Parallel Programming • KVM • Home Assistant • I2C Display • Archlinux

ODROID Magazine

Year Four
Issue #46
Oct 2017

Keep your files secure and available for all of your personal applications

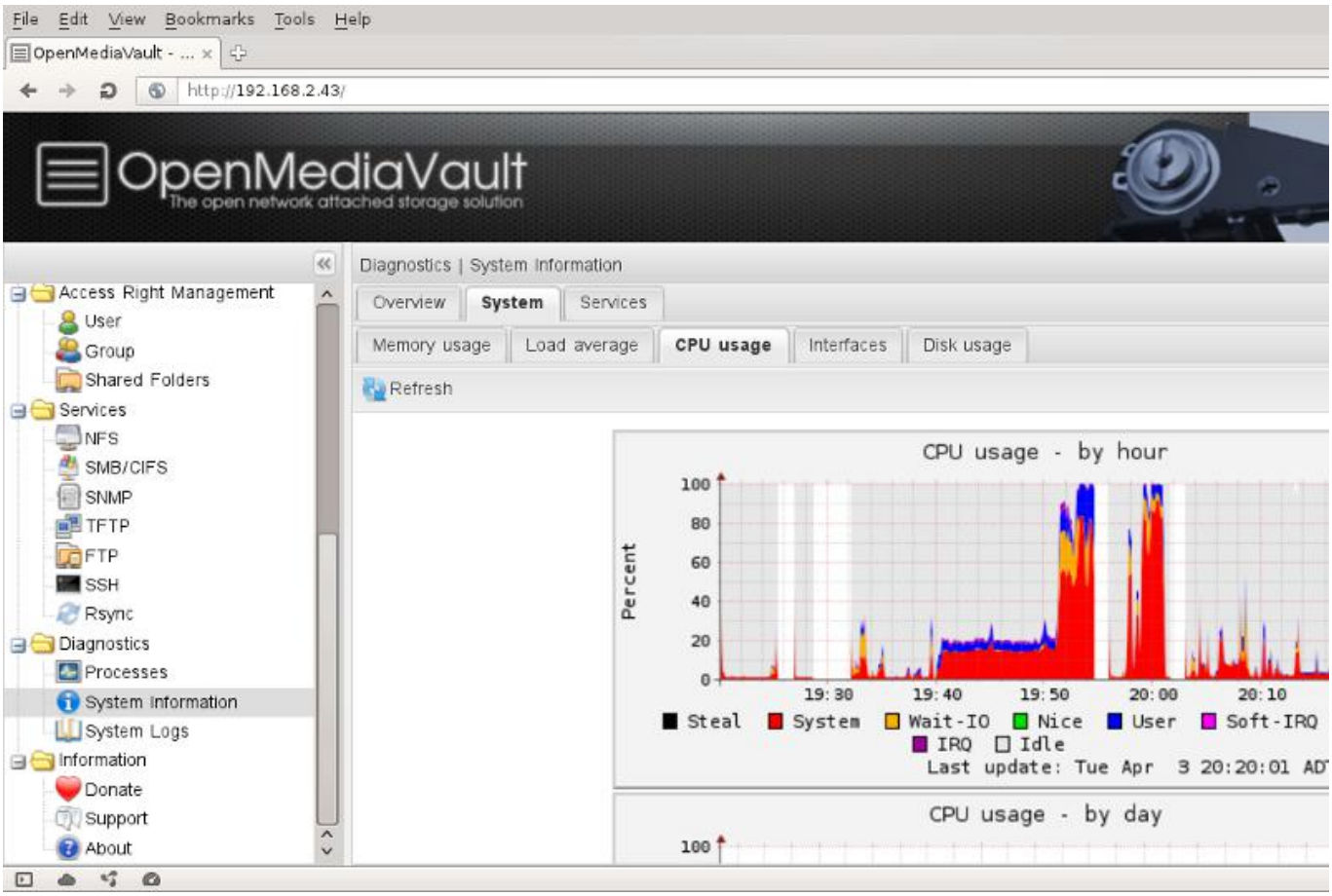
BUILD YOUR OWN *Multimedia* HOME SERVER

A Swarm of Swarms:

- ODROID-C2 Docker Swarm
- ODROID-MC1 Docker Swarm

Build Your Own Home Server: Storing A Large Amount Of Multimedia Files

October 1, 2017 By odroidinc.com Linux, Tinkering



Why would you need a network attached storage (NAS) server at home?

- Automatic backup of smartphone data
- Manage and share data on the Internet
- Stream saved videos
- Download and manage Torrents on a smartphone
- Host a personal blog
- Enable SSL for security

Required components:

- Internet service
- A WiFi router
- A typical computer or a laptop, such as a MacBook Pro
- ODROID-HC1 and its power supply
- MicroSD card for the operating system
- LAN cable to connect between WiFi router and ODROID-HC1
- Hard Disk Drive (2.5inch) for my multimedia data

Figure 1 – Home server using ODROID-HC1

You also need a little bit of understanding of the operating system as well as Open Media Vault (**www.openmediavault.org**), which will allow everyone to install and administrate a Network Attached Storage without deeper knowledge.

Preparation

First, download Open Media Vault (OMV) for ODROID-HC1 from **http://bit.ly/2xogExP** to your computer. Refer to the readme.txt file for the username and password.

web interface username = admin
web interface password = openmediavault
console/ssh username = root
console/ssh password (3.0.75+) = openmediavault

Figure 2 – Downloaded Open Media Vault image

Next, use a USB adapter with an 8GB microSD card, then open Etcher (etcher.io) to flash the operating system, as shown in Figure 3. Make sure to unzip the .7z file before selecting it in Etcher.

Figure 3 – Inserting the USB adapter and microSD card in the computer

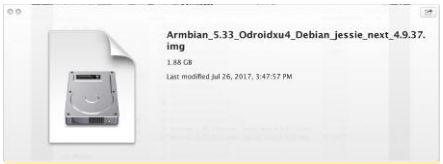


Figure 4 – The unzipped image file has a different filename than the .7z file

Figure 5 – Etcher allows you to write pre-built images to a microSD card

General configuration

Insert the completed Open Media Vault image into the ODROID-HC1, then slide and insert the hard disk drive to the SATA connector. Connect the LAN cable from the WiFi router to HC1 and plug the power supply to turn it on. It will take approximately 10 minutes for the first boot. With another LAN cable, connect the computer to the same WiFi router which is connected to the HC1.

Next, download and install Angry IP Scanner (http://bit.ly/2wCMeII) and scan the IP addresses of the connected devices. The Hostname is shown as odroidxu4.local. Open a browser and enter the ODROID-HC1 address.

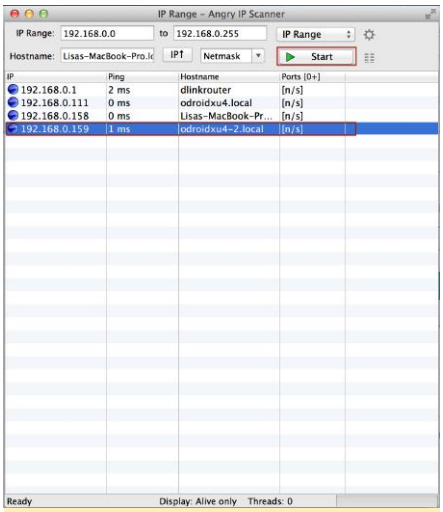


Figure 6 – Scanning the local IP addresses to locate the IP address of the ODROID-HC1

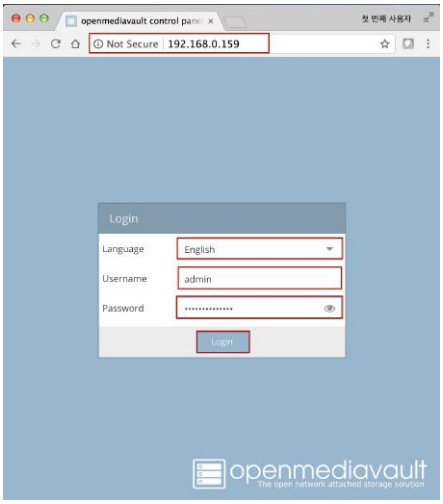


Figure 7 – Logging into the web interface of Open Media Vault

As mentioned above, the default username and password is in the readme.txt at http://bit.ly/2xogExP.

web interface username = admin
web interface password = openmediavault

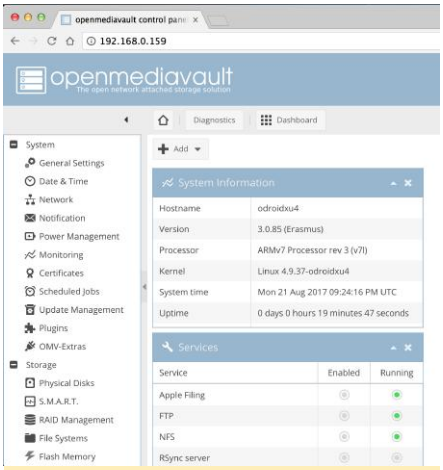


Figure 8 – Home screen of the Open Media Vault web interface

Go to "System -> Date & Time" and change the timezone to your current location, then press "Activate [Use NTP server] -> Save -> Apply".

