



Advanced Lab: IoT Development

David Reyna, Mark Hatle
Wind River Systems

**Yocto Project Developer Day • San Jose •
6 March 2015**

Agenda

- **The Mission**
- **Activity 1: Preparing the Projects**
- **Activity 2: Presentation on Advanced Topics**
- **Activity 3: Preparing the Kernel Module Simulator**
- **Activity 4: Preparing the Application**
- **Activity 5: Prototyping the IoT Edge Device**
- **Activity 6: Extra Credit**
- **Q & A**
- **Appendix**
 - Project Setup at Home
 - Parts List and ordering information
 - Resource links



Greetings, IoT Developer Hero!

Your company is about to release a new IoT product, but the engineer before you has been called away.

Your task is to finish this device. You will need to debug it, remotely update it, connect it to your home server, and demonstrate it to your boss, all in one 3 hour class!

But not to worry, we will show you the techniques you will need to be an IoT hero here and in the future!

The Product

- *A morse code edge device as part of an IoT product to support the severely handicapped*
- *This device must be able to connect with a local target, communicate with peers, and communicate with a remote server*

The Challenge

- **No edge device!**
 - *You have no edge device yet, so you need to simulate the hardware with a custom device driver*
- **No board!**
 - *You have no board yet, so you need to develop and validate the application using QEMU*
- **The Kernel Module has bugs!**
 - *You will need your skills to debug the kernel module*
- **The Application has bugs!**
 - *You will need your skills to debug the application*

Topics Covered

- ***Building on the Yocto Project Beginning Class***
 - *Layers, images, targets, modularity, debugging support*
- ***Advanced Topics***
 - *PRserver, sysfs and GPIO, non-blocking I/O*
- ***Kernel Space***
 - *Custom device drivers, parameter passing, sysfs, kernel timers, debugging techniques*
- ***Application Space***
 - *Custom applications, sysfs access, non-blocking character I/O, application timers, socket servers and clients, debugging techniques*
- ***Board Bring-up***
 - *Connecting an edge device to the board*

Topics Not Covered (for sake of brevity)

- **Security**
 - *Encryption on the line*
 - *Permission control at Sever*
 - *Permission control at Target*
- **Network Configuration**
 - *Firewalls*
 - *Proxies*
 - *Discovery*
- **GUI Tools**
 - *Eclipse*



Activity One

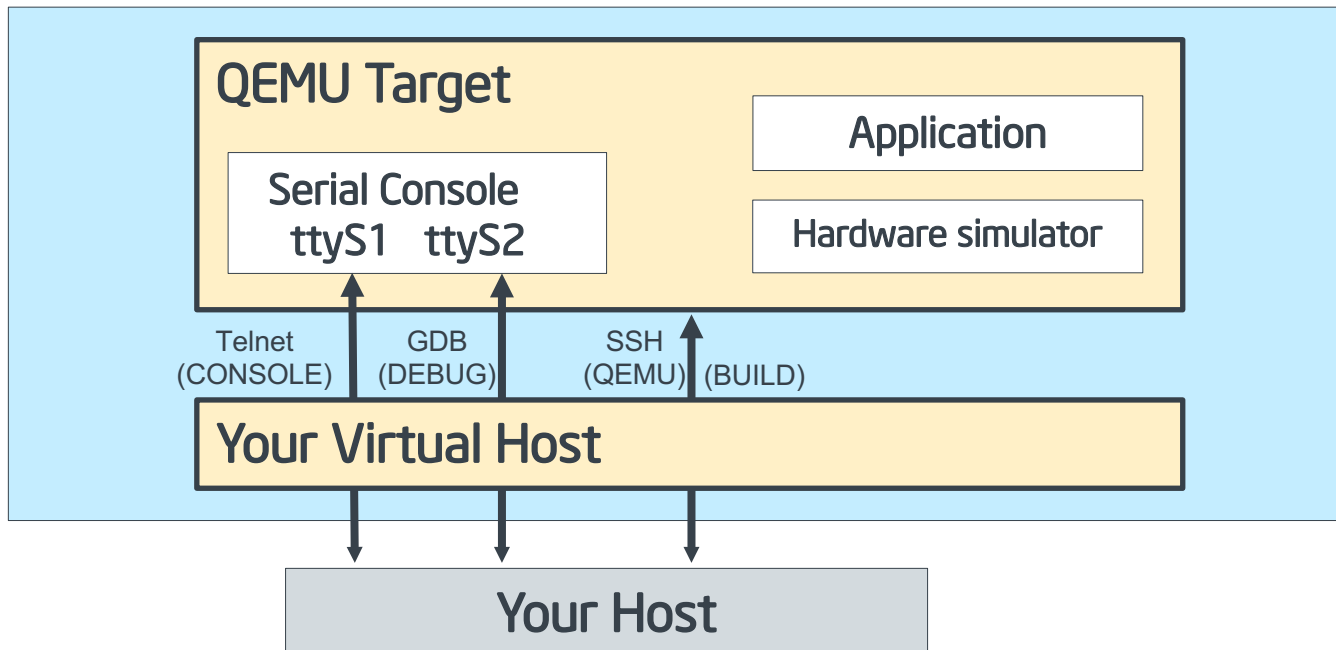
Preparing the Projects

Yocto Project Dev Day Lab Setup

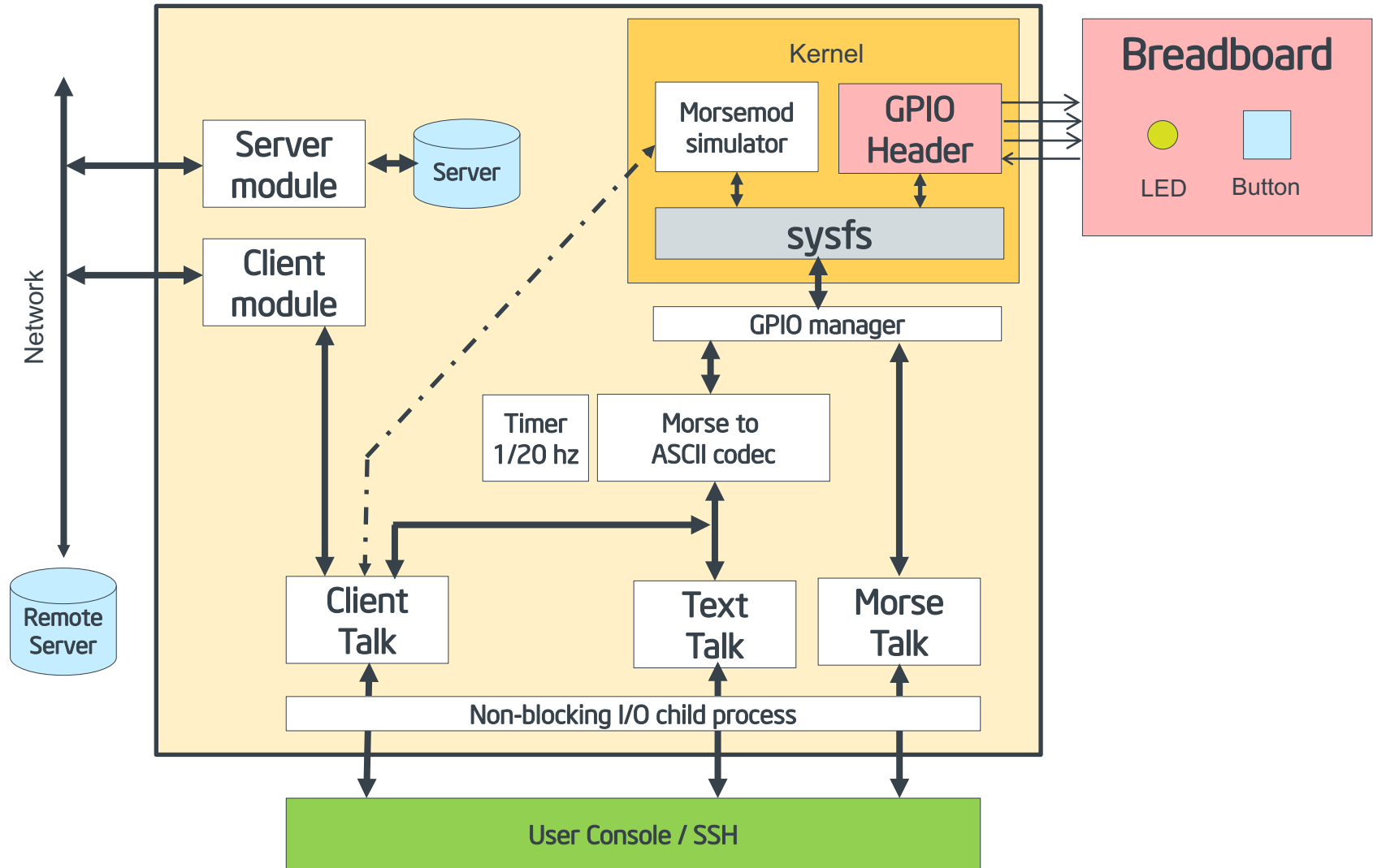
- **You will be given information on how to log into your personal virtual host.**
- **The virtual host's resources can be found here:**
 - Yocto Project: `"/scratch/yocto/sources/poky"`
 - Downloads: `"/scratch/yocto/downloads"`
 - Sstate-cache: `"/scratch/yocto/sstate-cache"`
 - Sources: `"/scratch/yocto/sources"`
- **You will be using SSH to communicate with your virtual server, to reduce the overhead that graphical systems like VNC would require**

Class setup

- Your host use the network to communicate with your virtual host over three SSH connections
- You will communicate with your QEMU target using two serial connections



Morseapp System Block Diagram



Build your QEMU project

- Prepare your QEMU project (for part 1)

```
$ cd /scratch/working
$ . /scratch/yocto/sources/poky/oe-init-build-env build-qemux86
$ vi conf/bblayers.conf

# add to conf/bblayers.conf to BBLAYERS:
  /scratch/yocto/sources/ypdd-adv-layer \

$ vi conf/local.conf
# add to local.conf:
SSTATE_DIR = "/scratch/yocto/sstate-cache"
DL_DIR = "/scratch/yocto/downloads"
IMAGE_INSTALL_append = " gdbserver morseapp morsemod openssh"
EXTRA_IMAGEDEPENDS_append = " gdb-cross-i586 "
INHIBIT_PACKAGE_STRIP_pn-linux-yocto = "1"
INHIBIT_PACKAGE_STRIP_pn-morsemod = "1"

$ bitbake core-image-base
```



Activity Two

Advanced Topics

Advanced Topic: PR Service

- **PR = Package Revision**
- **A package feed manager which is compatible with existing package manager applications like RPM/smart, Debian/apt and opkg.**
- **Attempting to maintain these values in the metadata is error prone, inaccurate and causes problems for people submitting recipes**
- **Example:**
 - `bash-3.2-r0.armv5l.rpm`
 - `bash-3.2-r1.armv5l.rpm`

Advanced Topic: PR Service

- Insures that that versions increase in a linear fashion and there are a number of version components that facilitate this, namely in order of decreasing priority PE, PV and PR (epoch, version and revision).
- Enable with this option:

```
# add to conf/local.conf:  
PRSERV_HOST = "localhost:0"
```

- More information can be found here:
 - https://wiki.yoctoproject.org/wiki/PR_Service

