Debugging
Embedded Linux Kernel & Drivers

@ User Space

KERNEL MASTERS

What is Debugging?

Debugging is the process of finding and resolving defects or problems within a computer program that prevent correct operation of computer software or a system.

(Ref: Wikipedia)

Types of Builds

Engineering Build	Production Build
This build contains debug information.	This build doesn't contain any debug information.
This build doesn't support source code optimization. Because unable to debug optimized source code.	This build support source code optimization.
Size of the image more	Size of the image less
No security	source code hacking somewhat difficult.

Session 1: @ User Space

User Space	Purpose
printf	Useful for monitoring
gdb	Source level debugger
strace	strace tool that shows all the system calls issued by a user-space program
valgrind	Memory checker like segmentation fault ERRORS.
debugfs	Debug FS useful for custom debug messages.
DDT	Device Driver Test cases like gpio, uart, i2c etc.
LTP	Linux Test Project useful for Tracing of events in the kernel and also checks kernel performance.
mmap()	User directly communicating with hardware for register programming.

GDB

GNU Debugger

What is GDB?

- “GNU Debugger”
- A debugger for several languages, including C and C++
- It allows you to inspect what the program is doing at a certain point during execution.
- Errors like segmentation faults may be easier to find with the help of gdb.

The GNU Project

- The ostensible goal of the GNU Project was to create a Un*x clone without any AT&T sources
    - **G**NU's **N**ot **U**nix
- There are several projects developed that all targeted the original goal
    - First, there was the GNU C compiler (gcc) and later g++ finally manifesting itself as GCC – the Gnu Compiler Collection
        - Front ends for C, C++, Objective-C, FORTRAN, Ada and Go with associated libraries like libstdc++, etc.
    - Architected as a front-end language parser and a back-end code generator
    - Also added numerous *binutils* such as the linker, librarian, etc.
- The GNU debugger (gdb) was built as a source debugger for the GCC
    - gdb supports Ada, Assembly, C/C++, D, FORTRAN, Go, Objective-C, OpenCL, Modula-2, Pascal and Rust

GDB Command Line Debug Levels

- **`-g0`** will explicitly produce no debug information
- **`-g1`** produces minimal information, enough for making back traces, but no information about local variables and no line numbers
- **`-g2`** default debug level when not specified. Typically this will produce symbols, line numbers, etc. needed for symbolic debugging
  - This is the default for the -g option to the compiler
- **`-g3`** includes extra information, such as all the macro definitions present in the program
- And, the mac-daddy of options: **`-ggdb3`**
  - This is like **`-g3`**, but generates debugging information specifically for gdb rather than normal COFF/XCOFF or DWARF 2 of **`-g`**

Example Compile for GDB

- Example compilation to enable debugging

```
$ arm-linux-gnueabi-gcc -ggdb3 -o hello helloWorld.c
```

- Example for examining the debug info in ELF header

```
  $ arm-linux-gnueabi-objdump -h hello
  …
   24 .comment       0000002a  00000000  00000000  00000a97  2**0
                     CONTENTS, READONLY
   25 .debug_aranges 00000020  00000000  00000000  00000ac1  2**0
                     CONTENTS, READONLY, DEBUGGING
   26 .debug_pubnames 00000031  00000000  00000000  00000ae1  2**0
                     CONTENTS, READONLY, DEBUGGING
   27 .debug_info    00000179  00000000  00000000  00000b12  2**0
                     CONTENTS, READONLY, DEBUGGING
   28 .debug_abbrev  000000d4  00000000  00000000  00000c8b  2**0
                     CONTENTS, READONLY, DEBUGGING
   29 .debug_line    000003ea  00000000  00000000  00000d5f  2**0
                     CONTENTS, READONLY, DEBUGGING
   30 .debug_frame   00000090  00000000  00000000  0000114c  2**2
                     CONTENTS, READONLY, DEBUGGING
   31 .debug_str     000000ea  00000000  00000000  000011dc  2**0
                     CONTENTS, READONLY, DEBUGGING
   32 .debug_loc     00000058  00000000  00000000  000012c6  2**0
                     CONTENTS, READONLY, DEBUGGING
   33 .debug_macinfo 00009e52  00000000  00000000  0000131e  2**0
```