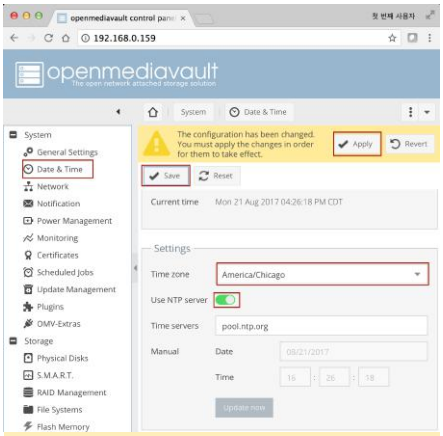


Figure 9 – Updating the date and time in Open Media Vault

You can also change the session timeout to "0" in order not to be logged out after a certain amount of idle time by selecting "General Settings -> Session timeout -> 0 -> Save -> Apply -> Yes".

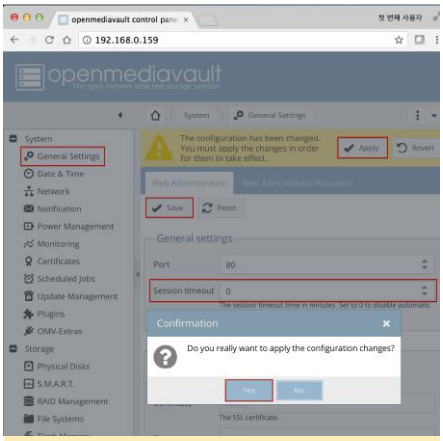


Figure 10 – Saving the configuration changes in Open Media Vault

Next, update the system to the latest version by selecting "Update Management -> Check Package information -> Upgrade", reload the page after the update completes, then reboot the ODROID-HC1 using the "Reboot" option in the Open Media Vault web interface.

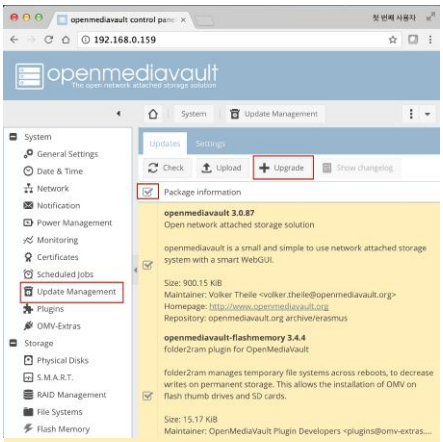


Figure 11 – Updating to the latest version of Open Media Vault

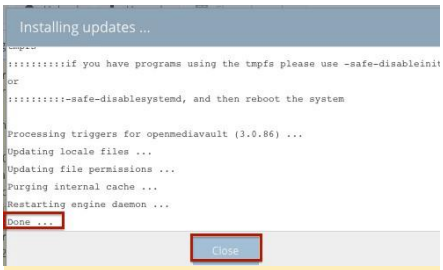


Figure 12 – The Open Media Vault update has been completed

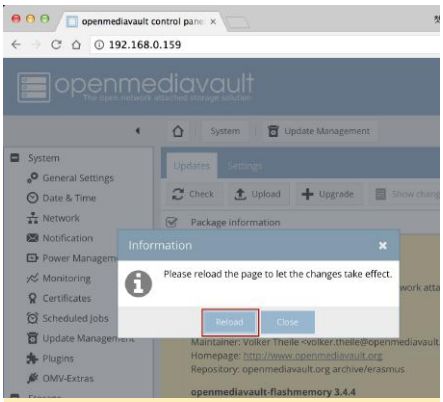


Figure 13 – The page should be reloaded after the Open Media Vault update completes

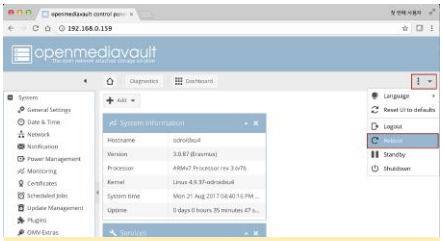


Figure 14 – Select "Reboot" from the Open Media Vault web interface

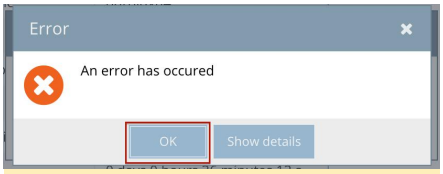
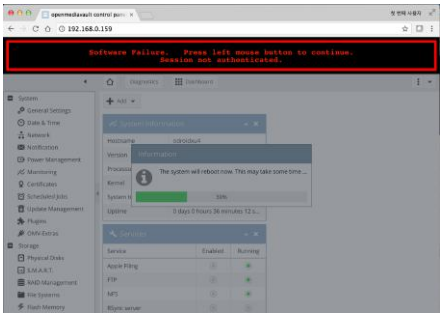


Figure 15 and 16 – Ignore the error messages after pressing "Reboot"



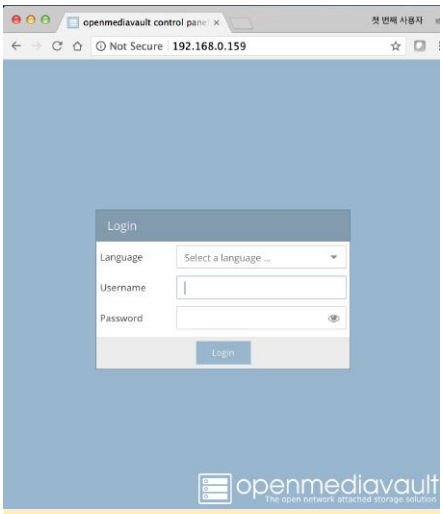
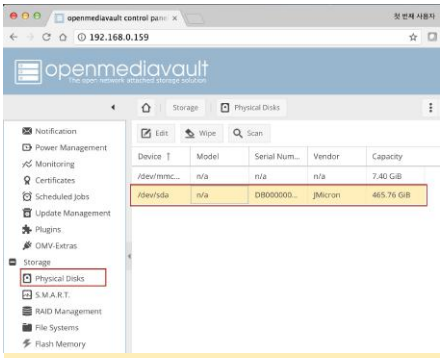


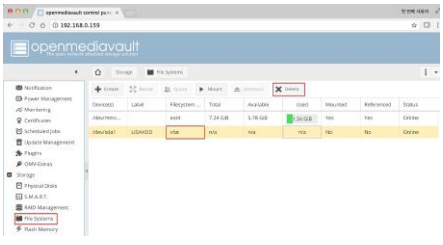
Figure 17 – Login to the Open Media Vault web interface after the reboot has completed

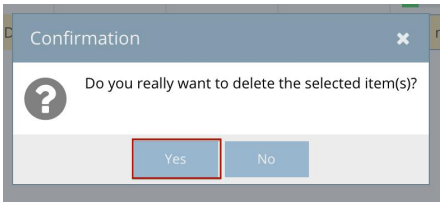
Setting permissions

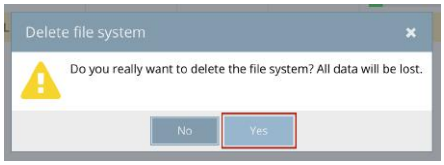
The hard drive needs to be in ext4 format in order to be compatible with Open Media Vault. If the file system of the hard drive is not ext4, you will need to create a new file system, as shown in Figure 17)

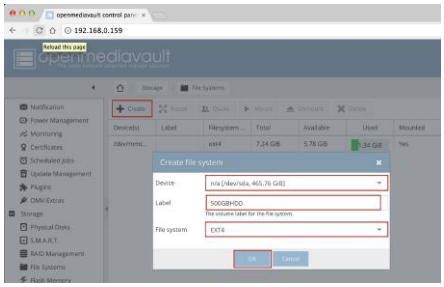


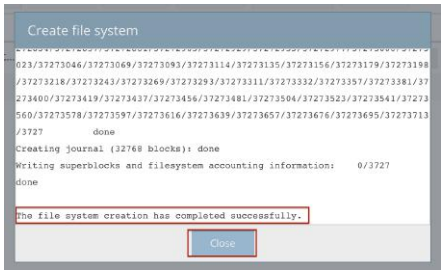
Figures 18 – 24 – Formatting the hard drive to ext4 format



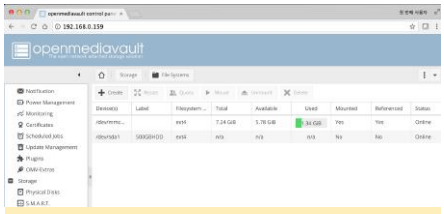




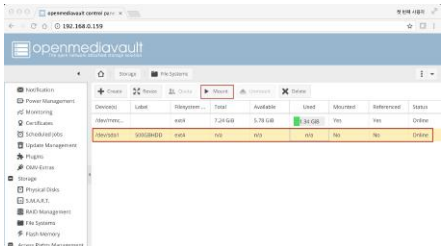


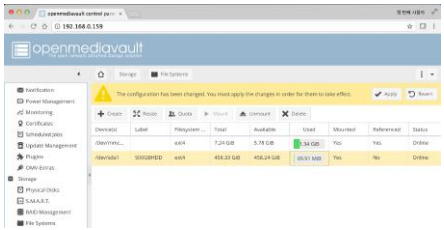


After the format has completed, select “Mount” as shown in Figures 19 and 20.