Advanced Topic: GPIO and sysfs

- The "sysfs" virtual file system is a service that provides access from user space to managed kernel resources.
- The GPIO port access is such a service that the Linux kernel provides, and all of the boards in the class support this.
- The GPIO sysfs interface allows simple echo statements to expose ports and set their direction and value.
- To instantiate a GPIO pin in the file system, you simply:

```
$ cd to "/sys/class/GPIO"  
$ echo $PORT > export  
$ cd gpio${PORT}  
$ echo "in" > direction (or) echo "out" > direction  
$ cat value (or) echo $VALUE > value
```
- For the QEMU simulator, we are going to use a custom kernel module to instantiate GPIO ports to simulate the hardware

Advanced Topic: Non-Blocking I/O

- For our application, we need to weave together asynchronous and time critical data from the GPIO hardware, the client-server socket, and the user's keyboard commands
- The C standard I/O libraries are built around line control, which is blocking in nature. We get around this by:
 - Changing STDIN to not wait for EOL
 - Changing STDOUT to not wait for EOL
 - Fork the application into a child that blocks on getchar(), and passes received characters over a pipe to the main thread
 - A side effect is that unexpected exits can leave the console in this non-standard state, needing a manual "reset" command
- The socket interface is normally blocking, but this is easy to reconfigure
- The GPIO sysfs interface is by nature non-blocking, at least for the process that opens and thus reserves the channel

Advanced Topic: Updating without Rebooting

- While performing a full rebuild and re-burning boot media is the cleanest and safest model for updates, it is not a timely method when you need a quick and focused debug and deploy cycle.
- The easiest method to get updated content into the target is to use scp from SSH. Most of the images come with SSH already included, and for images like "core-image-base" you only need to append "openssh".
- *In this class, as a shameless shortcut you will see that we used the morseapp's recipe to create a script that is inserted into the target file system that can be called to perform the pull scp for you. This recipe leverages the bitbake environment information to locate the desired generated content from their sometimes arcane positions in the tree.*
- You can also push the recipe's generated packages (e.g. binary rpm files) to the target for more complex content installing.
- In this class we will try only reboot the QEMU a few times (for the kernel module's sake).

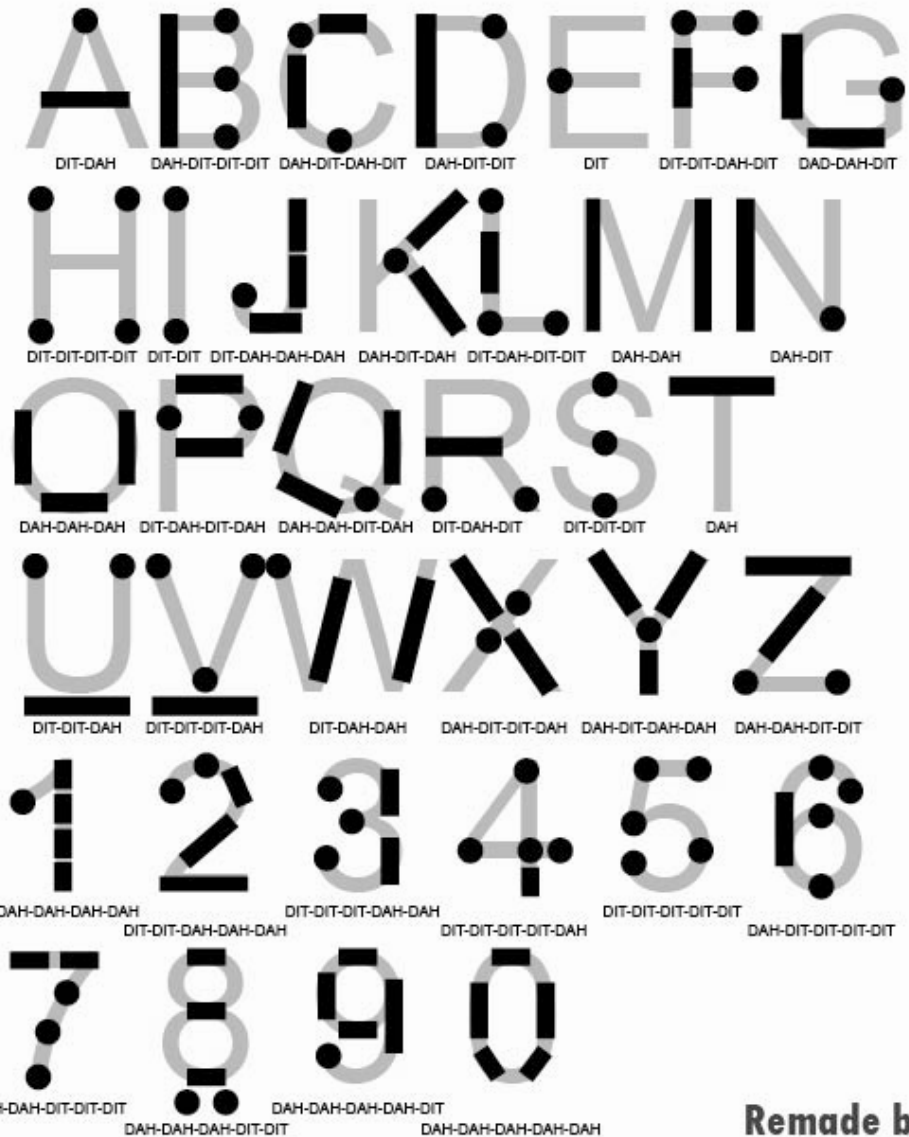
Advanced Topic: Morse Code

- **Timing**

- The "dit" (·) is one time unit long (1/2 second in this implementation)
- The "dah" (–) is three time units long
- The space between dits and dahs is one time unit long
- The space between letters is three time units long
- The space between words is seven time units long
- In this class's implementation, anything shorter than 3/4 second will be a dit, and anything longer will be a dah

- **Facts about Morse Code**

- Professionals can process more than 40 words per minute
- Several morse code readers have been developed for the handicapped, for example using breath control into air tubes
- Morse code together with the telegraph network was the internet of its day, and ran empires



<http://www.learnmorsecode.info/2010/12/a-cool-new-way-to-learn-morse-code/>

**Remade by
Fattredd**



Activity Three

Debugging the Kernel Module

Morsemod features

- **Morsemod Sysfs files**
 - "key" : the virtual key input port
 - "led" : the virtual LED output port
 - "simkey" : the backdoor port
- **Simkey**
 - echo '<' > simkey (turn on the broadcast mode)
 - echo '>' > simkey (turn off the broadcast mode)
 - echo '[' > simkey (turn on the loopback mode)
 - echo ']' > simkey (turn off the loopback mode)
 - cat simkey returns the inner state: "K=0 L=1 B=1 LP=0"
- **The "loopback" will echo the Key value to the LED value**
- **The "broadcast" mode will repeat a pre-recorded morse message**

Debugging the Kernel Module

- **We are now going to use KGDB to test and debug the kernel module**
- **We will show that when encountering an error, it is important to always check dmesg first before going to a debugger**
- **We will be using KGDBoS to debug the kernel over the serial port**
- **We will in fact need to set up QEMU with two serial ports, (a) one for the user console and (b) one for the secondary port for the KGDB access**
- **You should have three (3) windows open: build, target console and remote GDB.**
- **QEMU has a very powerful backend JTAG-like debugging facility; we will skip that in this demonstration to show techniques that would carry over to a physical target**

Interlude – Magic Numbers

- When you assign serial ports in QEMU, they are automatically mapped to `ttyS0`, `ttyS1`, and so forth

```
$ runqemu qemux86 nographic \  
  qemuparams="-serial telnet:localhost:2345,server \  
  -serial tcp:localhost:2346,server,nowait"
```

- That is why when we start the KGDBoC service we will specify `ttyS1`, because your telnet session already took `ttyS0`

```
$ telnet localhost 2345  
# echo ttyS1 > /sys/module/kgdboc/parameters/kgdboc
```

- Note that different arches sometimes have different port names, For example, `qemuarm` uses `ttyAMA0`. You may need to consult the device tree to see what was instantiated.

Debugging with KGDB – error inserting module

- **Build Window: Start the QEMU session with two serial ports**

```
$ runqemu qemu86 nographic \  
qemuparams="-serial telnet:localhost:2345,server \  
-serial tcp:localhost:2346,server,nowait"
```

(Note: to stop QEMU, ctrl-c in the Build Window)

- **Console Window: Start a user console, insert the module, and it fails!**

```
$ telnet localhost 2345  
qemu86 login: root  
root@qemu86:~# modprobe morsemod  
<Error!>
```

- **What happened? First look at dmesg or scrollback!**

```
# dmesg tail  
...  
52 sysfs_warn_dup+0x7a/0x90 ()  
sysfs: cannot create duplicate filename '/kernel/morse-mod/key'  
Modules linked in: morse_mod(O+) uvesafb...
```

Debugging the kernel module – fix #1

- Bug -- we tried to create the 'key' file twice.. whoops!
(trying to use the debugger on this will end up taking twice as long or worse..)
- Let's debug this:

```
$ cd ../ypdd-adv-layer/recipes-ypdd-adv/morsemod/morsemod-2014.10.0
$ vi morsemod.c
```

- Look for `__ATTR` (the structure that holds the sysfs file info)

```
static struct kobj_attribute simkey_attribute =
    __ATTR(key, 0666, simkey_show, simkey_store);
...
static struct kobj_attribute led_attribute =
    __ATTR(led, 0666, b_show, b_store);
static struct kobj_attribute key_attribute =
    __ATTR(key, 0666, b_show, b_store);
```

- Whoops, we defined 'key' twice, top one should be 'simkey'!

```
-     __ATTR(key, 0666, simkey_show, simkey_store);
+     __ATTR(simkey, 0666, simkey_show, simkey_store);
```

Debug the kernel module – try again

- **Build Window: Boot, try it again (see above)**

```
$ bitbake core-image-base
$ runqemu qemu86 nographic qemuparams="-serial \
telnet:localhost:2345,server \
-serial tcp:localhost:2346,server,nowait"
```

- **Console Window: load module and see if it works...**

```
$ telnet localhost 2345
qemu86 login: root
root@qemu86:~# modprobe morsemod
root@qemu86:~# cd /sys/kernel/morsemod
root@qemu86:/sys/kernel/morsemod# cat simkey
K=0 L=0 B=0 LP=0
```

Debug the kernel module – test loopback

- Let's see if it can remember the key and led values

```
root@qemux86:/sys/kernel/morsemod# echo '1' > key
root@qemux86:/sys/kernel/morsemod# cat simkey
K=1 L=0 B=0 LP=0
root@qemux86:/sys/kernel/morsemod# echo '0' > key
root@qemux86:/sys/kernel/morsemod# cat simkey
K=0 L=0 B=0 LP=0
root@qemux86:/sys/kernel/morsemod# echo '1' > led
root@qemux86:/sys/kernel/morsemod# cat simkey
K=0 L=1 B=0 LP=0
root@qemux86:/sys/kernel/morsemod# echo '0' > led
root@qemux86:/sys/kernel/morsemod# cat simkey
K=0 L=0 B=0 LP=0
```

- Let's see if we can enable loopback...

```
root@qemux86:/sys/kernel/morsemod# echo '[' > simkey
root@qemux86:/sys/kernel/morsemod# cat simkey
K=0 L=0 B=0 LP=1
root@qemux86:/sys/kernel/morsemod# echo '1' > key
root@qemux86:/sys/kernel/morsemod# cat simkey
K=1 L=1 B=0 LP=1
root@qemux86:/sys/kernel/morsemod# echo '0' > key
root@qemux86:/sys/kernel/morsemod# cat simkey
K=0 L=0 B=0 LP=1
root@qemux86:/sys/kernel/morsemod# echo ']' > simkey
root@qemux86:/sys/kernel/morsemod# cat simkey
K=0 L=0 B=0 LP=0
```

Debug the kernel module – Bug #2 : broadcast

- Let's try enabling the demo broadcast message...

```
root@qemux86:/sys/kernel/morsemod# echo '<' > simkey
root@qemux86:/sys/kernel/morsemod# cat simkey
K=0 L=0 B=1 LP=0
root@qemux86:/sys/kernel/morsemod# while true ; do cat key ; sleep 1 ; done
0
0
0
0
^C
```

- It doesn't seem to be working. Time to try to debug the module!