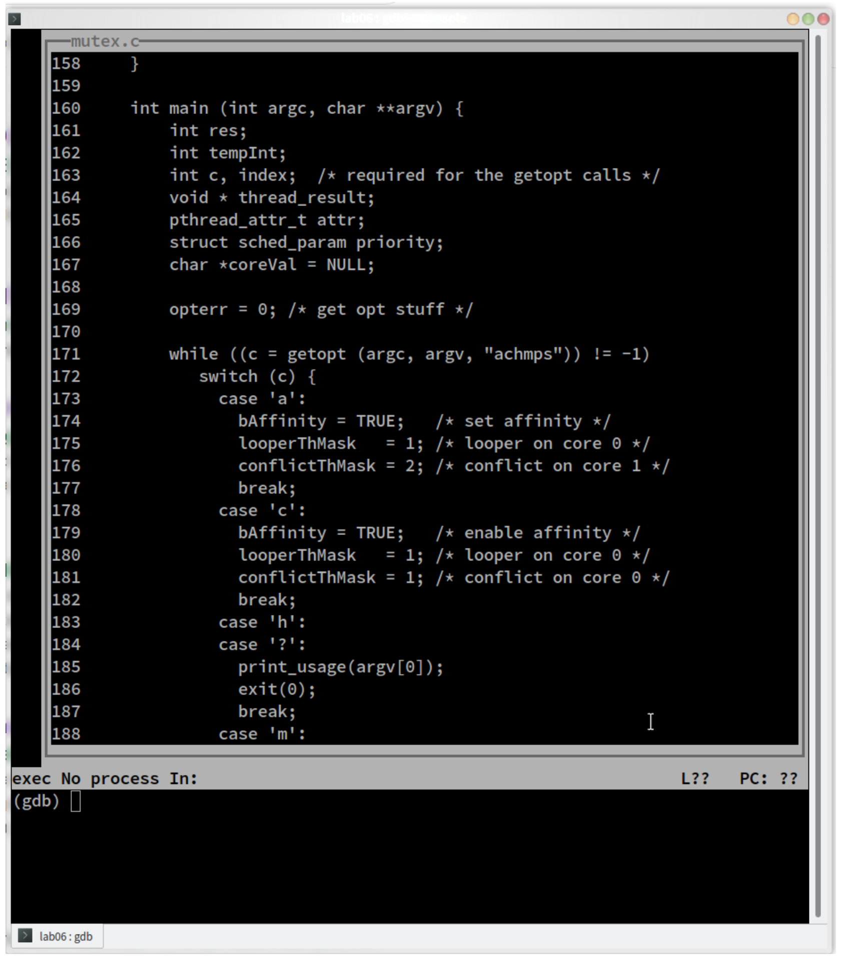
Running GDB CLI

- When gdb is built from source, we will specify the host and target environments
  - Allows for Windows x Linux, x86 host x ARM target, etc.
- When running gdb from the CLI, we can just use the gdb command:

```
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb)
```

Running gdb in TUI Mode

- TUI mode is a text-based user interface that separates out the program text from the gdb command line
    - Uses curses
- More clear cut than using the typical CLI, but maybe not as good as the GUIs for gdb like ddd
- You can start gdb in TUI mode using the --tui command line argument
- You can switch in and out of TUI mode using <CTRL> X A keyboard sequence

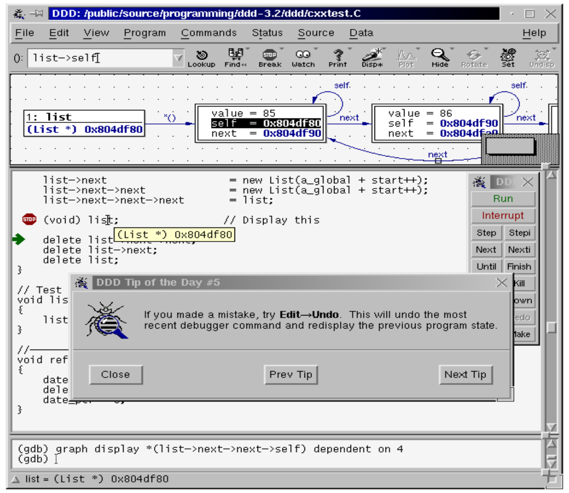


```
mutex.c
}

int main (int argc, char **argv) {
    int res;
    int tempInt;
    int c, index;  /* required for the getopt calls */
    void * thread_result;
    pthread_attr_t attr;
    struct sched_param priority;
    char *coreVal = NULL;

    opterr = 0; /* get opt stuff */

    while ((c = getopt (argc, argv, "achmps")) != -1)
       switch (c) {
         case 'a':
           bAffinity = TRUE;   /* set affinity */
           looperThMask   = 1; /* looper on core 0 */
           conflictThMask = 2; /* conflict on core 1 */
           break;
         case 'c':
           bAffinity = TRUE;   /* enable affinity */
           looperThMask   = 1; /* looper on core 0 */
           conflictThMask = 1; /* conflict on core 0 */
           break;
         case 'h':
         case '?':
           print_usage(argv[0]);
           exit(0);
           break;
         case 'm':
exec No process In:                                          L??   PC: ??
(gdb)
```

KERNEL MASTERS

GDB GUIs

- There are standalone and IDE-based front ends to gdb
- These include:
  - ddd
    - Data Display Debugger
      - Also works with cross debugging
      - http://www.gnu.org/software/ddd/
  - MS Visual Studio Code
    - Yes, it's OSS
    - GDB plug-in that enables graphical debug in IDE
    - https://code.visualstudio.com/download
- IDE support includes Eclipse, Kdevelop, Slickedit®, CodeWarrior®, Arriba® and more


DDD: /public/source/programming/ddd-3.2/ddd/cxxtest.C
File Edit View Program Commands Status Source Data Help
(): list->self
Lookup Find« Break Watch Print Disp* Plot Hide Rotate Set Undisp
self.
next
Run Interrupt Step Stepi Next Nexti Until Finish Kill
DDD Tip of the Day #5
Close Prev Tip Next Tip