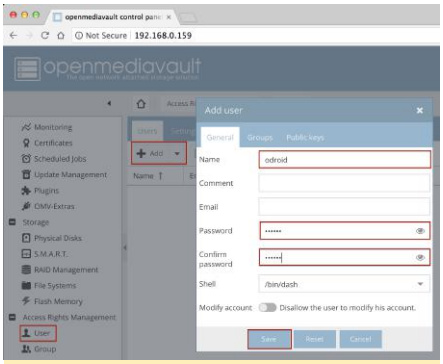


Figures 25, 26 and 27 – Mounting the newly formatted hard drive

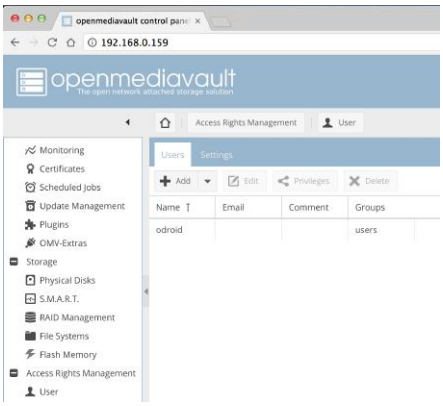




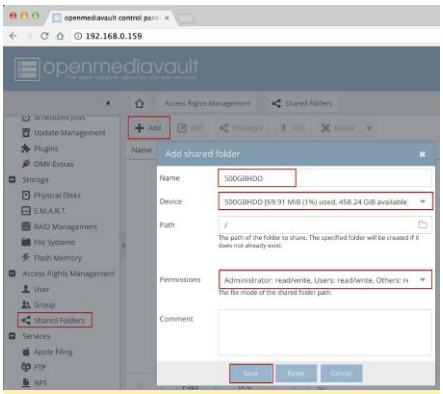
The next step is to register users who have permissions to transfer data to/from the server.



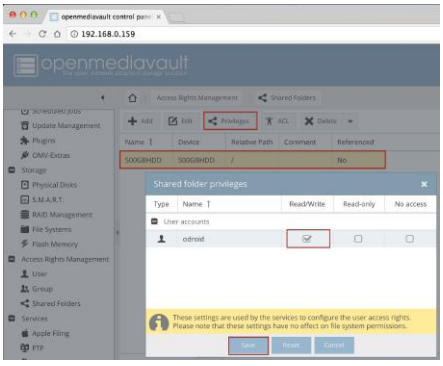
Figures 28 and 29 – Registering the “odroid” user to be able to transfer data to/from the server



After the user has been created, create a shared folder by selecting “Shared Folders -> Add -> Name -> Select Device -> Set Permissions -> Save”. Each user then needs to be granted privileges. Grant the user “odroid” shared read/write folder privileges and save the settings.



Figures 30 and 31 – Creating the shared folder and assigning individual user privileges



ACL is another type of permission that needs to be granted, as described at **http://bit.ly/2xn98sb**. The user “odroid” needs read/write/execute permissions, and other users can be given permissions as needed.

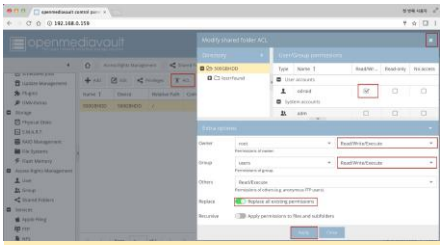
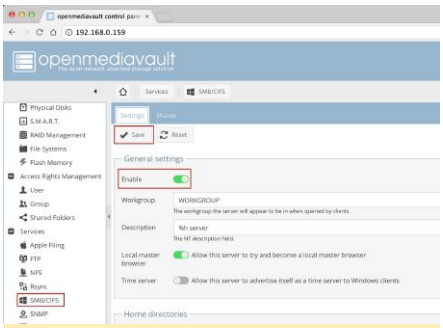


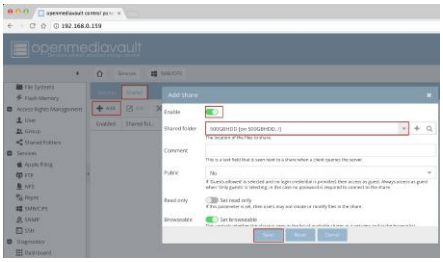
Figure 32 – Giving ACL permissions to the "odroid" user

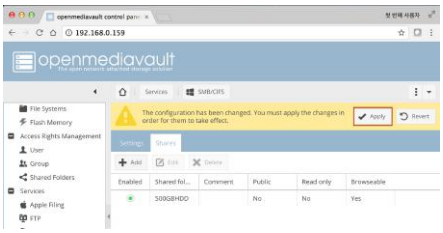
Data transfer using Samba

The server can be shared with the workgroup using Samba (SMB). Click "Apply" to see the shared folder.



Figures 33, 34 and 35 – Sharing the server using Samba





Note that if you have two or more of the same shared devices or folders, your computer may rename one for you. For example, if you have two ODROID-HC1s attached to the router, it will recognize the first as odroidxu4 and name the second one, odroidxu4–2, to differentiate the two. If you do not see the two automatically, try rebooting your computer.

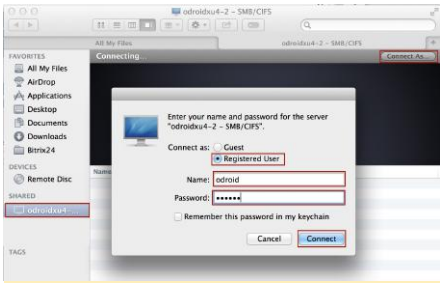


Figure 36 – Accessing the shared folder from a networked computer

Open Finder and check "Shared" to see the odroidxu4 shared server, which is the ODROID-HC1. Click "Connect As" and enter the name and password which matches the username and password that was created on the server.

After connecting, files and folders can be transferred to and from the ODROID-HC1 server.

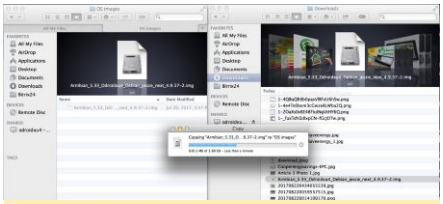


Figure 37 – Copying files and folders to the ODROID-HC1 using Samba

Data transfer using FTP

File Transfer Protocol (FTP) is a standard network protocol used for the transfer of coputer files between a client and server on a computer network. First, enable FTP on Open Media Vault as shown in Figure 38.

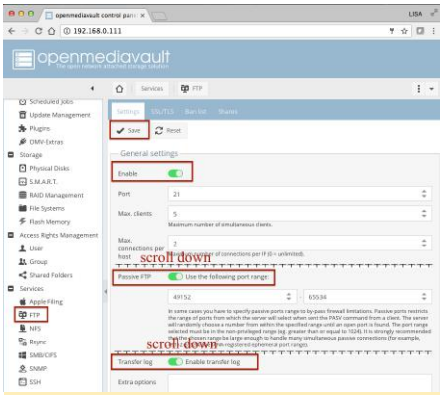


Figure 38 – Enabling FTP in Open Media Vault

Next, enable the shared folder by selecting "Services -> FTP -> Shares -> Add -> Enable -> select Shared folder -> Save".