Debug the kernel module – Bug #2 : broadcast

- Production image usually lack debug capabilities (stripped symbols, no included debug sources, etc.) In the Yocto Project, debug information is separate from the runtime in most cases. It is packaged into ‘-dbg’ labeled packages.
- By creating a companion debug capable filesystem, you can use remote (cross) debugging to debug a production filesystem.
- **Debug Window: Construct a debug-fs**
 - Extract ‘dbg’ packages (rpm in this example), user space and kernel
 - Add the production filesystem components

```
$ rm -rf debugfs ; mkdir debugfs ; cd debugfs
$ for pkg in ../tmp/deploy/rpm/*/*-dbg* ; do
  ../tmp/sysroots/x86_64-linux/usr/lib/rpm/rpm2cpio $pkg | cpio -i ; done
$ ../tmp/sysroots/x86_64-linux/usr/lib/rpm/rpm2cpio \
  ../tmp/deploy/rpm/*/kernel-dev* | cpio -i
$ tar xvfj ../tmp/deploy/images/qemuarm/core-image-base-qemuarm.tar.bz2
$ cd ..
```

Magic Numbers

- The script that gathers the user space debug RPM files assumes that you have only one arch.

```
$ for pkg in ../tmp/deploy/rpm/*/*-dbg* ; do  
  ../tmp/sysroots/x86_64-linux/usr/lib/rpm/rpm2cpio $pkg | cpio -i ; done
```

- If you can build multiple archs, then you need to explicitly select the ones you need else you may end up with the wrong content.
- For this example where you have both qemux86 and qemuarm built, for qemux86 you will only want the RPMs from "all", "i586", and "qemux86".

```
$ ls tmp/deploy/rpm/  
all  armv5te  i586  qemuarm  qemux86  
$
```

Find the Kernel Source Substitution

- **We need to derive the path to the kernel source**
 - This is a little tricky, in that the path is subject to change since it contains build stamp information. The easiest way of doing this is simply to inspect the vmlinux on the host.
 - Do an objdump, and search for a file directory entry (there should be some near the top). These are what GDB uses to look up the paths for the corresponding file entries.

```
$ objdump --dwarf boot/vmlinux-3.14.26ltsi-yocto-standard | less
```

```
...
```

```
The Directory Table:
```

```
<projdir>/tmp/work/qemux86-poky-linux-gnueabi/linux-yocto/3.14.24+gitAUTOINC+a2...47-r0/linux/arch/arm/include/asm
```

```
<projdir>/tmp/work/qemux86-poky-linux-gnueabi/linux-yocto/3.14.24+gitAUTOINC+a2...47-r0/linux/arch/arm/kernel
```

```
...
```

- The common ancestor in the example above is:

```
<projdir>/tmp/work/qemux86-poky-linux-gnueabi/linux-yocto/3.14.24+gitAUTOINC+a2...47-r0/linux
```

- **Remember this path for the GDB substitution step next**
 - *Note: if you do not issue the substitute-path, the system will attempt to resolve on the local machine and the path specified. This -will- work in many cases, as the user will still have their project available. But where they don't, it will fail.*

KGDBoC - Start

- **Console Window: enable KGDBoC**

```
# echo ttyS1 > /sys/module/kgdboc/parameters/kgdboc
# echo g > /proc/sysrq-trigger
(system will stall and drop into KDB on the 'ttyS1' serial port)
```

(See: <https://www.kernel.org/doc/html/docs/kgdb/EnableKGDB.html>)


- **Debug Window: Connect to the remote kernel (KGDBoS)**
 - Use kernel source substitution path from previous page

```
$ ./tmp/sysroots/x86_64-linux/usr/bin/i586-poky-linux/i586-poky-linux-gdb
(gdb) set sysroot debugfs
(gdb) set substitute-path /usr/src/debug debugfs/usr/src/debug
(gdb) set substitute-path <projdir>/tmp/work/qemux86-poky-linux-gnueabi/linux-yocto/
3.14.24+gitAUTOINC+a2...47-r0/linux debugfs/usr/src/kernel
(gdb) file debugfs/boot/vmlinux-3.14.26ltsi-yocto-standard
Reading symbols from
/scratch/working/build-qemuarm/debugfs/boot/vmlinux-3.14.26ltsi-yocto-standard...done.
(gdb) target remote localhost:2346
Remote debugging using localhost:2346
kgdb_breakpoint ()
    at
/scratch/working/build-qemuarm/tmp/work/qemux86-poky-linux/linux-yocto/...
1043          wmb(); /* Sync point after breakpoint */
(gdb)
```

KGDBoC - Connect

- **Debug Window: Load into GDB the kernel module**

```
(gdb) monitor lsmod
Module                Size  modstruct      Used by
morsemod              4896  0xbf05a778     0 (Live) 0xbf05a000 [ ]
nfsd                  292794 0xbf0395b0    11 (Live) 0xbf000000 [ ]
(gdb) add-symbol-file \
    debugfs/lib/modules/3.14.26ltsi-yocto-standard/extra/morsemod.ko 0xbf05a000
add symbol table from file
"debugfs/lib/modules/3.14.26ltsi-yocto-standard/extra/morsemod.ko" at
    .text_addr = 0xd081a000
(y or n) y
Reading symbols from
/scratch/working/build-qemuarm/debugfs/lib/modules/3.14.26ltsi-yocto-standard/extra/\
morsemod.ko...(no debugging symbols found)...done.
(gdb)
```



- **When broadcast mode is enabled what we're trying to do is iterate over the 'message_str' value using the message_index, so set the break point at the top of the: "if (broadcast == 1)"**

```
(gdb) break morsemod.c:68
(gdb) c
```

KGDBoC - walking

- **Console Window:**

```
root@qemux86:~# cd /sys/kernel/morsemod
root@qemux86:/sys/kernel/morsemod# echo '<' > simkey
```

- **Debug Window:**

Use 'n' to walk through the code flow

NOTE: You likely can't inspect variables (might be a bug in gdb?)

But if you disassemble you'll see something like:

(disassemble /m)

```
73          message_index++;
0xd081a0ea <+90>:  add    $0x1,%ebx
0xd081a0f7 <+103>: mov    %ebx,0xd081a7c0
```

So the value is in the location pointed to by 0xd081a7c0:

(gdb) print *(int*)0xd081a7c0

Eventually you see:

```
74          if (message_index <= strlen(message_str)) {
(gdb)
75          message_index=0;
```

Wait, the message_index keeps getting set back to 0! We should be checking if it -overflows-, not -underflows-!

KGDBoC – fix bug #2

- Change the code to be

```
- if (message_index <= strlen(message_str)) {  
+ if (message_index >= strlen(message_str)) {
```

- Rebuild ...

- (refer to previous steps for build and boot -- remember to regen the debugfs, slides 30-32)

- Restart the module ...

- Console Window:

```
telnet localhost 2345  
qemux86 login: root  
root@qemux86:~# modprobe morsemod  
root@qemux86:~# cd /sys/kernel/morsemod  
root@qemux86:/sys/kernel/morsemod# cat simkey  
K=0 L=0 B=0 LP=0  
root@qemux86:/sys/kernel/morsemod# echo '<' > simkey  
root@qemux86:/sys/kernel/morsemod# cat simkey  
K=1 L=0 B=1 LP=0  
root@qemux86:/sys/kernel/morsemod# while true ; do cat key ; sleep 1 ; done
```

KGDBoC – Bug #3 : stopping broadcast

- Does it work yet...

```
root@qemux86:/sys/kernel/morsemod# while true ; do cat key ; sleep 1 ; done
1
1
0
1
1
1
1
1
.
```

- Great, it works! Now turn it off...

```
.
^C
root@qemux86:/sys/kernel/morsemod# echo '>' > simkey
root@qemux86:/sys/kernel/morsemod# cat simkey
K=0 L=0 B=1 LP=0
root@qemux86:/sys/kernel/morsemod# echo '>' > simkey
root@qemux86:/sys/kernel/morsemod# cat simkey
K=0 L=0 B=1 LP=0
```

- Cannot turn it off, new bug

KGDBoC - Reconnect

- Console Window: Setup kgdb/gdb again ...

```
echo ttyS1 > /sys/module/kgdboc/parameters/kgdboc
echo g > /proc/sysrq-trigger
```

- Debug Window

```
$ ./tmp/sysroots/x86_64-linux/usr/bin/i586-poky-linux/i586-poky-linux-gdb
(gdb) set sysroot debugfs
(gdb) set substitute-path /usr/src/debug debugfs/usr/src/debug
(gdb) set substitute-path <projdir>/tmp/work/qemu86-poky-linux-gnueabi/linux-yocto/
3.14.24+gitAUTOINC+a2...47-r0/linux debugfs/usr/src/kernel
(gdb) file debugfs/boot/vmlinux-3.14.26ltsi-yocto-standard
Reading symbols from
/scratch/working/build-qemuarm/debugfs/boot/vmlinux-3.14.26ltsi-yocto-standard...done.
(gdb) file debugfs/boot/vmlinux-3.14.26ltsi-yocto-standard
Reading symbols from ``
/scratch/working/build-qemuarm/debugfs/boot/vmlinux-3.14.26ltsi-yocto-standard...done.
(gdb) target remote localhost:2346
Remote debugging using localhost:2346
kgdb_breakpoint ()
    at
/scratch/working/build-qemuarm/tmp/work/qemu86-poky-linux/linux-yocto/...
1043          wmb(); /* Sync point after breakpoint */
(gdb)
```

KGDBoC – break in 'simkey_store'

- ... and monitor the module ...

```
(gdb) monitor lsmmod
Module                Size  modstruct    Used by
morsemmod             3973  0xd081a640   0  (Live)  0xd081a000 [ ]
nfsd                  206660 0xd18f1160   11 (Live)  0xd18c6000 [ ]
uvesafb               22894  0xd083baa0   1  (Live)  0xd0837000 [ ]
(gdb) add-symbol-file
debugfs/lib/modules/3.14.261tsi-yocto-standard/extra/morsemmod.ko 0xd081a000
add symbol table from file
"debugfs/lib/modules/3.14.261tsi-yocto-standard/extra/morsemmod.ko" at
      .text_addr = 0xd081a000
(y or n) y
Reading symbols from
/scratch/working/build-qemuarm/debugfs/lib/modules/.../morsemmod.ko...done.
(gdb)
```

- We know the bug is likely in 'simkey_store' because that is the routine that is called when the user attempt to write to the 'simkey' file to control behaviors

```
(gdb) break simkey_store
Breakpoint 1 at 0xd081a000: file
/scratch/working/build-qemuarm/tmp/work/qemux86-poky-linux/.../morsemmod.c,
line 193.
(gdb) c
```

KGDBoC – stepping to bug

- Does it work yet... Console Window:

```
echo '>' > simkey
```

- Debug Window:

```
(gdb) list  
(centered on the first line of the function...)  
  
verify the input was processed correctly  
  
(gdb) print buf[0]  
$1 = 62 '>'
```

- Step through the code to where '>' is processed.
- Observe that due to optimization it skips over that line!
Something must be wrong here!

