Digi Home Health Hub (HHH)

Michael Erickson

2013-01-18 Fri

Contents

1 Introduction

This document contains a list of instructions and developer notes related to recreating the Digi Home Health Hub (HHH) Panic Button demonstration.

2 Development System

The demonstration was developed using a 64-bit Debian Linux version 6.0.5 system. Specifically, the developer worked on a MAC and used VMware Fusion to run Debian inside a virtual box. This section details the setup and configuration of the Linux development system. It is targeted at using VMware, but the list of necessary packages and configuration steps will apply to a dedicated machine as well. I used a 64-bit Debian 6.0.5 image.

2.1 Build a Debian Virtual Box

- Download Debian AMD64 Net installation disk[1]
- Start VMWare Fusion and install Debian from the ISO
- Adjust the Processor/Memory to 1024 MiB of memory
- Adjust disk size to 20 GiB
- Under *Network*, tell this virtual machine to make its own connection to the network (so TFTP will work)
- If Debian can't find the network during the install, switch to hardwired Ethernet, reboot the MAC, then try again
- Have VMWare *Connect Directly To* the MAC's network using *autodetect*. If you don't do this, the kit won't be able to find our Linux image.

[1] I have found that you *must* use the net-install disk.

2.2 Configure the Development System

These instructions will install the minimum number of packages necessary to make a working development system.

- Select *base-system* and *ssh-server* for the Debian install
- Install
- Login to the system as the *root-user*
- Edit `/etc/apt/apt.conf` to not install recommends and suggests

  ```
  echo 'APT::Install-Recommends "0";' >> /etc/apt/apt.conf
  echo 'APT::Install-Suggests "0";'   >> /etc/apt/apt.conf
  ```

- Update the system: `apt-get update && apt-get upgrade`
- Install vim: `apt-get install vim`
- *(optional)* Install emacs-nox: `apt-get install emacs23-nox`
- *(optional)* Set emacs as the default editor: `update-alternatives --config editor`
- Install sudo: `apt-get install sudo`
- Use `visudo` to edit the file `/etc/sudoers/` as below

  ```
  XXX ALL=(ALL) PASSWD: ALL , NOPASSWD: /usr/bin/rpm, /opt/freescale/ltib/usr/bin/rpm
  ```

  Replace *XXX* with the name of your user account. The above line should be added as the *last line* of the file. This will let you properly run Freescale's Linux Target Image Builder (LTIB) tool without constantly entering your password, thus facilitating unattended builds.

2.3 Prepare for LTIB Install

Make sure you are *not* the `root` user, then run the following commands:

```
sudo apt-get install build-essential
sudo apt-get install nfs-kernel-server
sudo apt-get install nfs-common
sudo apt-get install portmap
sudo apt-get install tftpd-hpa
sudo apt-get install tftp
sudo apt-get install rpm
sudo apt-get install wget
sudo apt-get install bison
sudo apt-get install flex
```

```
sudo apt-get install ncurses
sudo apt-get install ncurses-dev
sudo apt-get install libncurses5
sudo apt-get install libncurses5-dev
sudo apt-get install tcl
sudo apt-get install tclreadline
sudo apt-get install zlib1g
sudo apt-get install zlib1g-dev
sudo apt-get install liblzo2-2
sudo apt-get install liblzo2-dev
sudo apt-get install uuid
sudo apt-get install uuid-dev
sudo apt-get install libuuid1
sudo apt-get install gettext
sudo apt-get install libgtk2.0-dev
sudo apt-get install libdbus-glib-1-dev
sudo apt-get install liborbit2-dev
sudo apt-get install intltool
sudo apt-get install ccache
sudo apt-get install libtool
sudo apt-get install tcl

# Because we are on a 64-bit machine
sudo apt-get install ia32-libs
sudo apt-get install libc6-dev-i386
sudo apt-get install lib32z1

su # become root
perl -MCPAN -eshell
install CPAN
reload CPAN
install LWP::UserAgent
exit # exit Perl

exit # exit su
whoami # should be you, not root
```

2.4 Install LTIB

Follow the instructions in the *i.MX28 Linux BSP User Guide*. Once everything is unpacked do:

```
./ltib  # this will take awhile
./ltib --selectype
        # choose platform type  -->  imx28
        # choose packages profile  -->  mfg firmware profile
        # save and exit
```

Assuming ltib completes, you will now have two new files `updater.sb` and `updater_ivt.sb` in the ltib directory.