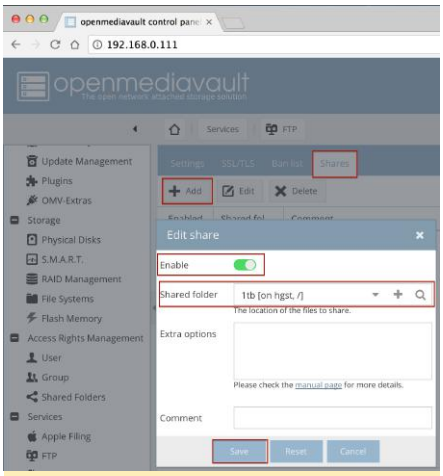


Figure 39 – Selecting the shared FTP folder in Open Media Vault

After FTP is enabled, files can be transferred to/from the server by visiting ftp://192.168.0.111 in a browser, using the address of the ODROID-HC1 server in place of 192.168.0.111.

Figures 40 and 41 – Visiting the Open Media Vault server via FTP using Firefox

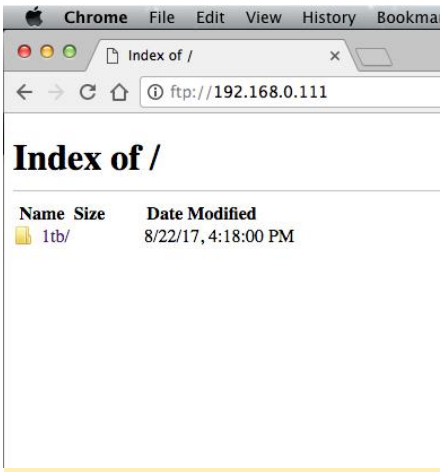
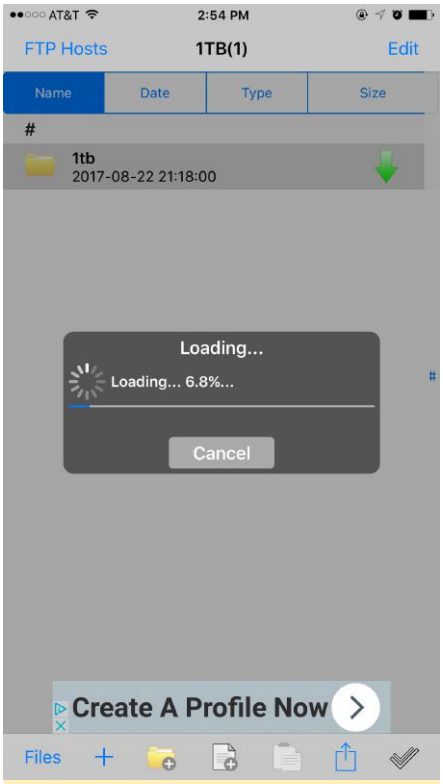
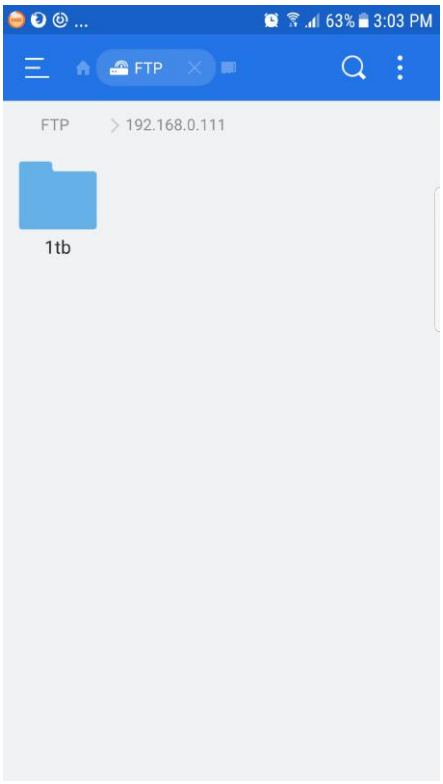


Figure 42 – Visiting the Open Media Vault server via FTP using Chrome

Next, install FTP on your smartphone, using an app such as FTP Sprite for iPhone, or ES File Explorer for Android.

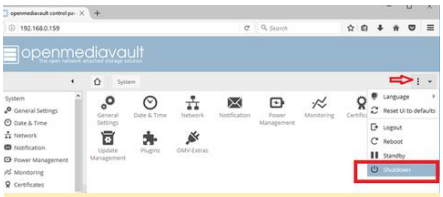


Figures 43 and 44 – Accessing the Open Media Vault server using FTP on a smartphone

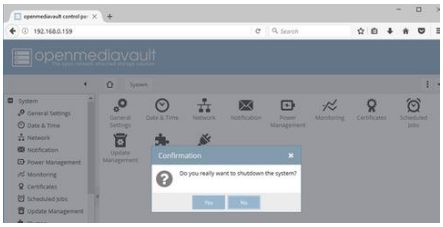


Shutdown

On your Open Media Vault web interface, below the banner, click the three vertical dots on the right, and select “Shutdown”.



Figures 45 and 46 – Shutting down the server via the Open Media Vault menu



When the screen shown in Figure 47 appears, your operating system has stopped running, and the blue blinking LED should be off on the ODROID-HC1. A this point, you can unplug the power supply and remove the microSD card. Follow this shutdown procedure anytime you need to change the hard drive, update the operating system on the microSD card, or unplug the power. This will help avoid damaging the ODROID-HC1.

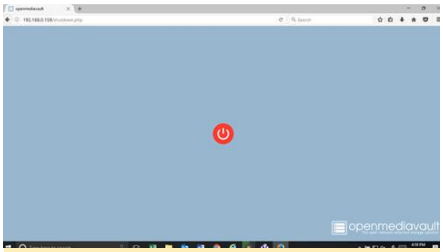


Figure 47 – Post-shutdown screen in Open Media Vault

For comments, questions, or suggestions, please visit the original article at https://medium.com/p/6a3771d9172.

KVM On The ODROID-XU4

October 1, 2017 By Brian Kim ODROID-XU4, Tutorial

This is a step-by-step guide for enabling KVM on an ODROID-XU4. This guide is only available in u-boot odroidxu4-v2017.05 and Linux kernel 4.9.x versions. The first step is to rebuild the kernel. KVM needs the arch timer instead of MCT (Multi-Core Timer), which is the default timer of ODROID-XU4 (by exynos5422-odroidxu4-kvm.dtb). And there are the virtualization related configurations in odroidxu4_kvm_defconfig file.

```
$ sudo apt update
$ sudo apt install git
$ git clone --depth 1
https://github.com/hardkernel/linux -b
odroidxu4-4.9.y
$ cd linux
$ make odroidxu4_kvm_defconfig
$ make -j8
$ sudo make modules_install
$ sudo cp arch/arm/boot/zImage
/media/boot/zImage_kvm
$ sudo cp arch/arm/boot/dts/exynos5422-
odroidxu4-kvm.dtb /media/boot/
```

Modify the boot.ini file by changing “zImage” to “zImage_kvm”, and “exynos5422-odrooidxu4.dtb” to “exynos5422-odrooidxu4-kvm.dtb”

/media/boot/boot.ini

```
(......)
# Load kernel, initrd and dtb in that
sequence
fatload mmc 0:1 0x40008000 zImage_kvm
(......)
if test "${board_name}" = "xu4"; then
fatload mmc 0:1 0x44000000 exynos5422-
odrooidxu4-kvm.dtb; setenv fdtloaded "true";
fi
(......)
```

Reboot the ODROID-XU4, then check whether KVM is enabled after the booting process is finished:

```
$ dmesg | grep HYP
[ 0.096589] CPU: All CPU(s) started in HYP
mode.
[ 0.777814] kvm [1]: HYP VA range:
c0000000:ffffffff
$ dmesg | grep kvm
[ 0.777771] kvm [1]: 8-bit VMID
[ 0.777793] kvm [1]: IDMAP page: 40201000
[ 0.777814] kvm [1]: HYP VA range:
c0000000:ffffffff
[ 0.778642] kvm [1]: Hyp mode initialized
successfully
[ 0.778713] kvm [1]: vgic-v2@10484000
[ 0.779091] kvm [1]: vgic interrupt IRQ16
[ 0.779127] kvm [1]: virtual timer IRQ60
$ cat /proc/interrupts | grep arch_timer
58: 0 0 0 0 0 0 0 0 GIC-0 29 Level
arch_timer
59: 0 1857 1412 1345 16986 6933 5162 3145
GIC-0 30 Level arch_timer
```

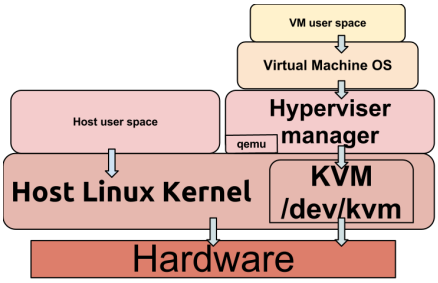


Figure 1 Virtual Machine architecture

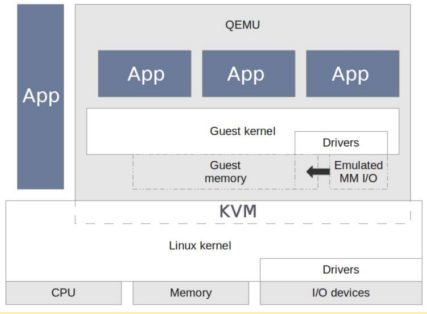


Figure 2 – Virtual Machine architecture

Ubuntu Minimal 16.04.3 Running using QEMU and KVM/ARM

To follow this section, make sure that KVM is already enabled, with 4GB or more storage space available. In this section, we will run the Ubuntu Minimal 16.04.3 image on the virtual machine using QEMU and KVM/ARM.