KGDBoC - disassembly

- If you disassemble the code (disassemble /m), you'll see that both '<' and '>' are setting '1'

```
196         if      ('<' == buf[0]) broadcast=1;
0xd081a008 <+8>:   movzbl  (%ecx),%eax
0xd081a00b <+11>:  mov     %eax,%edx
0xd081a00d <+13>:  and    $0xffffffff,%edx
0xd081a010 <+16>:  cmp    $0x3c,%dl
0xd081a013 <+19>:  je     0xd081a040 <simkey_store+64>
0xd081a043 <+67>:  movl   $0x1,0xd081a7c4

197         else if ('>' == buf[0]) broadcast=1;
198         else if ('[' == buf[0]) loopback=1;
0xd081a015 <+21>:  cmp    $0x5b,%al
0xd081a017 <+23>:  je     0xd081a060 <simkey_store+96>
0xd081a063 <+99>:  movl   $0x1,0xd081a7c8

...pointed to by 0xd081a7c4

(gdb) print *(int *)0xd081a7c4
$2 = 1
```

KGDBoC – manually changing variables

- We can manually clear the value (note this has to be done -after- the code sets it to 0)

```
(gdb)
kobj_attr_store (kobj=<optimized out>, attr=<optimized out>,
    buf=<optimized out>, count=2)
    at
/scratch/working/build-qemuarm/tmp/work/qemux86-poky-linux/.../kobject.c:776
776     }
(gdb) set *(int *)0xd081a7c4 = 0
(gdb) print *(int *)0xd081a7c4
$3 = 0
(gdb) c
```

- Console Window:

```
(root@qemux86:/sys/kernel/morsemod# cat simkey
K=0 L=0 B=0 LP=0
```

- OK we found the bug, go fix the code, retest the code

```
-         else if ('>' == buf[0]) broadcast=1;
+         else if ('>' == buf[0]) broadcast=0;
```

- Observe that it now passes all of the tests!



Activity Four

Preparing the Application

Exercising the morseapp Application

- **We will exercise the morseapp features one step at a time**
- **This will allow us to verify each feature as we build up to the full functionality**
- **This will also provide you with the templates to easily fork and build your own new features**
- **These are the morseapp components:**
 - `morse_app`: set up the non-blocking I/O, configure the interface, provide the basic talk modes
 - `morse_gpio`: manage the GPIO sysfs interface
 - `morse_codec`: manage the morse/text translation
 - `morse_server`: manage the morseapp local/remote server
 - `morse_client`: manage the morseapp client

Run the morseapp Application

- Start morseapp first time

```
# /usr/bin/morseapp
```

- Or re-fetch and start morseapp after a rebuild

```
# /opt/upload_morseapp.sh  
<enter your host password>  
# ./morseapp
```

- **How? The shameless hack in bb file! Bitbake knows where the content is, so you can let it do the work. *For the sake of this class's setup, you will need to insert your special port number into the scp command using "-P portnum"***

```
do_install() {  
    ...  
    # shameless hacking: add helper upload scripts  
    mkdir -p ${D}/opt  
    export ipaddr=`ifconfig eth0 | grep "inet addr" | sed -e "s/.*inet addr:/" -e "s/ .*//"`  
    echo "scp -P portnum $USER@$ipaddr:${D}/usr/bin/morseapp ." > ${D}/opt/upload_morseapp.sh  
    echo "scp -P portnum $USER@$ipaddr:`ls ${TOPDIR}/tmp/work/*/morsemod/0.1-r0/morsemod.ko` ." > ${D}/opt/upload_morsemod.sh  
    echo "insmod morsemod.ko; echo 8 > /sys/kernel/morsemod/simkey" > ${D}/opt/broadcast_morsemod.sh  
    chmod +x ${D}/opt/upload_morseapp.sh  
    chmod +x ${D}/opt/upload_morsemod.sh  
    chmod +x ${D}/opt/broadcast_morsemod.sh  
}
```

Step by Step: the morseapp Application (1)

- Set the input device as the simulator with commands "1,5,m"
- The "loopback" mode will echo changed values from the morsemmod 'device'

```
[ Loopback : device keys are echoed back to the device LED ]
Type '#' to quit
```

```
Device Key: [*]
```

- The "talk morse code" mode will support a morse conversation between the app and the 'device', with backdoor keys for morsemmod

```
[ Talk Morse code ]
```

```
Type the '.' period key to toggle your 'key'
```

```
* a 'dit' is about 1/2 seconds
```

```
* a 'dah' is about 1 1/2 seconds (three 'dits')
```

```
* a letter space is about 1/2 seconds
```

```
* a word space is about 1 1/2 seconds
```

```
Type these keys for the simulated device
```

```
'/' : toggle the device's key
```

```
'<' : broadcast mode on
```

```
'>' : broadcast mode off
```

```
Type '#' to quit
```

```
UserKey: [*] | DeviceKey: [*] || SIM: KEY LED
```

Step by Step: the morseapp Application (2)

- The "talk text" mode will support a text-over-morse conversation between the app and the 'device'

```
[ Talk Text (ASCII) ]
Type your text message: '
 * use the letters a-z, 0-9 (case is ignored)
 * use a space for word separation
Type these keys for the simulated device
 '<' : broadcast mode on
 '>' : broadcast mode off
 '\ ' : force clear the out buffer
Type '#' to quit

In: [????_____] (** ) | Out: [_____] ()
```

- Hit the "<" to remotely start the broadcast mode
- Observe that there are dits but no dahs coming in and that cannot be right - time to bring out the debugger!
- Quit the application with "#" and 'q'

Using GDB - Setup

- Let's step into the program and see how to configure the debugging
- **Console Window:**

```
root@qemux86:~# gdbserver /dev/ttyS1 morseapp
Process morseapp created; pid = 458
Remote debugging using /dev/ttyS1
```

- **Debug Window:**

```
./tmp/sysroots/x86_64-linux/usr/bin/i586-poky-linux/i586-poky-linux-gdb

(gdb) set sysroot debugfs
(gdb) set substitute-path /usr/src/debug debugfs/usr/src/debug
(gdb) file debugfs/usr/bin/morseapp
Reading symbols from
/scratch/working/build-qemuarm/debugfs/usr/bin/morseapp...Reading symbols
from /scratch/working/build-qemuarm/debugfs/usr/bin/.debug/morseapp...done.
done.
(gdb) target remote localhost:2346
Remote debugging using localhost:2346
Reading symbols from debugfs/lib/ld-linux.so.2...Reading symbols from
/scratch/working/build-qemuarm/debugfs/lib/.debug/ld-2.19.so...done.
done.
Loaded symbols for debugfs/lib/ld-linux.so.2
0x4d872d00 in _start () from debugfs/lib/ld-linux.so.2
(gdb)
```


Using GDB - running

- The application was started in the halted position, so let's let it run (*note if you see references or failed to load messages for linux-gate.so.1, that's a virtual library that the kernel sets up for syscalls - there is nothing to load*)

- Debug Window:

```
(gdb) c
```

- In the console follow the steps to enable Talk Text w/ Broadcast (1,5,m,4,>)
 - Use (ctrl-c) to signal the application and debugger with a SIGINT

- Program received signal SIGINT, Interrupt.
0x4d9381fb in __nanosleep_nocancel () at ../sysdeps/unix/syscall-template.S:81
81 T_PSEUDO (SYSCALL_SYMBOL, SYSCALL_NAME, SYSCALL_NARGS)
(gdb) bt
#0 0x4d9381fb in __nanosleep_nocancel ()
 at ../sysdeps/unix/syscall-template.S:81
#1 0x4d968273 in usleep (useconds=useconds@entry=50000)
 at ../sysdeps/unix/sysv/linux/usleep.c:32
#2 0x080494ec in talk_text () at morse_app.c:263
#3 0x08049de9 in parent (p=p@entry=0xbffffd18, child_pid=460)
 at morse_app.c:524
#4 0x08048d68 in main () at morse_app.c:457

Using GDB - stepping with 'finishing'

- Lets take a look at `talk_text` (use 'finish' to complete the current function and return to the caller)

```
(gdb) finish
Run till exit from #0 0x4d9381fb in __nanosleep_nocancel ()
    at ../sysdeps/unix/syscall-template.S:81
usleep (useconds=useconds@entry=50000)
    at ../sysdeps/unix/sysv/linux/usleep.c:33
33     }
(gdb) finish
Run till exit from #0  usleep (useconds=useconds@entry=50000)
    at ../sysdeps/unix/sysv/linux/usleep.c:33
0x080494ec in talk_text () at morse_app.c:263
263     usleep(50000);
Value returned is $1 = 0
(gdb) finish
Run till exit from #0 0x4d9381fb in __nanosleep_nocancel ()
    at ../sysdeps/unix/syscall-template.S:81
usleep (useconds=useconds@entry=50000)
    at ../sysdeps/unix/sysv/linux/usleep.c:33
33     }
(gdb) finish
Run till exit from #0  usleep (useconds=useconds@entry=50000)
    at ../sysdeps/unix/sysv/linux/usleep.c:33
0x080494ec in talk_text () at morse_app.c:263
263     usleep(50000);
Value returned is $1 = 0
```

Using GDB - stepping with 'next'

- **We can step the program with 'next'**
 - Note since the code is optimized stepping is out of order!
 - We can also use 'until' to step until the next line of code.

```
(gdb) n
263                 usleep(50000);
(gdb) until
227                 strcpy(dispatch_outTextStr, outTextStr);
```

Also using 'bt' to get a look at the function back trace:

```
(gdb) bt
#0  talk_text () at morse_app.c:227
#1  0x08049de9 in parent (p=p@entry=0xbffffd18, child_pid=460)
    at morse_app.c:524
#2  0x08048d68 in main () at morse_app.c:457
```

- **Lets set a break point on 'scan_morse_in()'**

```
(gdb) break scan_morse_in
Breakpoint 1 at 0x8049f90: file morse_codec.c, line 145.
```

Using GDB - continuing to the bug

- Continue execution until we hit that function

```
(gdb) c
Continuing.

Breakpoint 1, scan_morse_in () at morse_codec.c:145
145     char scan_morse_in(void) {

(list the sources - repeat 'list' until you see the below code)
(gdb) list
```

- You can see around 159 and 161 where the * and - are put into the buffer... lets break on the *...

```
(gdb) break 159
(gdb) clear scan_morse_in
(gdb) c
Breakpoint 2, scan_morse_in () at morse_codec.c:159
159         strcat(inMorseBuf, "*");
(gdb) list
154             lowcnt++;
155
156             /* parse the 1->0 transition */
157             if ((1 == lastkey) && (0 == key)) {
158                 if (highcnt <= dit_max_cnt)
159                     strcat(inMorseBuf, "*");
160                 else
161                     strcat(inMorseBuf, "-");
162                 highcnt=0;
163             }
```

Using GDB – finding the bug

- Ok, the problem is that we are returning dits for both short and long signals, and the fix is simple

```
157         if ((1 == lastkey) && (0 == key)) {
158             if (highcnt <= dit_max_cnt)
159                 strcat(inMorseBuf, "*");
160             else
- 161                 strcat(inMorseBuf, "*");
+ 161                 strcat(inMorseBuf, "-");
162             highcnt=0;
163         }
```

- You can now fix and rebuild

Using GDB - extra credit

- **BTW, we can inspect the variables...**

```
(gdb) print lastkey
$6 = 1 '\001'
(gdb) print key
$7 = <optimized out>
(gdb) print highcnt
$8 = 30
```