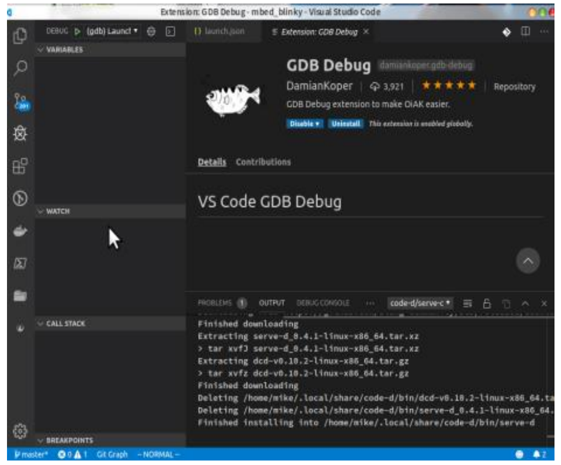
```
1: list
(List *) 0x804df80

value = 85
self  = 0x804df80
next  = 0x804df90

value = 86
self  = 0x804df90
next  = 0x804df
```

```
    list->next                  = new List(a_global + start++);
    list->next->next            = new List(a_global + start++);
    list->next->next->next      = list;

    (void) list;               // Display this
                 (List *) 0x804df80
    delete list
    delete list->next;
    delete list;
}

// Test
void lis
{
    list
}

//
void ref
{
    date
    dele
    date
}
```

If you made a mistake, try **Edit→Undo**. This will undo the most recent debugger command and redisplay the previous program state.

```
(gdb) graph display *(list->next->next->self) dependent on 4
(gdb)
```

list = (List *) 0x804df80



GDB Debug extension to make OiAK easier.

This extension is enabled globally.

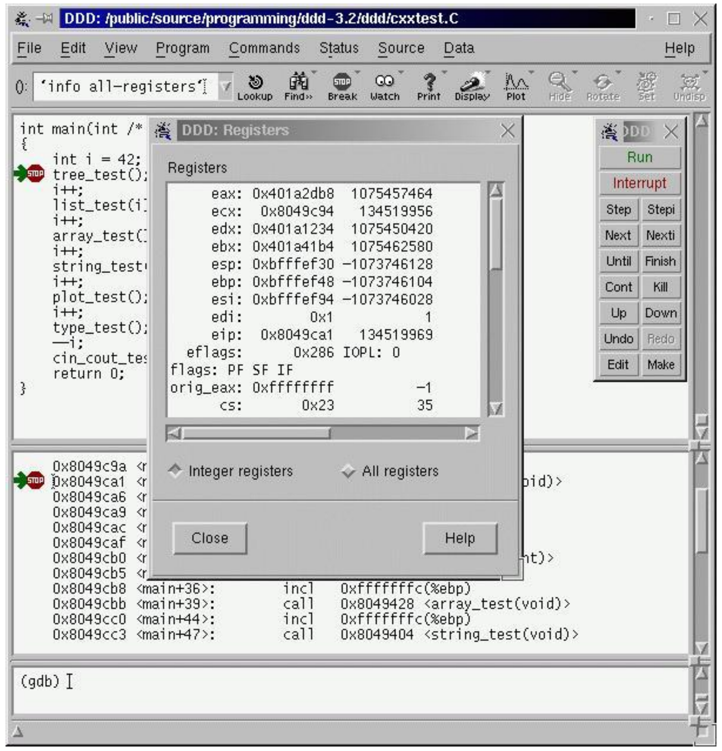
VS Code GDB Debug

```
Finished downloading
Extracting serve-d_0.4.1-linux-x86_64.tar.xz
> tar xvfJ serve-d_0.4.1-linux-x86_64.tar.xz
Extracting dcd-v0.10.2-linux-x86_64.tar.gz
> tar xvfz dcd-v0.10.2-linux-x86_64.tar.gz
Finished downloading
Deleting /home/mike/.local/share/code-d/bin/dcd-v0.10.2-linux-x86_64.ta
Deleting /home/mike/.local/share/code-d/bin/serve-d_0.4.1-linux-x86_64.
Finished installing into /home/mike/.local/share/code-d/bin/serve-d
```

ddd Front End GUI

- ddd is the GNU-supported graphical interface for gdb
- ddd supports:
  - gdb, jdb, Python, Perl, TCL and PHP
- You can automatically load the application into gdb at invocation
- ddd can be started with the **–debugger** option to run a gdb backend other than the default gdb instance

```
$ ddd –debugger arm-linux-gnueabi-gdb
  myapp
```



DDD: /public/source/programming/ddd-3.2/ddd/cxxtest.C
File Edit View Program Commands Status Source Data Help
(): 'info all-registers'
Lookup Find» Break Watch Print Display Plot Hide Rotate Set Undisp
DDD: Registers
Registers
Integer registers
All registers
Close
Help
Run
Interrupt
Step Stepi
Next Nexti
Until Finish
Cont Kill
Up Down
Undo Redo
Edit Make

```
int main(int /*
{
    int i = 42;
    tree_test();
    i++;
    list_test(i
    i++;
    array_test(
    i++;
    string_test
    i++;
    plot_test();
    i++;
    type_test();
    --i;
    cin_cout_tes
    return 0;
}
```

```
      eax: 0x401a2db8  1075457464
      ecx:  0x8049c94   134519956
      edx: 0x401a1234  1075450420
      ebx: 0x401a41b4  1075462580
      esp: 0xbfffef30 -1073746128
      ebp: 0xbfffef48 -1073746104
      esi: 0xbfffef94 -1073746028
      edi:        0x1           1
      eip:  0x8049ca1   134519969
   eflags:      0x286 IOPL: 0
flags: PF SF IF
orig_eax: 0xffffffff          -1
       cs:       0x23          35
```

```
0x8049c9a <r
0x8049ca1 <r                                  oid)>
0x8049ca6 <r
0x8049ca9 <r
0x8049cac <r
0x8049caf <r
0x8049cb0 <r                                  nt)>
0x8049cb5 <r
0x8049cb8 <main+36>:      incl   0xfffffffc(%ebp)
0x8049cbb <main+39>:      call   0x8049428 <array_test(void)>
0x8049cc0 <main+44>:      incl   0xfffffffc(%ebp)
0x8049cc3 <main+47>:      call   0x8049404 <string_test(void)>
```

```
(gdb)
```