To begin, Install qemu-system-arm which is to virtualize the arm machine and required packages:

```
$ sudo apt update
$ sudo apt install qemu-system-arm kpartx
```

Next, prepare the guest OS kernel and dtb images. It is needed to set clock frequency for timer in dts file by adding a "clock-frequency = <100000000>;" line in the timer node).

```
$ wget
https://www.kernel.org/pub/linux/kernel/v4.x
/linux-4.13.tar.xz
$ tar Jxvf linux-4.13.tar.xz
$ cd linux
$ nano arch/arm/boot/dts/vexpress-v2p-ca15-
tc1.dts
```

arch/arm/boot/dts/vexpress-v2p-ca15-tc1.dts

```
timer {
compatible = "arm,armv7-timer";
interrupts = <1 13 0xf08>,
<1 14 0xf08>,
<1 11 0xf08>,
<1 10 0xf08>;
clock-frequency = <100000000>;
};
```

Build and copy zImage and dtb images to the working directory:

```
$ make vexpress_defconfig
$ make menuconfig
```

Enable the block layer —>
[*] Support for large (2TB+) block devices and files

```
$ make zImage dtbs -j8
$ cp arch/arm/boot/zImage ../
$ cp arch/arm/boot/dts/vexpress-v2p-ca15-
tc1.dtb ../
$ cd ..
```

Prepare Ubuntu minimal root filesystem image by downloading the Ubuntu minimal 16.04.3 image and generate the root filesystem image from the image.

```
$ wget
https://odroid.in/ubuntu_16.04lts/ubuntu-
16.04.3-4.9-minimal-odroid-xu4-
20170824.img.xz
$ unxz ubuntu-16.04.3-4.9-minimal-odroid-
xu4-20170824.img.xz
$ sudo kpartx -a ubuntu-16.04.3-4.9-minimal-
odroid-xu4-20170824.img
$ sudo dd if=/dev/mapper/loop0p2 of=ubuntu-
minimal-16.04.3.img
$ sudo kpartx -d ubuntu-16.04.3-4.9-minimal-
odroid-xu4-20170824.img
```

Modify the root filesystem for the guest environment by removing the ODROID-specific file and configuration:

```
$ mkdir rootfs
$ sudo mount ubuntu-minimal-16.04.3.img
rootfs
$ cd rootfs
$ sudo rm ./first_boot
$ sudo rm ./etc/fstab
$ sudo touch ./etc/fstab
$ cd ..
$ sudo umount rootfs
```

Run qemu, where the host is Ubuntu Mate 16.04.3 / 4.9.50 kernel, and the guest is Ubuntu Minimal 16.04.3 / 4.13 kernel

```
$ qemu-system-arm -M vexpress-a15 -smp 2 -
cpu host
-enable-kvm -m 512 -kernel zImage -dtb
vexpress-v2p-ca15-tc1.dtb
-device virtio-blk-device,drive=virtio-blk
-drive file=ubuntu-minimal-
16.04.3.img,id=virtio-blk,if=none
-netdev user,id=user -device virtio-net-
device,netdev=user
-append "console=tty1 root=/dev/vda rw
rootwait fsck.repair=yes"
```

Figure 3 – The Host operating system runs the LTS Kernel 4.9.50 while the guest operating system runs the upstream Kernel 4.13

My ODROID-C2 Docker Swarm – Part 2: Deploying a Stack to a Swarm

October 1, 2017 By Andy Yuen Docker

In Part 1, I deployed services in my ODROID-C2 cluster using the Docker command line. It works, but there must be a better way to do deployment, especially when an application requires multiple components working together. Docker 1.13.x introduced the new Docker stack deployment feature to allow deployment of a complete application stack to the swarm. A stack is a collection of services that make up an application. This new feature automatically deploys multiple services that are linked to each other obviating the need to define each one separately. In other words, this is docker-compose in swarm mode. To do this, I have to upgrade my Docker Engine from V1.12.6 that I installed using apt-get from the Ubuntu software repository to V1.13.x. Having already built V1.13.1 on my ODROID-C2 when I was experimenting unsuccessfully with swarm mode months ago, as documented in my previous article, it is just a matter of upgrading all my ODROID-C2 nodes to V1.13.1 and I am in business.

The httpd-visualizer stack

The first thing I did was to deploy the same applications (httpd and Visualizer) as in my previous article using 'docker stack deploy'. To do this, I need to create a yaml file. This is actually docker-compose yaml file version "3". This is relative easy to do as data persistence is not required. Here is the yaml file:

```
version: "3"
services:
 httpd:
   # simple httpd demo
   image: mrdreambot/arm64-busybox-httpd
   deploy:
     replicas: 3
     restart_policy:
       condition: on-failure
     resources:
       limits:
         cpus: "0.1"
         memory: 20M
   ports:
     - "80:80"
   networks:
     - httpd-net
 visualizer:
   image: mrdreambot/arm64-docker-swarm-
visualizer
   ports:
     - "8080:8080"
   volumes:
     -
"/var/run/docker.sock:/var/run/docker.sock"
   deploy:
     placement:
       constraints: [node.role == manager]
   networks:
     - httpd-net
networks:
 httpd-net:
```

Note that the use of "Networks" in the yaml file is not strictly necessary. If omitted, dDocker will create a default overlay network as you will see in a later section. The 2 applications, in this case, do not need to talk to each other anyway! To deploy it, just change to the directory where the yaml file is located and issue the command:

```
$ docker stack deploy -c simple-stacks.yml
httpd-dsv
```

This creates a stack named httpd-dsv. You can find out regarding the state of the stack by issuing a number of stack commands as shown in Figure 1.

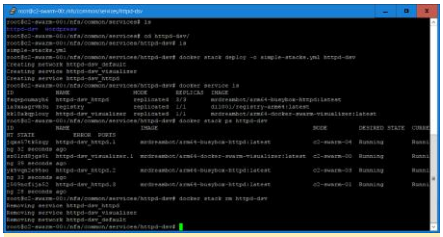


Figure 1 – httpd dsv stack commands

You can point your browser to the swarm manager or any swarm node at port 8080 to visualize the deployment using the Visualizer.

Figure 2 shows a screenshot of the VuShell display for visualization taken from a previous stack deployment:

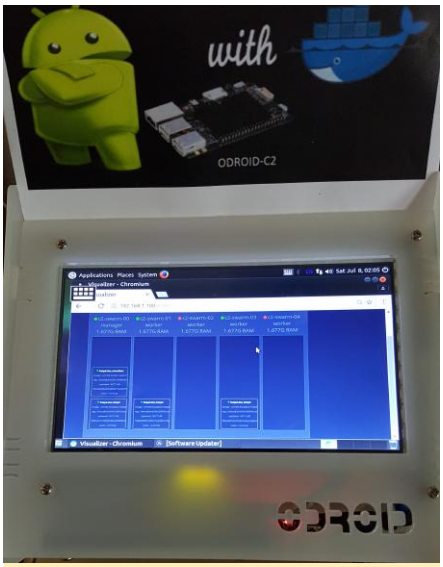


Figure 2 – VuShell Visualizer

To undeploy the stack, issue the following command:

```
$ docker stack rm httpd-dsv
```

Migrating my WordPress blog to the swarm

To illustrate a more realistic stack deployment, I decided that a good test is to migrate my blog to the swarm. This is useful to me as it enables me to bring up my blog easily to another environment when disaster strikes. To do this, I have to do some preparation work:

- Create a dump of the WordPress database using mysqldump to create: mysql.dmp.
- Use a text editor to replace all references of my domain name (mrdreambot.ddns.net) in the .dmp file with the swarm manager's IP address which is 192.168.1.100.
- Tar up /var/www/html directory which contains scripts and uploaded assets
- Pick the docker images to use: mrdreambot/arm64-mysql and arm64v8/wordpress.
- Armed with the above, I can proceed to create a docker stack deployment for my WordPress blog.