- Note the "optimized out", tricks like what we did in the kernel may be needed here... otherwise we can watch the code flow and determine the likely values!
- Note if you are ever looking at a file and are not sure where the source is, use 'info source'

```
(gdb) info source
Current source file is morse_codec.c
Compilation directory is /usr/src/debug/morseapp/2014.10.0-r0
Located in
/scratch/working/build-qemuarm/debugfs/usr/src/debug/morseapp/2014.10.0-r0/morse_codec.c
Contains 247 lines.
Source language is c.
Compiled with DWARF 2 debugging format.
Does not include preprocessor macro info.
```

Using GDB - extra credit

- You can do the same debugging process as above, but instead of using a serial console, use TCP/IP if your device supports it.
- Simply replace the `/dev/ttyS1` with the address and port of the GDB client.

```
# Console
```

```
root@qemux86:~# gdbserver 192.168.7.1:2345 morseapp  
Process morseapp created; pid = 456  
Listening on port 2345
```

```
# Debug - Server connect such as:
```

```
target remote 192.168.7.2:2345
```

Step by Step: the morseapp Application (3)

- The working "talk text" mode will support a text-over-morse conversation between the app and the 'device'

```
[ Talk Text (ASCII) ]
Type your text message:
 * use the letters a-z, 0-9 (case is ignored)
 * use a space for word separation
Type these keys for the simulated device
 '<' : broadcast mode on
 '>' : broadcast mode off
 '\ ' : force clear the out buffer
Type '#' to quit

In:[?p dev_____] (-* ) | Out: [adbqwer_____] (*- *-- * +4)
```

- The "talk local server" mode will support simple morse code queries to a local server

```
[ Client to Local Server Mode : respond to device queries ]
Server commands are:
 'e' ('*') : echo an 'e' ('*')
 'd' ('-**') : echo the day of the week ('mon' .. 'sun')
 't' ('-') : echo the time ('hhmmss')
 'f' ('**-*') : echo a fortune place
 * use a space for word separation
Type these keys for the simulated device:
 '/' : toggle the device's key
 '\ ' : force clear the out buffer
Server prompt at device is 's' ('---')
Type '#' to quit

In:[_____?___] (- ) | Out: [_____] ()
Local server:d => 'Mon'
In:[_____?___] (---- ) | Out: [_____mon s_____] ( --- -* +5)
```


Step by Step: the morseapp Application (4)

- **Test the server/client functionality**

Setup the Server/Client test connection, using the default address "localhost"

- In one shell, start the test server with the "s" command
- In a second shell, start the test client with the "t" command
- Observe the passing of test strings between these components

- **Run the full server/client functionality**

Setup the Server/Client test connection, using the default address "localhost"

- In one shell, start the morseapp server with the "7" command
- In a second shell, start the client with the "6" command
- Use the "/" key to insert morse code clicks, observe the server replies
- It should work like the "local server" but this time over a local socket

Step by Step: the morseapp Application (5)

- **Connect to the class remote server**
 - Set the class's remote server address in the configuration page ("1" > "7") and start the client again
 - It should work like the "local server" but this time over a remote socket



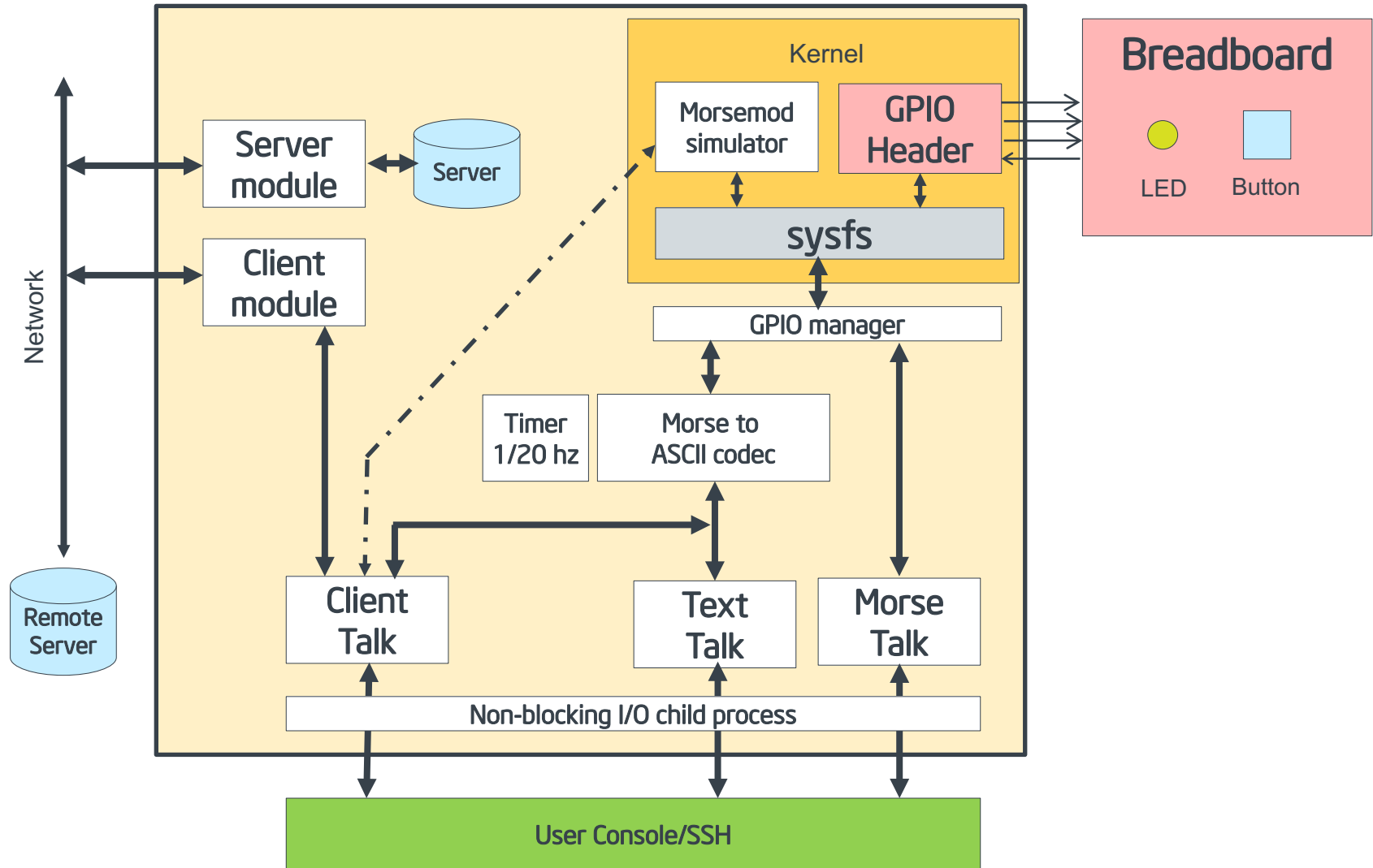
Activity Five

Prototyping the IoT Edge Device

Building a Prototype Edge Device

- **This next section describes how you can build your own prototype device, using a breadboard and a target's GPIO pins**
- **There are three target examples included:**
 - Beaglebone Black
 - Minnowboard MAX
 - Wand Board Quad

Morseapp System Block Diagram



Notes about GPIO (1)

- **Double check that the GPIO pins you wish to use are actually available (e.g. BeagleBone GPIO_20), Here is the test:**

```
$ cd /sys/class/gpio
$ export pin=20
$ echo $pin > export
$ cd gpio${pin}
$ echo out > direction
$ echo 1 > value
$ cat value
```

If the value does not equal 1, then the pin is not fully enabled

- **Make sure that the voltages applied match the accepted range, as most boards use 3.3v for their pins.**
 - The WandBoard has only +5v available on the header yet requires 3.3v for its inputs. You can use a voltage regulator from the 5 volts, but in this case since we are using low current we can simply use a voltage divider resistor network to get the +3.3v reference.

Notes about GPIO (2)

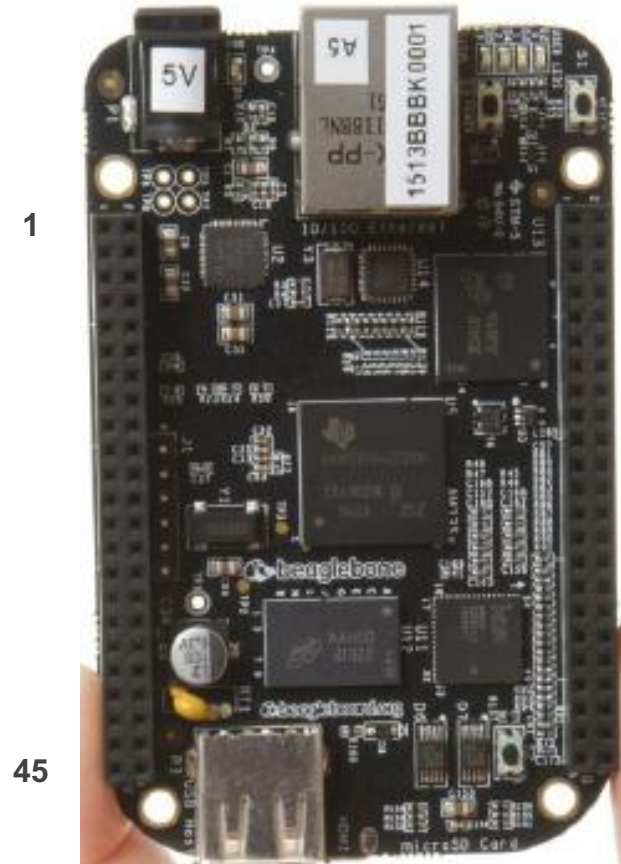
- **Be aware that some GPIO pins can only drive or pull a low amount of current**
 - The Minnowboard Max is an example. In this case use can use a simple FJN3302R Transistor (with bias resistors 10K,10K) to provide the extra pull down current we need for the LED.
- **The extension modules from the board vendors can be your inspiration for your own circuits**
 - Most board vendors in this space publish the detailed schematics of both the boards and the many available I/O devices that they provide, which can provide examples on how exactly to properly design your own interfaces