KERNEL MASTERS

Command Definition and Macros

- gdb has the ability to define your own commands/scripts
- Use the **`define <name>`** command to define a sequence of gdb commands
  - Enter each one line-by-line and finish with a single line "**`end`**"
    - Useful for creating debugging command scripts that you can save for later use or just to save repeated typing
- Use the **`document <name>`** to write documentation for your defined commands
  - Again, enter each one line-by-line and finish with a single line "**`end`**"
- There is also the ability to define C/C++ preprocessor macros using the **`macro define`** command
  - Visible to all of the inferior's source files

gdb Scripts

- If it exists, gdb will execute all of the commands found in **`.gdbinit`** in the current directory
  - Useful for executing a sequence of gdb commands at gdb initialization:

```
set history save on
set print pretty on
set pagination off
set confirm off
```

- The **`-x`** command line option to gdb also allows for running scripts at gdb load time
- You can also define keyboard macros that launch python scripts or shell off to the OS using **`shell <cmd>`**

Load/Execute Your Code

- If you don't load the program from the command line, you can load additional files using the **`file <filename>`** command
- Once the code is loaded into gdb, you can execute it using the **`run`** command
- You can pass parameters in the same command or you can use the **`set args`** command
  - **`show args`** will allow you to see the arguments

Attach to a Running Program

- gdb has the ability to attach to a running program

```
$ gdb –p <process id>
```

- This will stop the running program at its current execution point
- You can then load the executable’s code and symbol table using the `file` command to load the source if it hasn’t already been loaded.

Examining Code

- Once the program is loaded in gdb, you can list any of the source files using the **`list`** command
  - Options to list a **`LINENUM`**, **`FILE:LINENUM`**, **`FUNCTION`**, **`FILE:FUNCTION`** or **`*ADDRESS`**
- You can specify the number of lines to list as a second parameter
  - Defaults to 10 but can be changed with
    `set listsize <value>`
- You can change the options using the **`set`** command
  - E.g., **`set output-radix 16`** would set the display radix to hexidecimal
  - Use **`show`** to see the available options
  - **`help set <option>`** to get help on the different options

Additional step when compiling program

- Normally, you would compile a program like:

  gcc [flags] <source files> -o <output file>

  For example:

  gcc hello.c -o hello.x

- Now you add a -g option to enable built-in debugging support (which gdb needs):

  gcc [other flags] ***-g*** *<source files> -o <output file>*

  For example:

  gcc ***-g*** *hello.c -o hello_debug*

Starting up gdb

- Just try “gdb” or “gdb a.out.” You’ll get a prompt that looks like this:

```
(gdb)
```

- If you didn’t specify a program to debug, you’ll have to load it in now:

```
(gdb) file a.out
```

Here, a.out is the program you want to load, and “file” is the command to load it.

Starting up gdb

- Just try “gdb” or “gdb a.out.” You’ll get a prompt that looks like this:

  (gdb)

- If you didn’t specify a program to debug, you’ll have to load it in now:

  (gdb) file a.out

Here, a.out is the program you want to load, and “file” is the command to load it.

Example Help Output

- gdb has an interactive shell, much like the one you use as soon as you log into the linux grace machines. It can recall history with the arrow keys, auto-complete words (most of the time) with the TAB key, and has other nice features.

  (gdb) help [command]

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
```

Running the program

- To run the program, just use:

```
(gdb) run
```

- This runs the program.
  - If it has no serious problems (i.e. the normal program didn't get a segmentation fault, etc.), the program should run fine here too.
  - If the program did have issues, then you (should) get some useful information like the line number where it crashed, and parameters to the function that caused the error:

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400524 in sum array region (arr=0x7fffc902a270, r1=2, c1=5,
r2=4, c2=6) at sum-array-region2.c:12
```

Setting breakpoints

- Breakpoints can be used to stop the program run in the middle, at a designated point. The simplest way is the command “break.”
- This sets a breakpoint at a specified file-line pair:

```
(gdb) break file1.c:6
```

- This sets a breakpoint at line 6, of file1.c. Now, if the program ever reaches that location when running, the program will pause and prompt you for another command.
- You can set as many breakpoints as you want, and the program should stop execution if it reaches any of them.