State persistence using bind-mount volumes

The first approach I took was to use host directories as data volumes (also called bind-mount volumes) for data persistence. The yaml file is shown below:

```
version: '3'
services:
  db:
    image: mrdreambot/arm64-mysql
    volumes:
      -
/nfs/common/services/wordpress/db_data:/u01/
my3306/data
      -
/nfs/common/services/wordpress/db_root:/root
    environment:
      MYSQL_ROOT_PASSWORD: Password456
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpressuser
      MYSQL_PASSWORD: Password456
    deploy:
      restart_policy:
        condition: on-failure
      placement:
        constraints: [node.role == manager]
  wordpress:
    depends_on:
      - db
    image: arm64v8/wordpress
    volumes:
      -
/nfs/common/services/wordpress/www_src/html:
/usr/src/wordpress
      -
/nfs/common/services/wordpress/www_data:/var
/www/html
    ports:
      - 80:80
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER: wordpressuser
      WORDPRESS_DB_PASSWORD: Password456
      WORDPRESS_DB_NAME: wordpress
    deploy:
      replicas: 3
      restart_policy:
        condition: on-failure
            placement:
        constraints: [node.role == manager]
```

Figures 3 and 4 show the screenshots for the stack deployment.

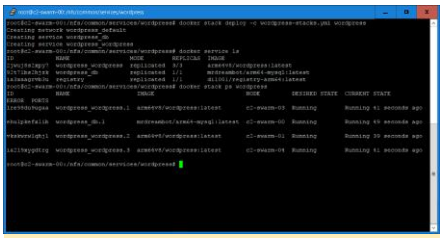


Figure 3 – WordPress bind mount volume deployment

Figure 4 – WordPress running

You have probably noticed that the WordPress site has lost some of its customized look as the arm64v8/wordpress docker image does not provide any PHP customization or libraries.

As mentioned earlier, if you do not define Networks in your yaml file, docker creates a 'wordpress_default' overlay network for the deployment automatically. The overlay network is required such that WordPress can reference the MySQL database using its name "db" as defined in the yaml file:

```
WORDPRESS_DB_HOST: db: 3306
```

The data volumes warrant some explanation. First thing to note is that all the host directories used as data volumes are NFS mounted and accessible to all swarm nodes.

/nfs/common/services/wordpress/db_data:/u01/my3306/data

The host directory /nfs/common/services/wordpress/db_data is an empty directory. It is mapped to the container's /u01/my3306/data directory where the MySQL database is located. How its content is created will be described next.

/nfs/common/services/wordpress/db_root:/root

I pre-populated the host directory /nfs/common/services/wordpress/db_root with 2 files:

- run.sh – the MySQL startup script which replaces the one located in the container's /root directory. This script is the entry point to the MySQL container. I changed the script to look for the mysql.dmp file located also in /root. If it is there, import the dump file into MySQL which will populate the /u01/my3306/data directory with data. If there is no mysql.dmp file, it will do nothing in additional to the usual processing.
- mysql.dmp – the dump file of my Blog's MySQL database

The changes in the run.sh file compared to the one that comes with the MySQL docker image are shown below:

```
...
DMP_FILE=/root/mysql.dmp
...
if [ "$MYSQL_DATABASE" ]; then
   mysql -uroot -e "CREATE DATABASE IF NOT
EXISTS `$MYSQL_DATABASE`"
   if [ -f "$DMP_FILE" ]; then
      mysql -uroot $MYSQL_DATABASE <
$DMP_FILE
   fi
fi
...
```

Note that this is required only when you run the container for the first time. Subsequent deployment will not require this volume mapping as the database will have been set up during the first run. This means that you can comment out this line in the yaml file after successfully deploying this stack once:

```
# -
/nfs/common/services/wordpress/db_root:/root
```

/nfs/common/services/wordpress/www_src/html:/usr/src/wordpress

arm64v8/wordpress initializes WordPress by copying the contents in its /usr/src/wordpress directory to its /var/www/html directory on startup if /var/www/html has no content. By pre-populating the host directory /nfs/common/services/wordpress/www_src/html with the content from the tar file created earlier, arm64v8/wordpress will initialize WordPress with my Blog's content. This is required only when you run the container for the first time. This means that you can comment out this line in the yaml file after successfully deploying this stack once:

```
# -
/nfs/common/services/wordpress/www_src/html:
/usr/src/wordpress
```

/nfs/common/services/wordpress/www_data:/var/www/html

The host directory /nfs/common/services/wordpress/www_data is an empty directory whose content will be initialized by the arm64v8/wordpress script as described above.

Why not use docker-compose?

You may be wondering why I did not use docker-compose to run the yaml file, for example, using once-off commands as the docker documentation suggests? The reason for it is that the docker-compose I installed using apt-get is version 1.8.0 which does not understand docker-compose yaml file version 3 which is required for "docker

stack deploy"! I tried to build the latest version of docker-compose from source without success. This is the reason I am not using docker-compose.

State Persistence Using Shared-storage Volumes

Using bind-mount volumes is host-dependent. Use of shared volumes has the benefit of being host-independent. A shared volume can be made available on any host that a container is started on as long as it has access to the shared storage backend, and has the proper volume plugin (driver) installed that allow you to use different storage backends such as: Amazon EC2, GCE, Isilon, ScaleIO, Glusterfs, just to name a few. There are lots of volume plugins or drivers available such as Flocker, Rex-Ray, etc. Unfortunately, no binaries for those plugins are available for ARM64 machines such as ODROID-C2. Fortunately, the inbuilt 'local' driver supports NFS. And it is the driver I am using for shared volume deployment.
The yaml file for this is shown below:

```
version: '3'
services:
  db:
    image: mrdreambot/arm64-mysql
    volumes:
      - db_data:/u01/my3306/data
#       -
/nfs/common/services/wordpress/db_root:/root
    environment:
      MYSQL_ROOT_PASSWORD: Password456
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpressuser
      MYSQL_PASSWORD: Password456
    deploy:
      placement:
        constraints: [node.role == manager]
      replicas: 1
      restart_policy:
        condition: on-failure
  wordpress:
    depends_on:
      - db
    image: arm64v8/wordpress
    volumes:
#       -
/nfs/common/services/wordpress/www_src/html:
/usr/src/wordpress
      - www_html:/var/www/html
    ports:
      - "80:80"
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER: wordpressuser
      WORDPRESS_DB_PASSWORD: Password456
      WORDPRESS_DB_NAME: wordpress
    deploy:
#       placement:
#         constraints: [node.role ==
manager]
      replicas: 3
      restart_policy:
        condition: on-failure
volumes:
    db_data:
      external:
        name: db_data
    www_html:
      external:
        name: www_html
```

Again, the volumes warrant some explanation:

/nfs/common/services/wordpress/db_root:/root

It serves the same purpose as in the bind-mount volume section. It is needed only when you run the stack for the first time to initialize the MySQL database.

/nfs/common/services/wordpress/www_src/html:/usr/src/wordpress

It serves the same purpose as in the bind-mount volume section. It is needed only when you run the stack for the first time to initialize the WordPress content.

db_data:/u01/my3306/data

db_data is a shared volume created outside of the stack deployment meaning it is created before the yaml file is deployed. It is used to store the MySQL database content and is uninitialized on creation.

www_html:/var/www/html

www_html is a shared volume created outside of the stack deployment meaning it is created before the yaml file is deployed. It is used to store the WordPress content and is uninitialized on creation.

Creating the shared volumes

You have probably noticed the section in the yaml file that reads:

```
volumes:
    db_data:
      external:
        name: db_data
    www_html:
      external:
        name: www_html
```

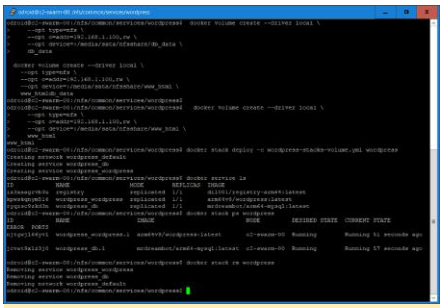
The db_data and www_html shared volumes are created using the following commands:

```
docker volume create --driver local
    --opt type=nfs
    --opt o=addr=192.168.1.100,rw
    --opt
device=:/media/sata/nfsshare/www_html
    www_html
docker volume create --driver local
    --opt type=nfs
    --opt o=addr=192.168.1.100,rw
    --opt
device=:/media/sata/nfsshare/db_data
    db_data
```

The directories /media/sata/nfsshare/db_data and /media/sata/nfsshare/www_htm must exist before you create the volumes. My /etc/exports file has an entry:

```
/media/sata/nfsshare
192.168.1.0/255.255.255.0(rw,sync,no_root_sq
uash,no_subtree_check,fsid=0)
```

To prove that the shared volumes work, I initially deployed only 1 mySQL and 1 WordPress replica on the Docker manager and let them initialize the shared volumes.