2.5 TFTP/NFS/Etc.

You will need the development machine to run some network services so that you can easily move files back and forth between the host and kit.

2.5.1 TFTP Server

- Run the following commands to create a directory named `/tftpboot` with global access. You can dump items here and then download them to the kit.

  ```
  sudo mkdir /tftpboot
  sudo chmod -R 777 /tftpboot
  sudo chown nobody /tftpboot
  ```

- Edit `/etc/default/tftp-hpa` to include:

  ```
  RUN_DAEMON="yes"
  ```

- Start the service: `sudo /etc/init.d/tftpd-hpa start`

- Test the service:

  ```
  echo hello > /tftpboot/hello.txt
  cd
  tftp localhost
  get hello.txt
  quit
  cat hello.txt
  ```

2.6 Dealing With the Serial Port

The firmware on the kit reports itself as *Freescale USB-to-Serial*. Check `/dev/tty*` to figure out where it is on your machine.

2.6.1 Using Cu

This is an example of using the *cu* program to communicate with the kit.

```
sudo chown uucp.uucp /dev/ttyACM0
sudo usermod -a -G uucp mikee
cu -l /dev/ttyACM0 -s 115200
~. # to exit
```

2.6.2 Using Screen

You can also use the *screen* program.

```
screen /dev/ttyACM0 115200
Ctl-A Ctl-\
```

3 Loading the Pre-Built Software from Digi

These instructions were mostly taken from the README file.

3.1 Necessary Items

We need the following items:

imx28_ivt_uboot_v3.sb This is the *bootlet* that gets passed to the iMX28.

When the iMX28 is reset, it executes its ROM. There is *no* alternative, no other code is permitted to handle the reset exception. The ROM code reads boot mode pins to detect the boot source (USB, SD/MMC, NAND, etc.) and negotiates with that source to retrieve a *boot stream*. A boot stream is an executable collection of bytes in *Safe Boot* (SB) format. This file is the boot stream for loading uboot.

Note, the *ivt* in the filename indicates that the *High Assurance Boot* (HAB) is operating. Specifically, that the `HAB_DISABLE` bit (`HW_OCOTP_ROM7:0x8002C210:bit11`) on the iMX28 is zero (0).

h3_flash_init This is a uboot initialization script. See the section below on modifying this to your needs.

uImage_heg This is the Linux kernel image.

rootfs_XXX.ubi In my case, this was `rootfs_3893.ubi`. This is a root file system image in UBIFS (Unsorted Block Image File System) format.

demo.tgz The demonstration software.

3.2 Overview of Process

The overall process is to tell the kit to boot from USB. This is done by setting `SW1-2` to the `ON` position. When the iMX28 powers up, its ROM code will detect that setting and attempt to retrieve a boot stream via USB. On the other end of the USB, is a program named `sb_loader.exe`. This program is part of Freescale's *iMX Manufacturing Tools bundle*, which can be found by starting at the i.MX28 Product Page and then following the link to the i.MX Manufacturing Toolkit. There are a few versions available. These instructions correspond to `Mfgtools-Rel-1.6.2.032.zip`.

Once the iMX28 has established contact with the sb_loader program over USB, we need to send the processor a *boot stream*. We will send it a version of the u-boot bootloader in safe boot format contained in file `imx28_ivt_uboot_v3.sb`.

Once the iMX28 is running u-boot, we want to prepare the board for hosting Linux. This is the job of the `h3_flash_init` script. This script will setup a proper u-boot environment, load the Linux kernel, and flash the root file system.

Once the Linux kernel is running, program the boot stream *(imx28_ivt_uboot_v3.sb)* into NAND flash. That way, when the kit is booted with `SW1-2` in the `OFF` position, the iMX28 will find a boot stream.