More fun with breakpoints

- You can also tell gdb to break at a particular function. Suppose you have a function my func:

  *int my func(int a, char *b);*

- You can break anytime this function is called:

  (gdb) break my func

gdb commands …

- Once you’ve set a breakpoint, you can try using the run command again. This time, it should stop where you tell it to (unless a fatal error occurs before reaching that point).
- You can proceed onto the next breakpoint by typing “continue” (Typing run again would restart the program from the beginning, which isn’t very useful.)

```
(gdb) continue
```

- You can single-step (execute just the next line of code) by typing “step.” This gives you really fine-grained control over how the program proceeds. You can do this a lot...

```
(gdb) step
```

gdb commands …

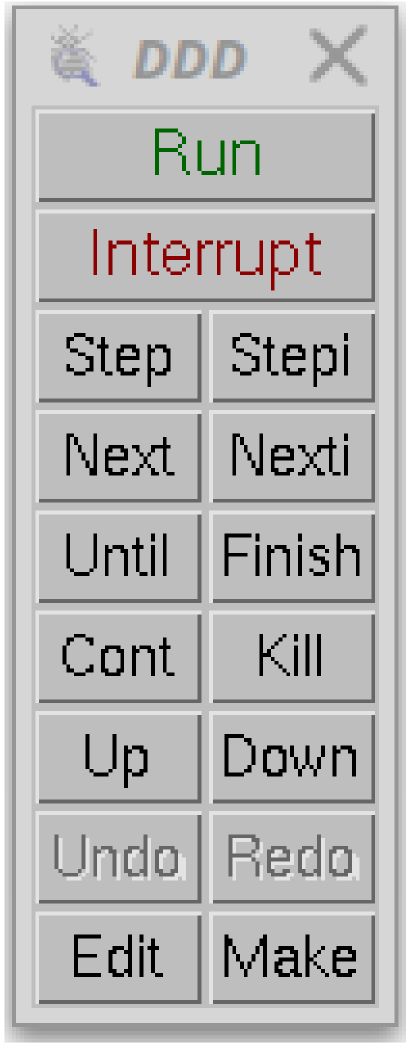
- Similar to “step,” the “next” command single-steps as well, except this one doesn’t execute each line of a sub-routine, it just treats it as one instruction.

```
(gdb) next
```

- Typing “step” or “next” a lot of times can be tedious. If you just press ENTER, gdb will repeat the same command you just gave it. You can do this a bunch of times.

Stepping Through Code

- gdb has a number of ways to step through the code after encountering a breakpoint
  - **step** – step one line and step into functions
  - **next** – step one line and step over functions
  - **finish** – you stepped into a function accidentally and you want to finish this routine
  - **stepi** – step one assembly language instruction and step into function calls
  - **nexti** – step one assembly language instruction but step over function calls



gdb commands …

- So far you've learned how to interrupt program flow at fixed, specified points, and how to continue stepping line-by-line. However, sooner or later you're going to want to see things like the values of variables, etc. This might be useful in debugging.
- The print command prints the value of the variable specified, and print/x prints the value in hexadecimal:

```
(gdb) print my var
(gdb) print/x my var
```

Setting watch points

- Whereas breakpoints interrupt the program at a particular line or function, watch points act on variables. They pause the program whenever a watched variable's value is modified.

For example, the following watch command:

```
(gdb) watch my var
```

- Now, whenever my var's value is modified, the program will interrupt and print out the old and new values

Other useful commands

- **backtrace** - produces a stack trace of the function calls that lead to a seg fault (should remind you of Java exceptions).
- **where** - same as backtrace; you can think of this version as working even when you’re still in the middle of the program
- **finish** - runs until the current function is finished
- **delete** - deletes a specified breakpoint
- **info breakpoints** - shows information about all declared breakpoints.

Working with Signals via gdb

- gdb is built on top of ptrace
  - When an inferior gets a signal, the inferior is suspended and the tracer gets notified
- Show signals

```
(gdb) info signals
```

- Prints a table of how signals and how gdb will handle each one

```
(gdb) info handle
```

- Change the way gdb will handle the signal
  - nostop – do not stop the program but still print that signal occurred
  - stop – stop program when signal occurs (implies print as well)
  - print – print a message when signal occurs
  - noprint – do not mention the occurrence of the signal
  - pass – allow your program to see the signal so it can be handled
  - nopass – do not pass the signal to your program

```
(gdb) handle signal keywords
```

- Delivers a SEGV signal to the current program

```
(gdb) signal SIGSEGV
```

Debugging Threads

- Show active thread ids

  (gdb) info threads

- Select a thread by id

  (gdb) thread n

- Restrict breakpoint to a particular thread

  (gdb) break <break ident> thread <id>

  - Not specifying a thread ID will cause the breakpoint to apply to all threads

- Show backtraces for all threads:

  (gdb) thread apply all bt

- Restrict thread execution to current thread

  (gdb) set scheduler-locking on | off

Strace/ltrace

System call & library trace

strace/ltrace

- Using tools like atsar, top, gkrellm, etc. will give you an idea as to what your system is doing globally. You can now focus on system call and library call tracing
  - **`strace`** and **`ltrace`**
- These require no special compilation flags
  - They can be attached to a running application at any time
  - They support time tagging for crude performance monitoring

Using strace to Watch System Calls

- When debugging what appears to be a kernel-space error, it can be helpful to watch the system calls that are made from user-space
  - See what events lead to the error
- strace displays all system calls made by a program Can display timestamp information per system call as well