WordPress shared volume deployment

Then I commented out the 2 lines for WordPress placement:

```
#       placement:
#         constraints: [node.role ==
manager]
```

and the 2 bind-mount volumes:

```
#       -
/nfs/common/services/wordpress/db_root:/root
#       -
/nfs/common/services/wordpress/www_src/html:
/usr/src/wordpress
```

Next, I want to deploy 3 replicas of WordPress on multiple nodes. Since we are using the "local" driver, we have to create the volumes on each node. As shown in Figure 5, I used "parallel ssh" to create them on all nodes using just 2 commands. Figure 5 shows the volume and the stack deployment:

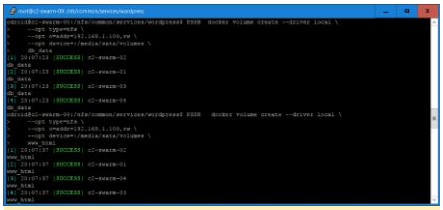


Figure 5 - Creating the volumes on nodes

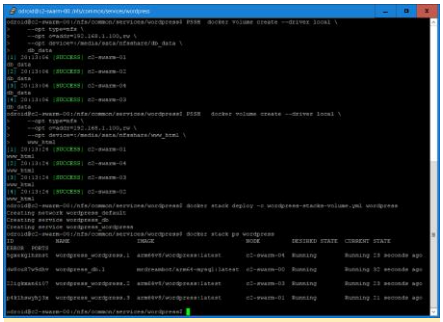


Figure 6 - WordPress shared volume deployment

I checked that all replicas are using the shared volumes by using "docker exec -it" to get into the WordPress containers on the nodes they were running on and examining the content in the /var/www/html directory to verify that everything was working.
Under the covers, both approaches use NFS for sharing among the nodes. However, shared volumes provide a higher-level host-independent abstraction than bind-mount volumes. Potentially, you can recreate the shared volumes using storage backends other than NFS such as AWS EC2 and Glusterfs. Bind-mount, on the other hand, is tied to your host file system which may be difficult to migrate to another environment.

Conclusion

I learned something new exploring the use of "docker stack deploy". I hope you'll find this article useful and informative. There are still many features such as rolling updates, Continuous Integration/Continuous Deployment (CI/CD), blue/green and A/B deployments, just to name a few, that I am yet to explore using my ODROID-C2 Swarm cluster. And there are other service orchestration frameworks such as Kubernetes and Openshift that are more prevalent in the Enterprise environment than Docker Swarm Mode. I shall explore additional Docker Swarm Mode use cases and Swarm Mode alternatives and report my findings in the future when the opportunity arises.

Linux Gaming: Mobile Entertainment System

October 1, 2017 By Tobias Schaaf Gaming, Linux

Hardkernel has done a great job with releasing new hardware recently. I saw an opportunity to create my own mobile entertainment system using a few components available through Hardkernel. This project is rather easy and well-suited for beginners, even children.

What you will need

This project is based on the VuShell and components that can be fit inside the case. In fact, there's quite a bit of space in this case, allowing for a variety of different layouts. For now, I'll focus on the layout I'm using, but if you want try this project you can exchange or add components as you see fit.

ODROID-VuShell (**http://bit.ly/2b8lk6a**)
As the case for our project, this is an absolute must-have!

ODROID-VU7 Plus (**http://bit.ly/2cmKyuN**)
You could also use the ODROID-VU7 instead (**http://bit.ly/1NWxgDx**) if you want to save a few dollars or use a screen with slightly less power consumption.

ODROID-C1+ (**http://bit.ly/1Upx5yI**)
You can also use an ODROID-C2 or XU4. Unfortunately, the C1 and XU3 won't work, as they don't have the necessary I2S connectors.

The C1+ is probably your best choice, since it uses very little power and allows you to use a battery pack. The board powers the VU7 over the USB 2.0 OTG connector, so only one power plug is needed.

Stereo Boom Bonnet (**http://bit.ly/2wbKkyE**)
As we want to have sound in our project, to be truly mobile, this is a must-have.

5V/2A PSU
If you use an ODROID XU4, you'll need an additional 5V/4A PSU.

SD card with 8MB or more storage
You could also use an eMMC module, but once assembled you will no longer be able to reach the eMMC module, making corrections impossible without disassembling everything. The SD card, on the other hand, will still be accessible with tweezers.

Spacers
I got my spacers from other ODROID products I had laying around, but they can also be bought cheaply on Amazon (**http://amzn.to/2yj4OG8**).

Keyboard, Mouse
After the initial setup, these may no longer be needed.

Following the list above, your costs should come to around $160 (not including your keyboard and mouse, or shipping).

Figure 1 – The main components for the project laid out together ready for assembling

There are a couple of other components you might want to get, but these are completely up to you:

Gamepad (for a better gaming experience)
I suggest a wireless XBox 360 controller with a Wireless PC Adapter, since one adapter supports up to four controllers, meaning you won't have to deal with any cables.

External storage (for storing large amounts of data)
For example, you may want to use a USB thumb drive or external HDD to store movies or games. If you use a large SD card (32GB or bigger) you don't necessarily need one, but they're probably easier to exchange than a SD card if you find you need extra storage.

WLAN Module
If you want to connect to a wireless network, you will need one of these.

UPS3 or any other *Battery Pack*
A power bank for your cellphone or tablet will also do. This way, you can make the system entirely mobile so that you don't need to have a power plug nearby. A decent power bank should give you somewhere between 3-5 hours runtime for the entire system.

Micro USB-DC Power Bridge Board (**http://bit.ly/2wbWQ1e**)
If you use an ODROID-XU4, this will make sure the power for the display is constant.

IR Remote Controller (***http://bit.ly/1M6UGiR***) *or any other IR Remote*
The C1+ and C2 come with a IR receiver. If you want to use it in Kodi, that's something you can do as well.

Solder Set
This is recommended for advanced users wanting "real" stereo sound

Software
Before you start to assemble the components, you should setup your ODROID, install the operating system (I used my own image ODROID GameStation Turbo for the ODROID-C1 Series), prepare the boot.ini, and, if you want to, put games, movies, and whatever on your board. It's better to do this up front, as it may be difficult to do at a later point if you don't have a network connection.

Make sure to set the options for the VU7 or VU7 Plus (depending on your choice of LCD screen) on your boot.ini:

```
$ setenv m "1024x600p60hz" # 1024x600

$ # HDMI DVI Mode Configuration
$ # setenv vout_mode "hdmi"
$ setenv vout_mode "dvi"
$ # setenv vout_mode "vga"
```

You can also configure the system to load the modules required for the Stereo Boom Bonnet.

Open a terminal and type the following commands:

```
$ su -
$ echo "snd-soc-pcm5102 snd-soc-odroid-dac"
>> /etc/modules
```

After that, you can copy over the games or movies you want to use, and configure EmulationStation, Kodi, and any other additions to your liking, or you can do this later once the system is assembled.
You will definitely need the boot.ini configuration at absolute minimum, or else you won't see anything on your screen later.

Assembly
Assembling is rather easy, just follow the step from Hardkernel on how to assemble the VuShell (http://bit.ly/2b8lk6a) with some slight modifications.

Figure 2 – Attaching the ODROID-C1+ on the back of the Vu7 Plus

Once you attach the screen to the front and add the first side on the board (Step 7) it's time for some modifications. First, connect the Stereo Boom Bonnet with the board. To do this, gently bend the parts that hold the speakers until they come apart and you have the board and the speaker separated. Unplug the cable for the speakers. It's best to connect the cable of the Stereo Boom Bonnet before you assemble it. Refer to the guide from Hardkernel to make sure you put the cable on the the correct way (http://bit.ly/2xuWVjA).

Remove the screw that was added in Step 3 of the VuShell assembly on the side of the VU7 Plus and replace it with a couple of spacers. Place the Stereo Boom Bonnet upside down on top of the spacers. Use the screw you originally removed to fasten the Stereo Boom Bonnet. Use (4) M3 20mm spacers to lift up the Stereo Boom Bonnet so the volume slider aligns with one of the holes of the VuShell, which will later allow you to regulate the volume.

