x86 then you should use 32. The `remotebaud` option specifies the speed at which the communication between your OS(gdb-stub) and GDB should happen. 9600 is safe but it is very slow. You can try increasing it by trail and error method. However make sure whenever you make changes this option you have change your OS `serialport_init()` function also.

Since you have to use these commands all the time you can create batch/macro/command file for this. In Linux environment gdb will execute all command in the `.gdbinit` file during start. However in windows `.gdbinit` is not a valid file name. So you have to use another batch file for this.

```
-----  
File: g.bat  
c:\cygwin\bin\gdb -q --command=gdbinit  
-----
```

```
File: gdbinit  
  
set remoteaddresssize 32  
set remotebaud 115200  
set remotedevice /dev/com4  
symbol-file e:/projects/ace/kernel/kernel.sys  
directory kernel kernel/debugger  
target remote /dev/com4  
-----
```

Now you can start gdb by just typing `g` and gdb will use initialization commands from the `gdbinit` file. After this gdb will wait for the remote program to start. Now start your OS in the remote machine / emulator. If you have inserted any breakpoint in your code(int 0), then gdb will stop at that instruction and prompt you for instructions. However inserting and removing breakpoints through code is painful, you can insert breakpoints using `gdb break` command. You can use function name or file name and line number to specify the breakpoint target.

Here is a sample debugging session on my OS.

```
Starting debugging session  
E:\Projects\Ace>c:\cygwin\bin\gdb -q --command=gdbinit  
Init_GDBStub () at gdb.c:15  
15          printf("connected");
```

```
Set breakpoint and continue  
(gdb) b InitSysCallTable  
Breakpoint 1 at 0xc000f626: file SysCall.c, line 31.  
(gdb) c  
Continuing.
```

```
Breakpoint 1, InitSysCallTable () at SysCall.c:31  
31          for(i=0;i<SYSCALL_END;i++)
```

You can print the source code any function using `list` statement.

```
(gdb) list InitSysCallTable  
28      void InitSysCallTable()  
29      {  
30          int i;  
31          for(i=0;i<SYSCALL_END;i++)  
32          {  
33              _SysCallDispatchTable[i].lpSysCall = NULL;
```

You can step through code using step command and next command.

(gdb) s

```
33      _SysCallDispatchTable[i].lpSysCall = NULL;
```

To view assembly source code you can use disassemble command

(gdb) disassemble InitSysCallTable

Dump of assembler code for function InitSysCallTable:

```
0xc000f620 <InitSysCallTable+0>:      push   %ebp
0xc000f621 <InitSysCallTable+1>:      mov    %esp,%ebp
0xc000f623 <InitSysCallTable+3>:      sub    $0x4,%esp
0xc000f626 <InitSysCallTable+6>:      movl   $0x0,0xffffffff(%ebp)
0xc000f62d <InitSysCallTable+13>:     cmpl   $0x3a,0xffffffff(%ebp)
0xc000f631 <InitSysCallTable+17>:     jle    0xc000f635 <InitSysCallTable+21>
0xc000f633 <InitSysCallTable+19>:     jmp    0xc000f658 <InitSysCallTable+56>
0xc000f635 <InitSysCallTable+21>:     mov    0xffffffff(%ebp),%eax
0xc000f638 <InitSysCallTable+24>:     movl   $0x0,0xc001d3b0(,%eax,8)
0xc000f643 <InitSysCallTable+35>:     mov    0xffffffff(%ebp),%eax
0xc000f646 <InitSysCallTable+38>:     movl   $0x0,0xc001d3b4(,%eax,8)
0xc000f651 <InitSysCallTable+49>:     lea   0xffffffff(%ebp),%eax
0xc000f654 <InitSysCallTable+52>:     incl  (%eax)
0xc000f656 <InitSysCallTable+54>:     jmp    0xc000f62d <InitSysCallTable+13>
0xc000f658 <InitSysCallTable+56>:     movl   $0xc0003c5f,0xc001d3b0
0xc000f662 <InitSysCallTable+66>:     movl   $0xc0003c69,0xc001d3b8
0xc000f66c <InitSysCallTable+76>:     movl   $0x1,0xc001d3bc
0xc000f676 <InitSysCallTable+86>:     movl   $0xc0003b90,0xc001d3c0
0xc000f680 <InitSysCallTable+96>:     movl   $0x1,0xc001d3c4
0xc000f68a <InitSysCallTable+106>:    movl   $0xc0001180,0xc001d3c8
---Type <return> to continue, or q <return> to quit---q
Quit
```

(gdb) print i

```
$2 = 0
```

(gdb) printf "%x\n", &i

```
c0053f9c
```

(gdb) x/10b 0xc0053f9c

```
0xc0053f9c:  0      0      0      0      -16     63     5      -64
0xc0053fa4:  43     25
```