3.3 Modify the U-Boot Script (h3_flash_init)

Here we want to modify the environment that will be programmed into u-boot. The script just automates the creation of the environment, we could enter its contents by hand if we needed to.

- Install necessary tools: `sudo apt-get install uboot-mkimage`
- Unpack the u-boot source code

  ```
  tar xzf u-boot-2011.09.tgz
  cd u-boot-2011.09/
  ```

- Modify `h3_flash_init.txt`. I changed `bootargs_net` to derive NFS options from environment variables as in:

  ```
  nfsroot=${serverip}:${nfsroot}
  ```

- Rebuild the boot script (this just puts a header on the contents so u-boot knows how to parse it).

```
mkimage -T script -C none -n 'Uboot Init Script' \
-d u-boot-2011.09/h3_flash_init.txt /tftboot/h3_flash_init
```

3.4 Procedure Detail

This is the detailed procedure for booting the kit.

1. Ensure that your TFTP server is properly setup. Ours is at `/tftpboot/`.
2. Copy the u-boot script, kernel, root filesystem, and demonstration software into the TFTP server root.

   ```
   cp  h3_flash_init     /tftpboot
   cp  uImage_heg        /tftpboot
   cp  rootfs_3893.ubi   /tftpboot
   cp  demo.tgz          /tftpboot
   ```

3. On a Windows machine, install the *Freescale iMX Manufacturing Tools*.
4. On the Windows machine, in the same directory where the program `sb_loader.exe`, was installed, copy the u-boot safe boot-formatted boot stream file *(imx28_ivt_uboot_v3.sb)*.
5. On the Windows machine, open a command prompt and change to the directory where `sb_loader.exe` was installed.
6. On the hardware, set `SW1-1` to `OFF` and `SW1-2` to `ON`. This will tell the iMX28 to boot from USB.
7. Connect two USB cables to the kit. One to `J5` for debug message and one to `J7` for the iMX28 boot stream.
8. Connect the USB cables to the Windows box. Using the Device Manager, determine which COM port the debug messages will be connected to. Open a terminal emulator on that COM port.
9. Power on the kit.
10. Execute the u-boot boot stream by entering the command below into the Windows command prompt.

    ```
    sb_loader.exe /f imx28_ivt_uboot_v3.sb
    ```

11. Verify that debug messages are appearing in the terminal emulator.
12. Interrupt u-boot by pressing any key before the 3-second timeout.
13. Adjust the u-boot environment to reflect your network settings:

```
printenv  # note current settings

setenv  ethaddr  00:04:9F:01:03:29 # adjust to your network
setenv  serverip  10.160.42.128     # adjust to your network
setenv  bootfile  h3_flash_init
setenv  tftp_kernel  uImage_heg
setenv  tftp_rootfs_ubifs  rootfs_3893.ubi
setenv  tftp_prefix  # you may or may not need this
saveenv
```

Note the *tftp_prefix* environment variable. Depending on your setup, you may or may not need this. We did not.

14. Execute the boot script

```
dhcp
source
```

You will see a lot of messages come by the debug terminal. If all goes well, you will see the kernel and the root file system being downloaded and burned into flash.

15. Login as *root* with password *root*

16. Flash the bootloader

```
cd  /boot
kobs-ng  init  imx28_ivt_uboot_v3.sb
sync
```

17. Flash the demonstration software

```
cd  /heg
dhclient  eth0
tftp  -g  -r demo.tgz  <your dev server ip>

# At this point you might want to check the file integrity
# using the md5sum command on the TFTP server and the device

tar xzf demo.tgz
rm demo.tgz
sync
```

18. *Properly shutdown* the board: `poweroff`

19. Power off the board

20. Set SW1-2 to OFF so that the next time the iMX28 boots, it will look for a boot stream in NAND flash rather than on USB.

After this point, you should be able to leave the Windows box and connect to the kit's debug serial port using the cu program:

```
cu -l /dev/ttyACM0 -s 115200
~. # to exit
```

Assuming the kit is setup to configure a network interface and launch dropbear, you can also just SSH into it.