BeagleBone Black: GPIO Layout

P9

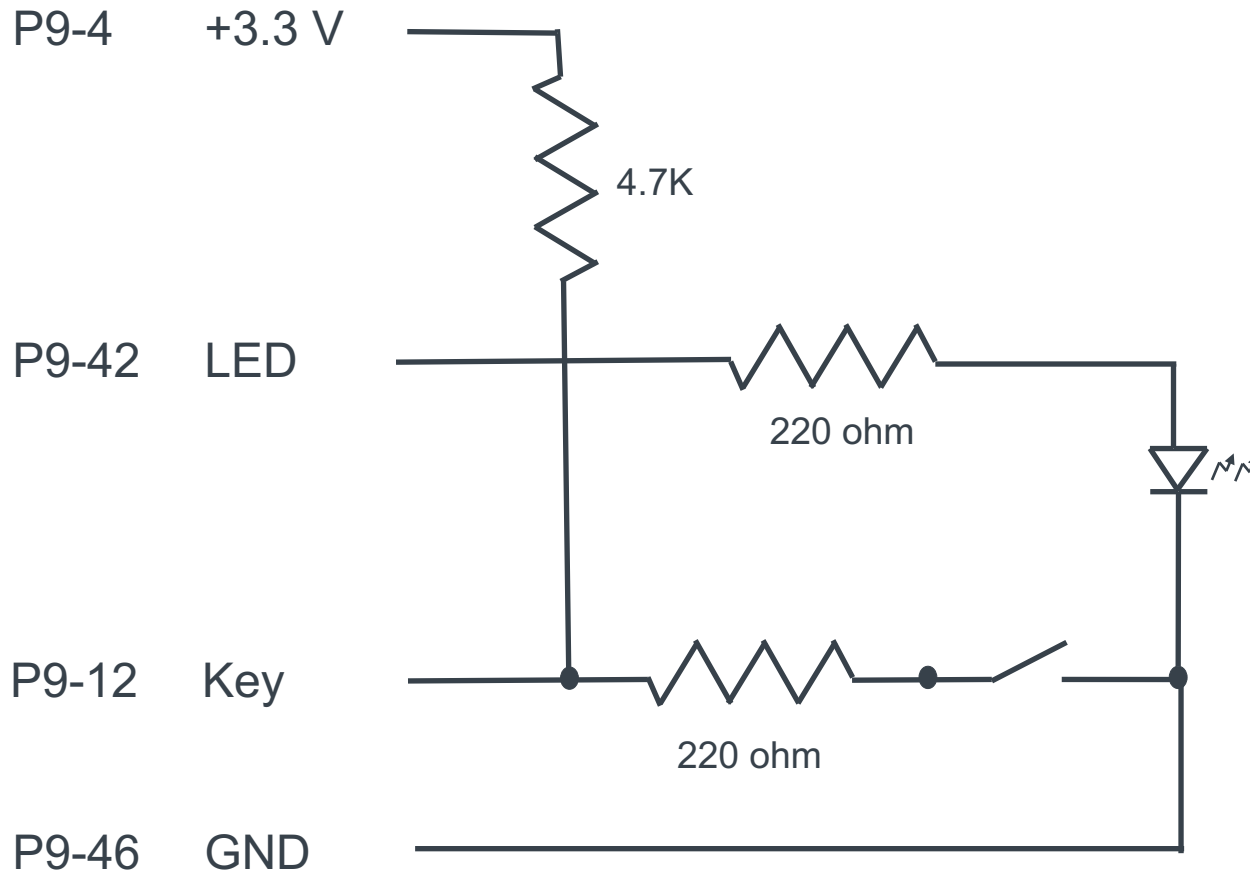
DGND	1	2	DGND	
VDD_3V3	3	4	VDD_3V3	--- +3.3V
VDD_5V	5	6	VDD_5V	
SYS_5V	7	8	SYS_5V	
PWR_BTN	9	10	SYS_RESETN	
GPIO_30	11	12	GPIO_60	--- KEY
GPIO_31	13	14	GPIO_40	
GPIO_48	15	16	GPIO_51	
GPIO_4	17	18	GPIO_5	
I2C2_SCL	19	20	I2C2_SDA	
GPIO_3	21	22	GPIO_2	
GPIO_49	23	24	GPIO_15	
GPIO_117	25	26	GPIO_14	
GPIO_125	27	28	GPIO_123	
GPIO_121	29	30	GPIO_122	
GPIO_120	31	32	VDD_ADC	
AIN4	33	34	GNDA_ADC	
AIN6	35	36	AIN5	
AIN2	37	38	AIN3	
AIN0	39	40	AIN1	
GPIO_20	41	42	GPIO_7	--- LED
DGND	43	44	DGND	
DGND	45	46	DGND	--- GND

Note: GPIO_20 at least is not actually free



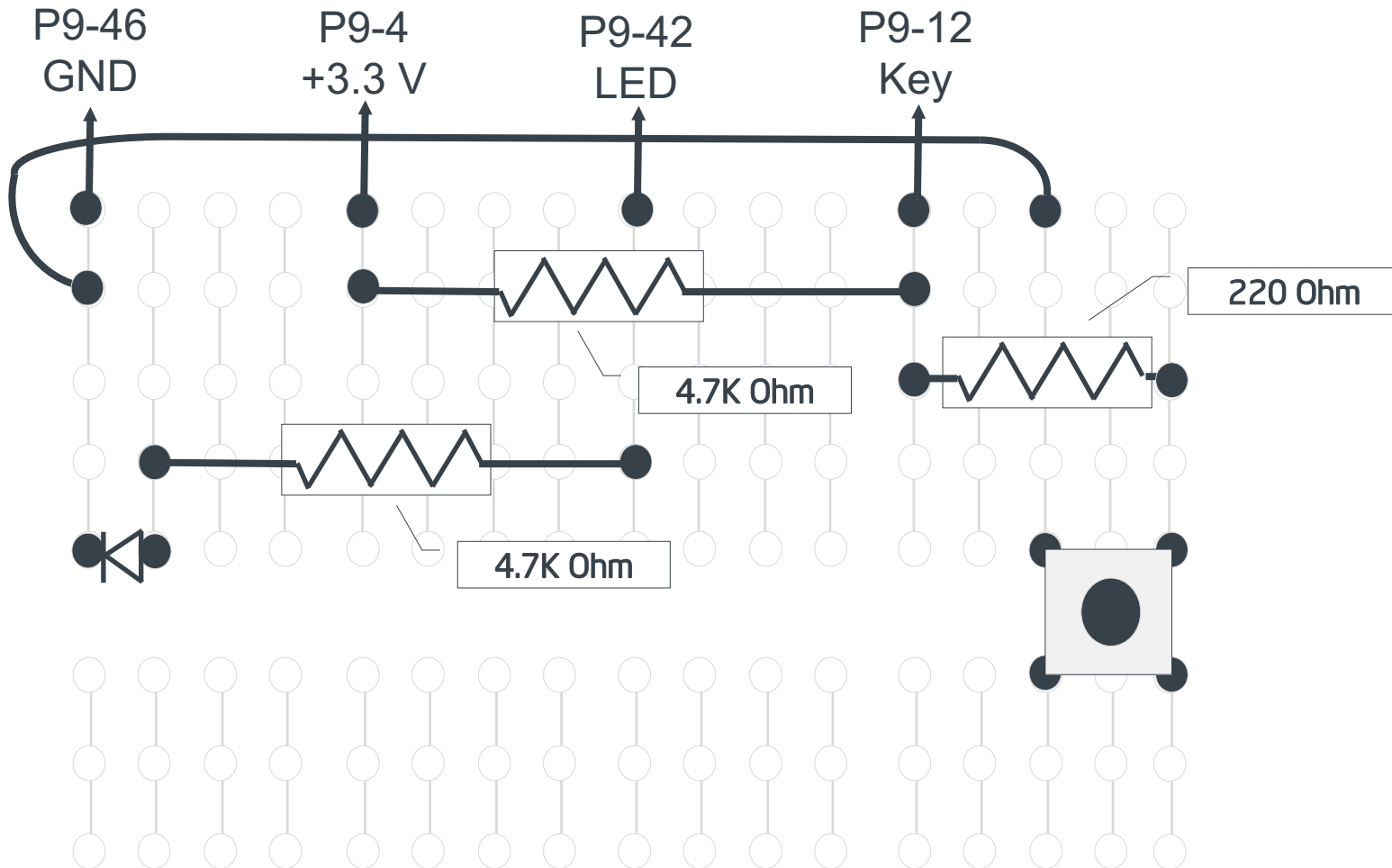
P9

Schematic: Beaglebone Black



220 ohm=red,red,brown 4.7K=yellow,violet,orange

Sample Breadboard Layout: Beaglebone Black



220 ohm=red,red,brown 4.7K=yellow,violet,orange

Minnowboard Max: GPIO Layout

- http://www.elinux.org/Minnowboard:MinnowMax#Low_Speed_Expansion_.28Top.29

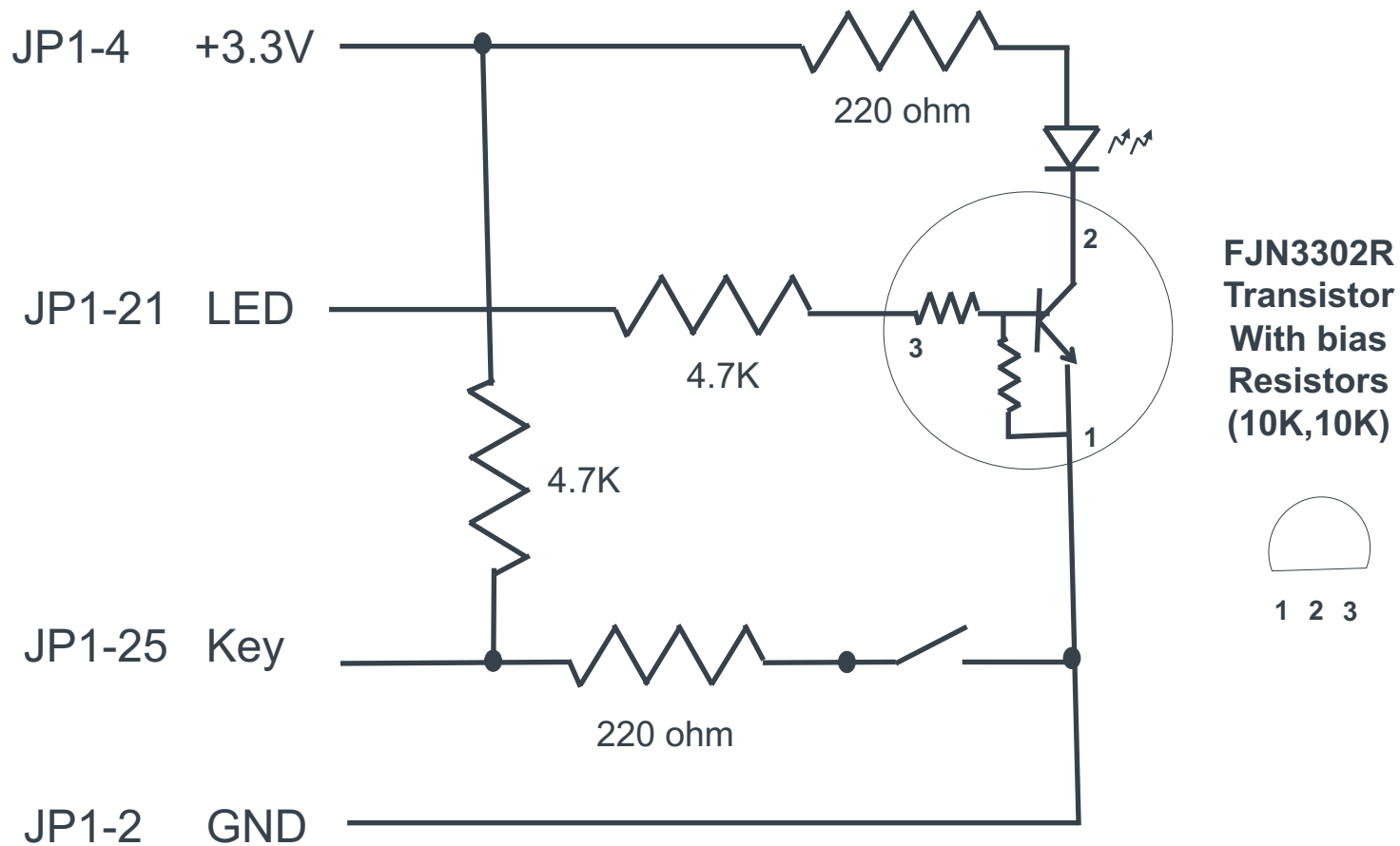


← Power Connector

Description	Name	Pin	Linux	Linux	Pin	Name	Description
			GPIO #	GPIO #			
Ground	Gnd	1			2	Gnd	Ground
+5V Power	VCC	3			4	+3V3	+ 3.3V Power
SPI Chip Select 1	GPIO_SPI_CS#	5	220	225	6	GPIO_UART1_TXD	UART Transmit
Master In / Slave Out	GPIO_SPI_MISO	7	221	224	8	GPIO_UART1_RXD	UART Receive
Master Out / Slave In	GPIO_SPI_MOSI	9	222	227	10	GPIO_UART1_CTS	CTS / GPIO
SPI Clock	GPIO_SPI_CLK	11	223	226	12	GPIO_UART1_RTS	RTS / GPIO
Clock / GPIO	GPIO_I2C_SCL (I2C #5)	13	243	216	14	GPIO_I2S_CLK	Clock / GPIO
Data / GPIO	GPIO_I2C_SDA (I2C #5)	15	242	217	16	GPIO_I2S_FRM	Frame / GPIO
UART Transmit / GPIO	GPIO_UART2_TXD	17	229	219	18	GPIO_I2S_DO	Data Out / GPIO
UART Receive / GPIO	GPIO_UART2_RXD	19	228	218	20	GPIO_I2S_DI	Data In / GPIO
GPIO / Wakeup	LED-- GPIO_S5_0	21	82	248	22	GPIO_PWM0	PWM / GPIO
GPIO / Wakeup	GPIO_S5_1	23	83	249	24	GPIO_PWM1	PWM / GPIO
GPIO / Wakeup	KEY-- GPIO_S5_4	25	84	208	26	GPIO_IBL_8254	Timer / GPIO