Example strace Output

```
/ # strace ls /dev/labdev
execve("/bin/ls", ["ls", "/dev/labdev"], [/* 8 vars */]) = 0
fcntl64(0, F_GETFD)                     = 0
fcntl64(1, F_GETFD)                     = 0
fcntl64(2, F_GETFD)                     = 0
geteuid()                               = 0
getuid()                                = 0
getegid()                               = 0
getgid()                                = 0
brk(0)                                  = 0x1028ad68
brk(0x1028bd68)                         = 0x1028bd68
brk(0x1028c000)                         = 0x1028c000
ioctl(1, TIOCGWINSZ or TIOCGWINSZ, {ws_row=0, ws_col=0, ws_xpixel=0, ws_ypixel=0
ioctl(1, TCGETS or TCGETS, {B9600 opost isig icanon echo ...}) = 0
ioctl(1, TCGETS or TCGETS, {B9600 opost isig icanon echo ...}) = 0
lstat("/dev/labdev", {st_mode=S_IFCHR|0644, st_rdev=makedev(254, 0), ...}) = 0
open("/etc/localtime", O_RDONLY)        = -1 ENOENT (No such file or directory)
lstat("/dev/labdev", {st_mode=S_IFCHR|0644, st_rdev=makedev(254, 0), ...}) = 0
fstat64(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(4, 64), ...}) = 0
ioctl(1, TCGETS or TCGETS, {B9600 opost isig icanon echo ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x300
write(1, "\33[1;35m/dev/labdev\33[0m\n", 23/dev/labdev) = 23
munmap(0x30000000, 4096)                = 0
Exit(0)                                 = ?
```

KERNEL MASTERS

strace – Where is time being spent?

- Use **-c** option while invoking the app
- Example – v4l2-based application

```
$ strace -c ./capture_stream -D /dev/video0 -w 640*480 -p 1|./viewer -w 640*480 -p 1
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 86.40    0.054382         365       149           write
  9.53    0.005999          41       148           select
  3.85    0.002425           8       310         2 ioctl
  0.21    0.000133          10        13           mmap
  0.00    0.000000           0         1           read
  0.00    0.000000           0         3           open
  0.00    0.000000           0         2           close
  0.00    0.000000           0         1           stat
  0.00    0.000000           0         3           fstat
  0.00    0.000000           0         4           mprotect
  0.00    0.000000           0         1           munmap
  0.00    0.000000           0         3           brk
  0.00    0.000000           0         3         3 access
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         1           arch_prctl
------ ----------- ----------- --------- --------- ----------------
100.00    0.062939                   643         5 total
```

- Most of the time is consumed in the write call

KERNEL MASTERS

strace – Where is time being spent?

Use **-c** option while invoking the app

Example : copy file application

Most of the time is consumed in the write call

```
km@KernelMasters:~/KM_GITLAB/debug/user/strace$ strace -c ./a.out
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 53.77    0.073944           8      9286           write
 46.23    0.063563           7      9288           read
  0.00    0.000000           0         2           close
  0.00    0.000000           0         2           fstat
  0.00    0.000000           0         5           mmap
  0.00    0.000000           0         4           mprotect
  0.00    0.000000           0         1           munmap
  0.00    0.000000           0         1           brk
  0.00    0.000000           0         3         3 access
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         1           arch_prctl
  0.00    0.000000           0         4           openat
------ ----------- ----------- --------- --------- ----------------
100.00    0.137507                 18598         3 total
```

KERNEL MASTERS

ltrace

- Just as strace allows us to trace system calls, ltrace allows us to trace library calls
    - You do not have to have the source nor compile the library for debugging to do this
- Can be used to trace glibc interaction
    - But be prepared to get a *lot* of output
- Command-line options exist to limit which libraries you are interested in tracing
    - Other options are similar to strace
- The output looks like strace

Memory Checker

Valgrind

Memory Debugging

- ***Memory leaks:*** Memory leaks are caused when a memory chunk that has been allocated is not freed. Repeated memory leaks can prove fatal to an embedded system because the system may run short of memory.
- ***Overflow:*** Overflow is the condition wherein addresses beyond the end of an allocated area are accessed. Overflow is a very grave security hazard and is used by intruders to hack into a system.
- ***Corruption:*** Memory corruption happens when the memory pointers hold wrong or invalid values. Usage of these pointers may lead to haywire program execution and usually lead to program termination.

What is Valgrind?

- Valgrind is an instrumentation framework for building dynamic analysis tools
  - Built to be user extensible
- It is widely used by Linux developers
- Valgrind tools can automatically detect potential memory management and threading problems
- Has tools for providing profiling data
- Mainly supports C and C++ programs
- Licensed under GPL v2
- Documentation is at http://valgrind.org/doc

Valgrind usage

- The following is the list of the memory checks that can be done using Valgrind.
  - Using uninitialized memory
  - Memory leaks
  - Memory overflows
  - Stack corruption
  - Using memory pointers after the original memory has been freed
  - Nonmatching malloc/free pointer

Coredump/crashdump

GNU Debugger

Generating “Core Dumps”

- When an application terminates abnormally, a core file can be generated
  - core file - (n.) A file created when a program malfunctions and terminates. The core file holds a snapshot of memory, taken at the time the fault occurred. This file can be used to determine the cause of the malfunction
- By default, this feature is disabled to preclude “core droppings” in the file system
  - Use:
    ```
    $ ulimit -c <max core size in 512-byte blocks>
    ```
    to re-enable core file generation