4 Porting the iDigi Connector

4.1 Download and Extract the Connector

```
curl -o idigi-connector.tgz \
http://ftp1.digi.com/support/sampleapplications/40003007_E.tgz

md5sum idigi-connector.tgz
07916d03dfa8a8647e1475e3a189d002  idigi-connector.tgz

tar xzf idigi-connector.tgz
```

4.2 Adjust the Platform Code

4.2.1 Configuration Routines: public/run/platforms/linux/config.c

Edit the file:

- Comment out (or remove) all `#error ...` preprocessor directives
- hard-code the MAC Address in the function `app_get_mac_addr`

4.2.2 Sample Configuration public/run/samples/xxx/idigi_config.h

Edit the file:

- Add `ENABLE_COMPILE_TIME_DATA_PASSING 1`
- Hardcode the Vendor ID to match your organization `#define IDIGI_VENDOR_ID 0x0...`

4.2.3 Makefiles `public/run/samples/xxx/Makefile`

Edit the file:

- Remove `CC = gcc` at the top of the file *(we will override this in our environment)*
- Remove the following `CFLAGS` as they will cause errors
  - `-Werror`
- Add the following CFLAGS (at the end of the CFLAGS definition) to keep them from cluttering up our warnings
  - `-Wno-padded`
  - `-Wno-cast-align`

4.2.4 Setup the Cross-Compilation Environment

You need to make sure that building the iDigi connector uses the cross compilers provided by LTIB. To keep things simple and repeatable, we recommend using a shell script to do this. Ours is named `setenv.sh` and kept in the home directory. When begining work, we open up a terminal and then `source ~/setenv.sh` to create the proper environment. This file is repeated below:

```
#!/bin/sh
## You need to 'source' this script
echo "Setup cross environment"

TOOL_PATH=/opt/freescale/usr/local/gcc-4.4.4-glibc-2.11.1-multilib-1.0/arm-fsl-linux-gn
export CROSS_COMPILE=arm-linux-
export HOST=arm
export CC=${TOOL_PATH}/${CROSS_COMPILE}gcc
export CXX=${TOOL_PATH}/${CROSS_COMPILE}g++
export CPP=${TOOL_PATH}/${CROSS_COMPILE}cpp
export AS=${TOOL_PATH}/${CROSS_COMPILE}as
export LD=${TOOL_PATH}/${CROSS_COMPILE}ld
export AR=${TOOL_PATH}/${CROSS_COMPILE}ar
export RANLIB=${TOOL_PATH}/${CROSS_COMPILE}ranlib
export NM=${TOOL_PATH}/${CROSS_COMPILE}nm
export OBJCOPY=${TOOL_PATH}/${CROSS_COMPILE}objcopy
export STRIP=${TOOL_PATH}/${CROSS_COMPILE}strip

export LTIB_BASE=${HOME}/linux-source/ltib
export ROOTFS=${LTIB_BASE}/rootfs
```

```
export CFLAGS="-I${ROOTFS}/usr/include"
export LDFLAGS="-L${ROOTFS}/usr/lib -L${ROOTFS}/lib"
```

4.3 Testing the Connector

You should do the steps above for the following project in `idigi/public/run/samples/`. Note, you should test these in the order listed, consulting the iDigi connector documentation as necessary.

- `compile_and_link`
- `connect_to_idigi`
- `send_data`

If the above sample projects work, then your development environment should be properly configured.

5 Demonstration Code

We use some code in `idigi/public/run/samples/hh_data_stream` as the application to connect to iDigi. This contains logic to count the number of active panic buttons during the last 30-second period.

You may find this application in the file `hhh_data_stream.tar.gz`. To use it, perform the following steps:

- Extract the archive into the `idigi/public/run/samples/` directory
- Ensure your organization's iDigi Vendor ID is inserted into `idigi_config.h`
- Setup the proper cross-compilation environment as discussed above
- `make clean`
- `make`

The demonstration attempts to report the number of active panic buttons during 30-second intervals. You should consult the source code for complete information. The main sections are highlighted below.