(gdb) info registers

```
eax      0x1      1
ecx      0x7      7
edx      0x3d5    981
ebx      0xc0054000  -1073397760
esp      0xc0053f9c  0xc0053f9c
ebp      0xc0053fa0  0xc0053fa0
esi      0x32dde   208350
edi      0x1535f0  1390064
eip      0xc000f635  0xc000f635 <InitSysCallTable+21>
eflags   0x200397 [ CF PF AF SF TF IF ID ]
cs       0x8      8
ss       0x10     16
ds       0x10     16
es       0x10     16
fs       0x10     16
gs       0x10     16
```

Another type of break point setting

(gdb) b ckernel.c:160

```
Breakpoint 2 at 0xc000192b: file CKernel.c, line 160.
```

```
(gdb) c
```

```
Continuing.
```

```
Breakpoint 2, StartCKernel (header=0x100000, info=0x0) at CKernel.c:160  
160         printf("done");
```

You can see the type definition of a variable/expression using whatis command.

```
(gdb) whatis header
```

```
type = multiboot_header_t *
```

```
(gdb) print header
```

```
$4 = (multiboot_header_t *) 0x100000
```

```
(gdb) set print pretty on
```

```
(gdb) print *header
```

```
$5 = {  
  magic = 464367618,  
  flags = 65539,  
  checksum = 3830534139,  
  header_addr = 1048576,  
  load_addr = 1048576,  
  load_end_addr = 1183744,  
  bss_end_addr = 1390064,  
  entry_addr = 1048608  
}
```

Own Debugger

As I mentioned in the introduction, this documentation is started while I tried to create my own debugger. So I want to mention how to create a simple debugger built-in to the kernel. It is just a tips how to implement a minimal debugger (only assembly disassembling, no symbol..) . A simple debugger built-in to the kernel will be handy if there is no test system. I have not implemented this so I don't have source code for this.

Commands

A debugger should support minimal the following are commands.

- Breakpoint
 - Set Virtual Breakpoint
 - Set Physical Breakpoint
 - List Breakpoints
 - Delete Breakpoint
- Status
 - View Memory
 - View Registers
- Disassemble

Design

The debugger can be split into three modules based on the functionality.

- User Interface – To read user commands and display results
- Debugger Operations – The core of the debugger
- Kernel/Hardware Interface – Interface between debugger and the kernel or hardware.

Implementation

Invoking the Debugger

The debugger can run only in ring 0, so the only way to reach the debugger is through some call gate/interrupt. One of the ways to invoke the debugger is executing the **int 1** instruction. The debugger should have handler for the int 1 exception. The other way is to include code in the keyboard handler that should check for some special key combination and invoke the debugger code.

User Interface

Input and output for debugger should be considered as a special case in kernel. Since the debugger built-in to the kernel and it may required to be start in the early stages, the better I/O option is keyboard and text mode VGA. This input/output should not use any kernel data structure or functions. In other words, there should be a separate keyboard and text mode VGA modules for the debugger.

If you want to enable remote debugging the input and output should also has option for serial input and output. Reading from and writing to a serial port is easy, you can easily implement this. This is same as the `getDebugChar()` and `putDebugChar()` implementation in GDB.

Kernel / Hardware Interface

Debugger should able to read and write any memory and also able to some IO operations. To avoid complexity, the IO operations on keyboard and serial port should not use any special driver mechanism. For reading and writing physical memory, the debugger may use do direct mapping on the fly if there is no mapping exists. The kernel should provide some mechanism to find whether there any valid mapping exist for physical to virtual and vice versa.

Debugger Core

- Breakpoint - X86 supports hardware level breakpoints (memory address, I/O address). When the specified address executed or the specified memory, I/O accessed the processor generates breakpoint exception. But hardware level breakpoints are very limited in number.

Software breakpoints on the other hand unlimited but they will slow down the execution. There are two ways to implement software breakpoints –

- The debugger should maintain a list of breakpoints and runs the CPU in single step mode (setting the TF flag in the EFlags registers which will cause debug exception for each instruction) during each debug exception the debugger check the EIP and/or referenced address with the breakpoint list and takes appropriate action.
 - Replacing the instruction specified in the break point address with the int 1. This will cause debug exception while executing at the specified address, the debugger should write the correct instruction and resumes the execution.
- Status
 - Memory - Since the debugger runs in the context of kernel reading/writing memory values are simple pointer operations. However special care should be taken care to avoid referencing a virtual address which is not mapped in the page table. In that case the debugger should give an error message instead of faulting.
 - Registers - Reading/writing registers are also indirect form of reading writing memory. The debug exception handler dumps the registers values into the stack while entering the debug mode and restores them while returning to normal execution. Reading register values is nothing but reading the stack values, similarly writing a value into a register is nothing but writing it into the stack.
 - Disassemble – Kernel disassembler is a function which reads machine instructions from memory and output equivalent assembly code in ASCII format. There are many open source dis-assemblers present such as NDISASM. Porting them into your OS will take less than 2 hours. Another option is writing your own disassembler but it will take at least 2 weeks.

Bug?

I am neither OS expert nor good writer; if you find any error in this article, you can send a mail to me. You can also send your suggestions to my email address samueldotj@gmail.com. I will try to keep updating this document.

You can find the latest version of this article at – www.samueldotj.com/articles.asp.

The End