Using the Core File

- Once you have a core file, you can use gdb to try to determine what went wrong
- Load the core file using:

```
$ gdb <application name> -core <corefile>
```

- gdb will load the application and will show you the point of failure
- This will also work with most gdb front-ends
  - eclipse
  - ddd
  - Insight

Communicate with Hardware

mmap() system call

Communicate with Hardware (Memory Mapped I/O)

Read CONTROL STATUS Register:

Control Module Base Address is 0x44E1.0000 ; CONTROL STATUS Register offset address is 0x40

PA of CONTROL STATUS REG = Base Address of control module + offset address of control status register

```
=  0x44E10000
+          40
-------------------------
  0x 44E10040
----------------------------
```

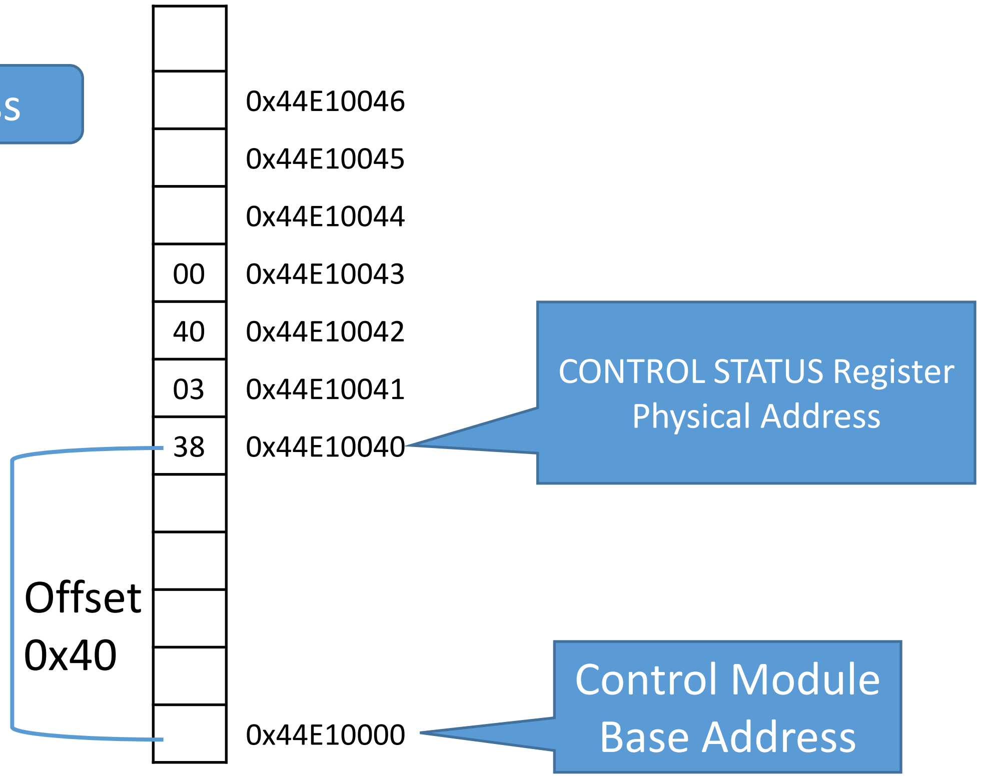
Physical Address = Base Address + Offset Address

Read control status register using md command:

```
=> md <address> [offset]
⇒ md 0x44E10040  (read word of data)
⇒ md.b 0x44E10040 (read byte of data)
=> md 0x44E10040
44e10040: 00400338 00000000 00000000 00000000
```

SYS Boot Switch OFF – 11100

ON – 11000



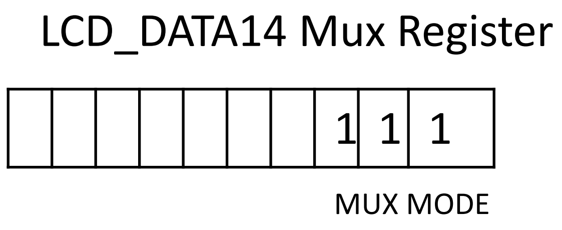
KERNEL MASTERS

MuX/Pad Configuration

What is Mux Configuration:

Most of the pins in microcontroller consists of more than one functionality.

Using multiplexer to select the required functionality. That is called Mux/pad configuration



PA of LCD_DATA14 pad Reg = CM base add + offset

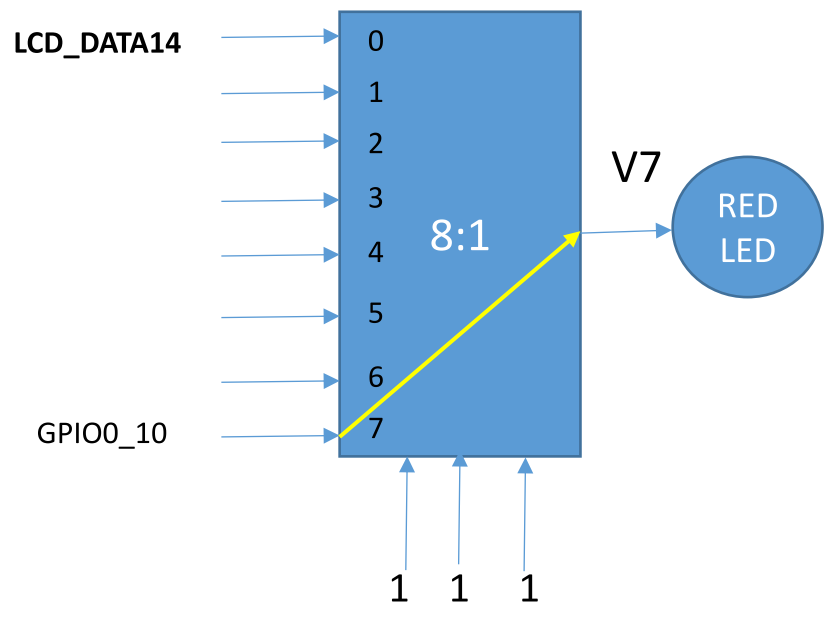
```
= 0x44E10000
+          8D8
--------------------
```