5.1 Active Panic Buttons

The number of active panic buttons is computed by the `active_buttons()` function inside `pb_status.c`. By going through the PB Manager code in *zorglub* you can determine that performing the IOCTL `IOCTL_ISM_GET_STAT` on the file `/dev/ism` will give you some idea of the number of panic buttons currently associated with the HHH and the state that they are in.

```
icarm_fd = open("/dev/ism", O_RDONLY)) < 0 );
ioctl(icarm_fd, IOCTL_ISM_GET_STAT, stat_table);

/ For items in stat_table, if bit-1 is set, the button is active. /
```

5.2 Main Loop

The program's main loop occurs in the function `application_run()` inside `application.c`. This repeatedly calls `app_send_put_request()` and then sleeps.

5.3 Sending Data

The function `app_send_put_request()` calls `active_buttons()` and includes its return value in the data streamed to iDigi. This function also sends some sample, made-up data such as; pulse, breaths, systolic, and diastolic to make the demonstration more interesting.

6 Daemonizing the iDigi Connector

You may want to run the iDigi connector as a system daemon. This can be easily done. In the file `idigi/public/run/platforms/linux/main.c`, add a function named `daemonize()` as shown below:

```
#define RUNNING_DIR     "/tmp"
#define LOCK_FILE       "idigi.lock"
#define LOG_FILE        "idigi.log"

void daemonize(void)
{
        int pid, i, lfp;
        char str[10];
        if ( 1 == getppid() )
                return; / already a daemon /
```

```
pid = fork();
if ( pid < 0 )
        exit(1); / fork error /
if ( 0 != pid )
        exit(0); / parent exits /
/ child (daemon) continues /
setsid(); / become a session leader /

signal(SIGHUP, SIG_IGN);
if ( 0 != (pid = fork()) )
        exit(0); / first child terminates /

/ second child continues /
chdir(RUNNING_DIR); / change working directory /
umask(0); / clear file mode creation mask /

/ close any potentially open files /
for ( i = getdtablesize(); i >= 0; —i )
        close(i);

/ open /dev/null and give it to STDIN/STDOUT/STDERR /
i = open("/dev/null", O_RDWR); / stdin /
dup(i); / stdout /
dup(i); / stderr /

lfp = open(LOCK_FILE, O_RDWR|O_CREAT, 0640);
if ( lfp < 0 )
        exit(1); / could not open lock-file /
if ( lockf(lfp, F_TLOCK, 0) < 0 )
        exit(0); / can not lock /

/ record PID in the lock file /
sprintf(str, "%d\n", getpid());
write(lfp, str, strlen(str));

/ ignore several signals /
signal(SIGCHLD, SIG_IGN);
signal(SIGTSTP, SIG_IGN);
signal(SIGTTOU, SIG_IGN);
signal(SIGTTIN, SIG_IGN);

/ catch hangup and kill signals:
```

```
        You may define another function named signal_handler to
        respond to SIGHUP and SIGTERM if you like. /
    / signal(SIGHUP, signal_handler); /
    / signal(SIGTERM, signal_handler); /
}
```

Then rename the existing iDigi `main()` function to `old_main()` and make a new definition as below:

```
int main(void)
{
        APP_DEBUG("PREPARING TO DAEMONIZE\n");
        daemonize();
        old_main ();
        APP_DEBUG("DAEMON END\n");
        return 0;
}
```

7 Starting the iDigi Connector

To start the iDigi Connector, insert it into the boot order after the panic button manager starts. The boot order is listed below.

- `/etc/rc.d/rcS` calls `/etc/rc.d/rc.local` if it exists
- `/etc/rc.d/rc.local` calls `/heg/demo/local/etc/rc.heg` if it exists
- Towards the bottom of `/heg/demo/local/etc/rc.heg`, the script figures out what it should do based on the value of the environment variable `HEG_DEMO_MODE` - which is set in `/etc/rc.d/rc.conf` to `H3`
- *H3-mode* causes the script to launch; postgres, ui, bt, panic-button manager, and others
- Start the iDigi connector *after* the panic-button manager (pb_{mgr}) so that the sample application can open `/dev/ism` and perform IOCTLs on it