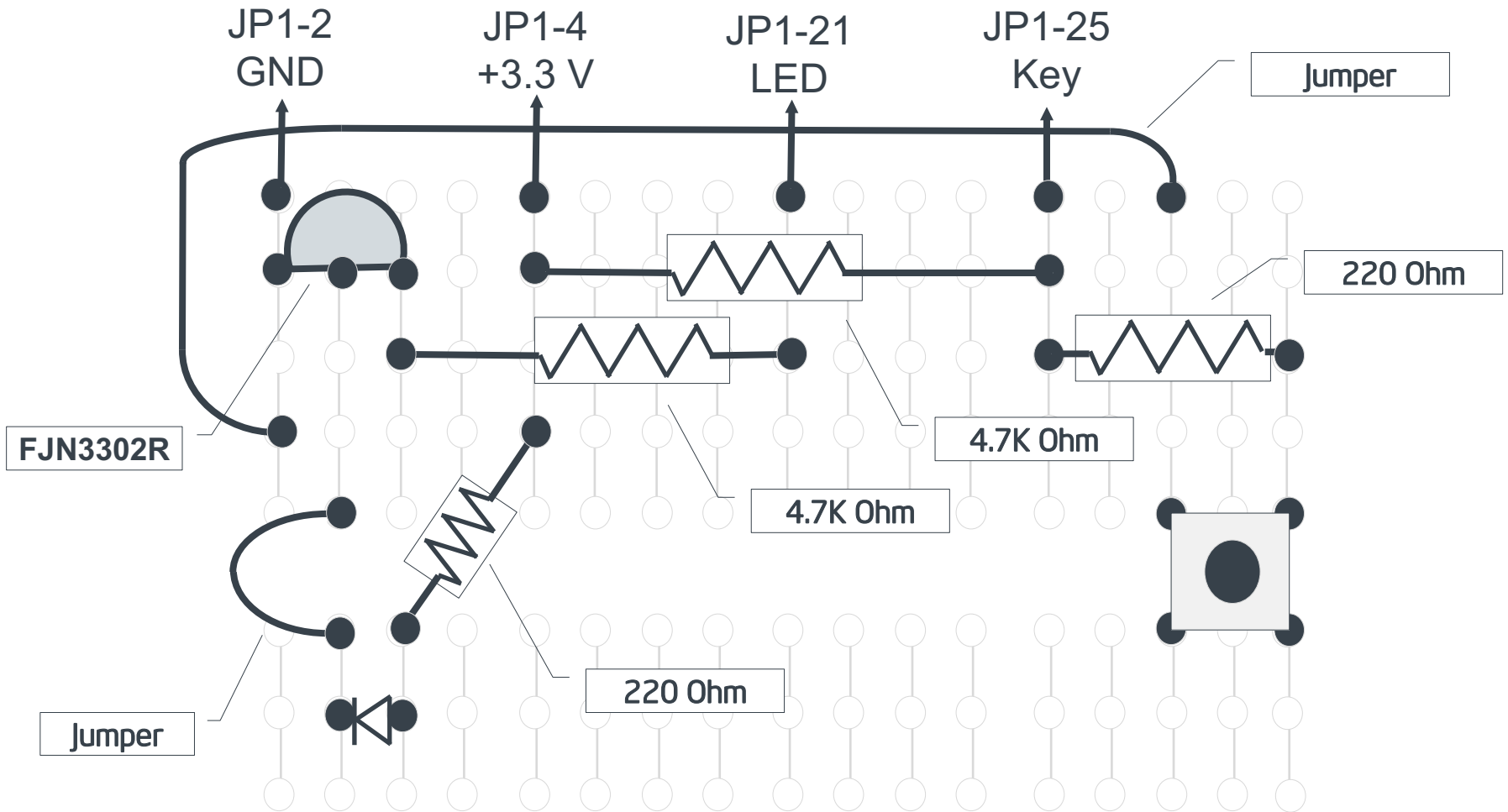
Note: The pins 21, 23, and 25 are the free GPIO pins.

Schematic: Minnowboard Max



220 ohm=red,red,brown 4.7K=yellow,violet,orange

Sample Breadboard Layout: Minnowboard Max

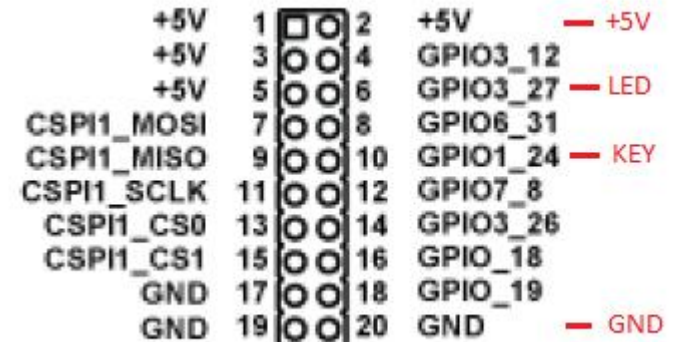


220 ohm=red,red,brown 4.7K=yellow,violet,orange

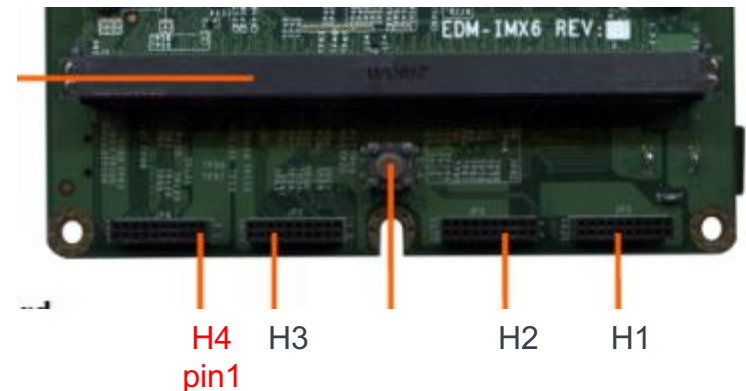
WandBoard : GPIO Layout

- http://wiki.wandboard.org/index.php/External_gpios

iMX6 GPIO (bank, number)	sysfs name	JP-4 pin	MorseApp
+5 V		2	+5 V
GPIO(3, 11)	gpio75	4	
GPIO(3, 27)	gpio91	6	LED
GPIO(3, 26)	gpio90	14	
GPIO(6, 31)	gpio191	8	
GPIO(3, 8)	gpio72	16	
GPIO(1, 24)	gpio24	10	KEY
GPIO(4, 5)	gpio101	18	
GPIO(7, 8)	gpio200	12	
GND		20	GND