0x44E108D8 (PA of LCD_DATA14 MUX)

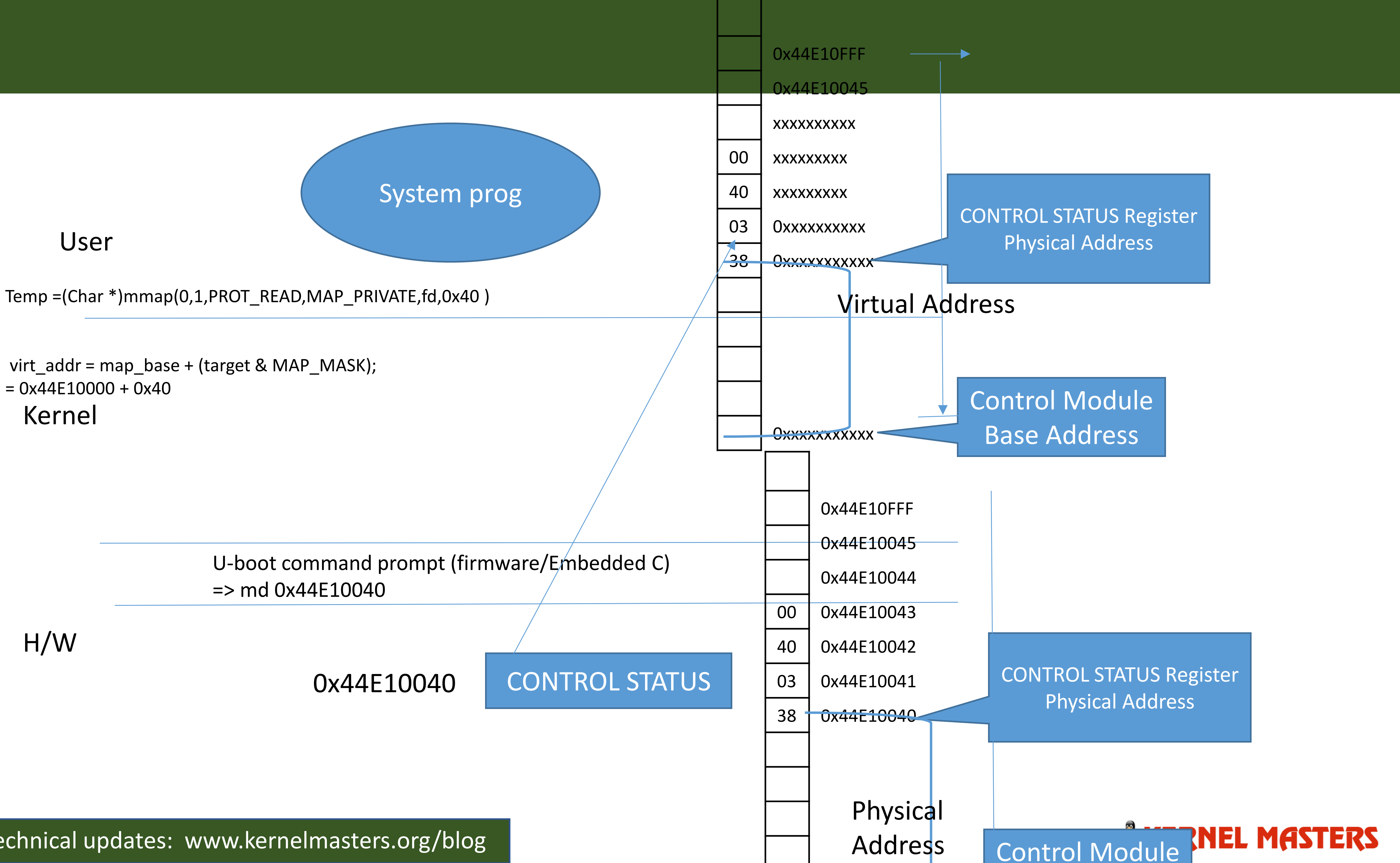
```
-------------------
```

PA of GPIO_OE Register = 0x44E07000 + 0x134 = **0x44E07134**

PA of GPIO_DATAOUT Reg = 0x44E07000 + 0x13C = **0x44E0713C**



KERNEL MASTERS



```
Temp =(Char *)mmap(0,1,PROT_READ,MAP_PRIVATE,fd,0x40 )
```

```
virt_addr = map_base + (target & MAP_MASK);
= 0x44E10000 + 0x40
```

U-boot command prompt (firmware/Embedded C)

```
=> md 0x44E10040
```