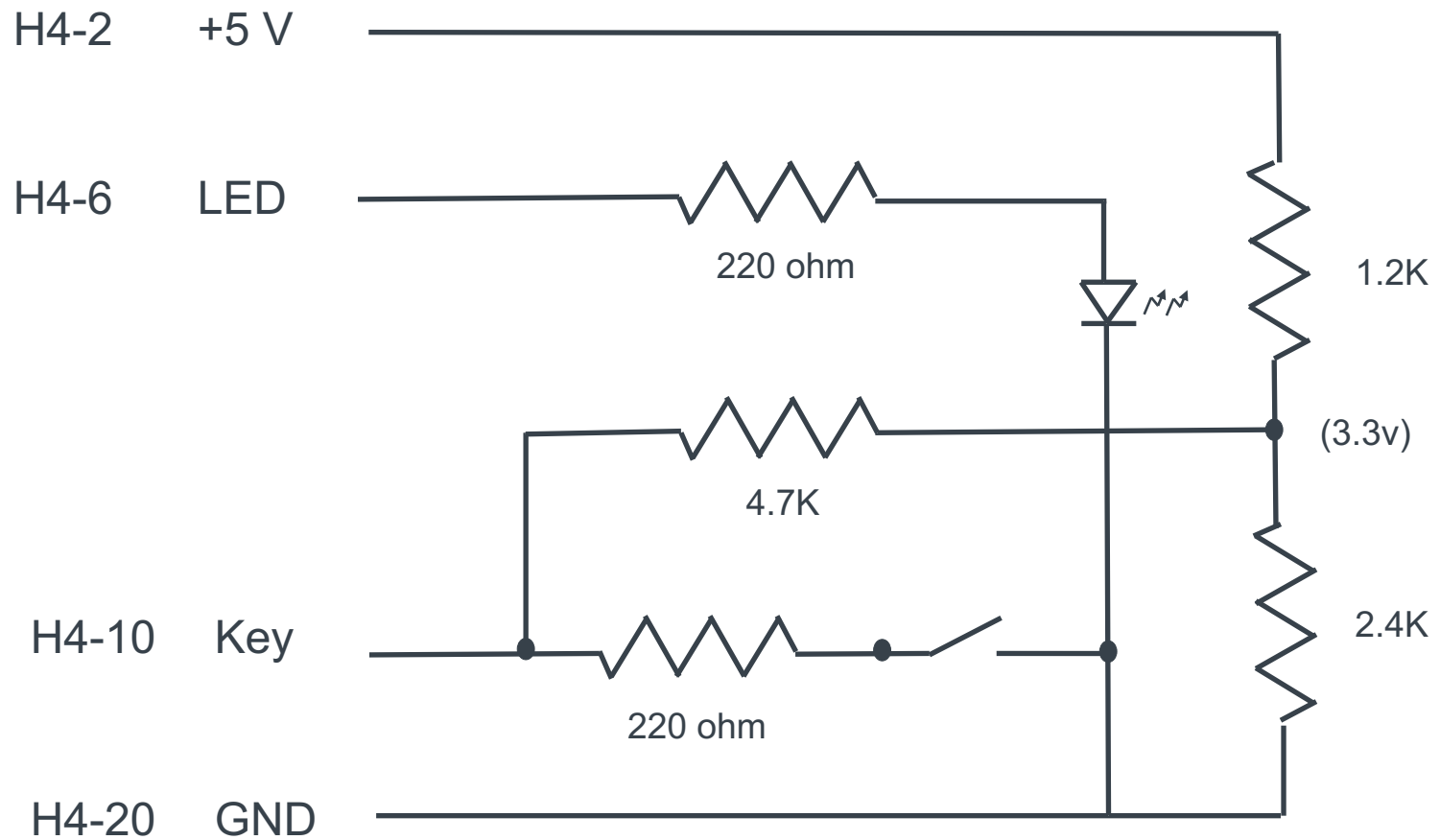


Header 4



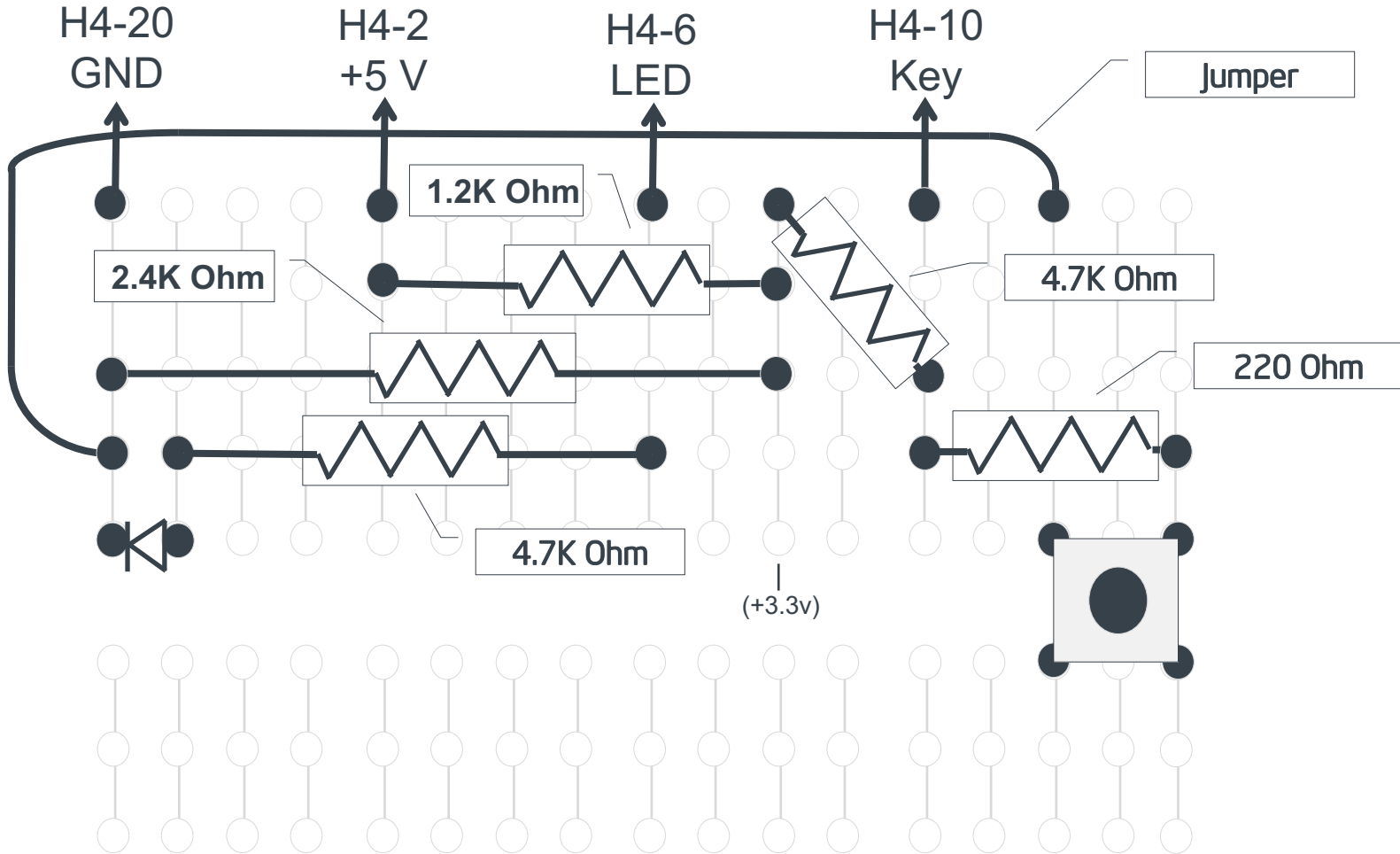
Note: use 20 gauge solid core wire for temporary jumping, but soldering to a male-male 0.05 header that you plug in is the recommend connection method for breadboarding.

Schematic: WandBoard Quad



220 ohm=red,red,brown 4.7K=yellow,violet,orange 1.2K=brown,red,orange 2.4K=red,yellow,orange

Sample Breadboard Layout: Wandboard Quad



220 ohm=red,red,brown 4.7K=yellow,violet,orange 1.2K=brown,red,orange 2.4K=red,yellow,orange



Activity Eight

- **Extra Credit**

Extra Credit Exercises - 1

- **Add punctuation**
 - There are morse codes for many punctuation marks. Add them to the application's codec (see for example "wikipedia").
- **Changing the 'dit' time reference**
 - Add code to the kernel module to allow user configuration of the broadcast mode timing (e.g. make it go faster)?
 - Add code to the application to also allow configuration of the morse code timing translation, and also pass that setting to the kernel module?
- **Pushing updates instead of pulling**
 - Modify the morseapp recipe to generate a script that pushes instead of pulls updated content to the target. Note that while the default root password makes it easier, each restart of QEMU results in a different SSH signature.

Extra Credit Exercises - 2

- **Setting the timers to account for delays**

- The timers in the morseapp scan the keypad input port every 1/20 second, but what the code is really doing is adding a 1/20 delay after all the loop's code (like printf and socket I/O)

How would you re-write the code such that it accounts for the time loss, and anticipates the proper delay to stay on track? How would you collect and present those statistics for monitoring?

- **Accessing the GPIO on the kernel side**

- The sysfs interface provides a basic interface to the GPIO ports. How could moving that access to the kernel side with your own kernel module help your product?
- What functionality would you move from the application into the kernel module (e.g. dit/dah timing)?
- Would a char block device be better than sysfs?

Extra Credit Exercises - 3


- **Finish Peer Mode**

- You can easily use this infrastructure to do peer-to-peer communication without a central server
- When the Client starts in peer mode, it should start its own server listening, and poll to see if the peer's server has yet started
- The client's main loop would be to send any pending traffic, see if anything came in for its server, and of course check the user input.
- When the client exits, it then must also stop its server.
- If the client times out sending to the peer server, then it should close.
- See if you can find the implementation, complete the missing steps, and start conversing across the room in morse code.



yocto
PROJECT™

Questions and Answers



**Thank you for your
participation!**

yocto ·
PROJECT

 THE
LINUX
FOUNDATION

Appendix: Beaglebone Black Setup at Home

- Here is how you can build your project for Beaglebone Black

```
$ export INSTALL_DIR=`pwd`
$ git clone -b daisy git://git.yoctoproject.org/poky
$ source poky/oe-init-build-env `pwd`/beagle
$ echo 'BBLAYERS += "/path/to/ypdd-adv-layer"' >> conf/bblayers.conf
$ echo 'MACHINE = "beaglebone"' >> conf/local.conf
$ echo 'IMAGE_INSTALL_append = " gdbserver morseapp morsemmod openssl"' >> conf/local.conf
$ echo 'EXTRA_IMAGEDEPENDS_append = " gdb-cross"' >> conf/local.conf
$ bitbake core-image-base
```

Appendix: Minnowboard Max Setup at Home

- Here is how you can build your project for Minnowboard Max

```
$ export INSTALL_DIR=`pwd`
$ git clone -b daisy git://git.yoctoproject.org/poky
$ git clone -b daisy git://git.yoctoproject.org/meta-intel
$ source poky/oe-init-build-env `pwd`/minnow
$ echo 'BBLAYERS += "$INSTALL_DIR/source/meta-intel"' >> conf/bblayers.conf
$ echo 'BBLAYERS += "/path/to/ypdd-adv-layer"' >> conf/bblayers.conf
$ echo 'MACHINE = "intel-corei7-64"' >> conf/local.conf
$ echo 'IMAGE_INSTALL_append = " gdbserver morseapp morsemod openssl"' >> conf/local.conf
$ echo 'EXTRA_IMAGEDEPENDS_append = " gdb-cross"' >> conf/local.conf
$ bitbake core-image-base
```


Appendix: WandBoard Setup at Home

- Here is how you can build your project for WandBoard

```
$ mkdir ~/bin
$ curl http://commodatastorage.googleapis.com/git-repo-downloads/repo > ~/bin/repo
$ chmod a+x ~/bin/repo

Create the BSP directory download all of the metadata for the BSP layers

$ PATH=${PATH}:~/bin
$ mkdir fsl-community-bsp
$ cd fsl-community-bsp
$ repo init -u https://github.com/Freescale/fsl-community-bsp-platform -b daisy
$ repo sync
$ ./setup-environment build
$ echo 'BBLAYERS += "/path/to/ypdd-adv-layer"' >> conf/bblayers.conf
$ echo 'MACHINE = "wandboard-quad"' >> conf/local.conf
$ echo 'IMAGE_INSTALL_append = " gdbserver morseapp morsemod openssl"' >> conf/local.conf
$ echo 'EXTRA_IMAGEDEPENDS_append = " gdb-cross"' >> conf/local.conf
$ echo 'ACCEPT_FSL_EULA = "1"' >> conf/local.conf
$
$ bitbake core-image-base
```

Appendix: Parts List Order Information

- Here is sample ordering information for the breadboard parts. These are all very common and can be obtained from most any source.

Part	Sample Ordering Link
Basic Breadboard	https://www.jameco.com/webapp/wcs/stores/servlet/Product_10001_10001_2155452_-1
220 ohm resistor	https://www.jameco.com/webapp/wcs/stores/servlet/Product_10001_10001_690700_-1
1.2K resistor	https://www.jameco.com/webapp/wcs/stores/servlet/Product_10001_10001_690881_-1
2.4K resistor	https://www.jameco.com/webapp/wcs/stores/servlet/Product_10001_10001_690953_-1
4.7 K resistor	https://www.jameco.com/webapp/wcs/stores/servlet/Product_10001_10001_691024_-1
LED, green	https://www.jameco.com/webapp/wcs/stores/servlet/Product_10001_10001_334086_-1
FJN3302R, TRANSISTOR NPN with 10K bias resistors, TO-92	http://www.digikey.com/product-detail/en/FJN3302RTA/FJN3302RTACT-ND/4213840
Switch N.O. suitable for breadboard	https://www.jameco.com/webapp/wcs/stores/servlet/Product_10001_10001_149948_-1
Jumpers Pin-to-Pin (BeagleBone)	https://www.jameco.com/webapp/wcs/stores/servlet/Product_10001_10001_126342_-1
Jumpers Socket-to-Pin (MinnowMax)	https://www.jameco.com/webapp/wcs/stores/servlet/Product_10001_10001_2214563_-1

Appendix: Useful Links

- **KObjects / sysfs**
 - <http://lwn.net/Articles/266722/>
 - http://www.crashcourse.ca/wiki/index.php/Kernel_sysfs
 - <http://www.ug.it.usyd.edu.au/~vnik5287/sysctl.pdf>
 - <http://www.linux.org/threads/sysfs-and-configfs.4956/>
 - <http://www.signal11.us/oss/udev/>
 - <http://www.cs.fsu.edu/~baker/devices/lxr/http/source/linux/samples/kobject/kobject-example.c>
 - <http://www.cs.fsu.edu/~baker/devices/lxr/http/source/linux/samples/kobject/kset-example.c#L214>
- **Wandboard**
 - <http://www.wandboard.org/index.php/51-20131028-wandboard-gpio-hands-on>
- **Beaglebone SD formatting**
 - <http://eewiki.net/display/linuxonarm/BeagleBone+Black#BeagleBoneBlack-SetupmicroSD/SDcard>
 - <https://www.yoctoproject.org/downloads/bsps/daisy16/beaglebone>
- **Parent/child for non-blocking getch**
 - http://www.albany.edu/~csi402/pdfs/handout_15.2.pdf
- **Sockets**
 - <http://www.tenouk.com/Module40c.html>
 - <http://stackoverflow.com/questions/10619952/how-to-completely-destroy-a-socket-connection-in-c>