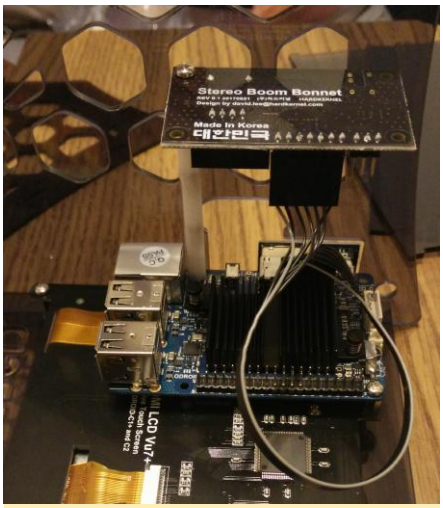


Figure 3 – Stereo Boom Bonnet connected upside down with spacers over the ODROID-C1+

After you connect the first side, you can do the same to the other side. Please note that the top hole of the C1+ is normally not connected to the case, as can be seen in Step 5 of assembling the VuShell. If you put a spacer in here, don't worry if they are not screwed into a socket. It will work fine without it. Once the second line of spacers is assembled to the C1+ and the Stereo Boom Bonnet, you can connect the first speaker that came with the Stereo Boom Bonnet.

Align the speaker to one of the holes in the VuShell case. I used transparent sticky tape to fasten the speaker to the case for my first test. Later, you can super-glue it to the case. Technically, one speaker is enough to have some rather good sound, but if you choose, you can connect the second speaker to one of the other holes on the same side.

If you want real stereo sound, you'll need to lengthen the cable on the second speaker so it can be connected to the other side of the VuShell. Please note that some soldering is required, so although it's rather easy, it should be done with care, and children should be supervised by an adult.

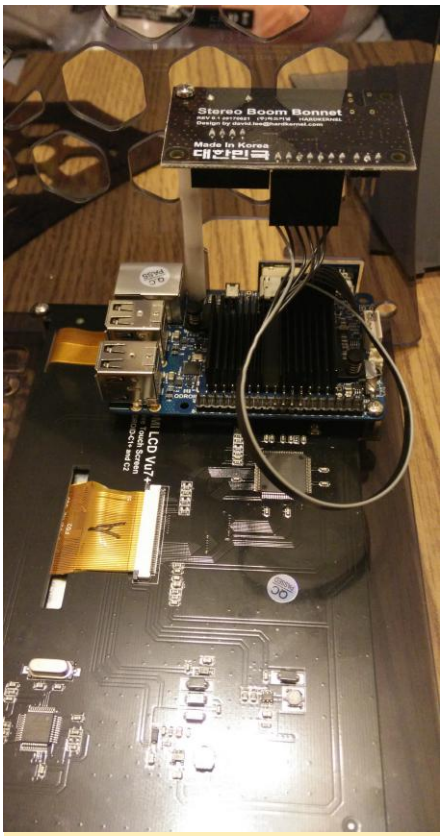
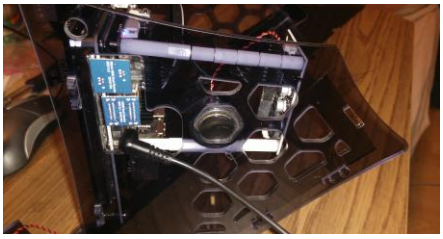


Figure 4 and 5 – I ran out of 20mm spacers and switched to 10mm. It doesn't look pretty. Please use 20mm instead. Don't be as lazy as me.



Even with just one speaker attached, the sound should be good enough to watch movies or play games. I made a video where I tested video playback with ffmpeg http://bit.ly/2xox1wb.

In this video, I turn the volume up and down using the slider that is easy to reach thanks to the spacers.

After that I also tried some good old 8-bit sounds by starting Cave Story from within EmulationStation (http://bit.ly/2xlDxGo). This also worked perfectly. Only having one speaker connected was really no big deal.

Advanced Assembly
As you may have noticed, the cable of the second speaker is too short to reach the other end of the VuShell. Therefore, I needed to lengthen the cable to be able to reach the other side of the case. This process is fairly easy and can probably even be done by children, but only with adult supervision.

You will need some basic soldering equipment. Mainly just extra wire, soldering tin, and some heat shrink wire wrap (http://amzn.to/2wH9edl) if you have it. Unfortunately, I didn't have these. It works without it, but it's better to have the heat shrink wrap in order to protect the cables once they're soldered. You will also need something to cut the wire. A wire/cable cutter will do nicely, and since the cables are rather thin, a pair of scissors or even a knife would probably do as well.

Figure 6 – Soldering equipment and a second speaker

When you have all of the items, you can start by unwinding the cables close to the speaker to a length of about 5 cm (2 inches). Then cut the wire with the wire cutter and expose the blank wires.

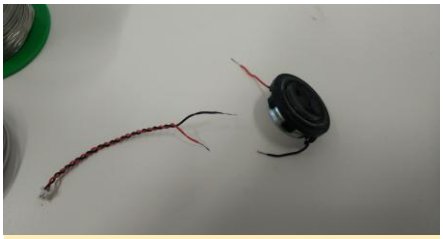


Figure 7 – Don't cut the wires too close to the speakers, in case you have to start over again. Twist the exposed wire-ends together.

Cut two longer wires about 20-25 cm (8-10 inches). I strongly suggest using different colors for the wires so you see which cable needs to connect to which other cable. Make sure the two cables you cut are nearly the same length. I also suggest using similar thin cables as the speaker cables. Mine were just slightly thicker and they fit perfectly.

After cutting the wires, expose the ends by slowly removing the cover of the cable. Be careful not to cut the cable in the process. Once that is done, twist the exposed wires so they hold together. Then, you can apply tin to the exposed ends cover all exposed ends in a thin layer of tin. This would also be a good time to apply the heat shrink wrap to the extension cords (two for each cable). After that, you can solder the cable ends together. Make sure to connect the right cables.

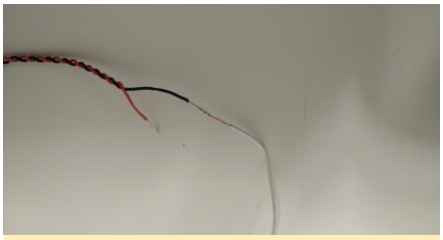
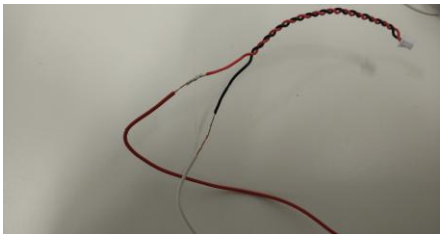


Figure 8 and 9 – Combine the ends of the cable to the extension cords, one side after another.



After you solder one end of the cable, you can connect the other end to the speaker.

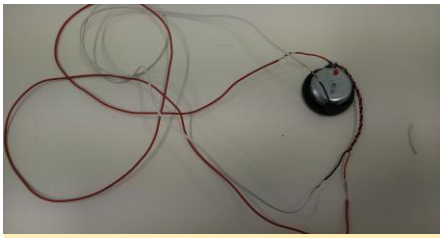


Figure 10 – Both ends are connected and the speaker now has a nice long cable to work with.

In the end, I twisted the cable like the original cable was twisted, so it's easier to handle. This actually took a little while, but the result was good and allowed for much easier assembly in the VuShell. However, make sure that you don't stress the solder points too much when you twist the cable, or they may come apart again.

Now would also be a good time to put the heat shrink wraps over the exposed cable ends and heat them up so they seal the exposed wires. I tried to do the same with electrical tape but the cables were too thin to wrap it around properly. Once you're done twisting the cable, it should look like a longer version of of the original cable, just with some soldering points.

Figure 11 – Make sure cables match up at the end

Now it's time to put the unit together and place the new speaker inside the VuShell. When you assemble the speakers, the speaker connector on the top is for the left channel and the speaker connector on the bottom is for the right channel. You can also use some YouTube videos to test if the left and right speaker are connected in the right order. You can fasten the left speaker either with super glue or sticky tape.

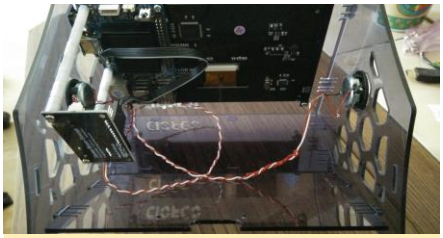


Figure 12 – Speakers are assembled and there is still plenty of room in the case

After that, I turned on the device and tested to see if both speakers worked right at the start (http://bit.ly/2xuF4ct). Because there is plenty of room inside the case, you can add additional components rather easily. As already shown in the assembly instructions from Hardkernel, there are already screw holes to place an HDD or power bank inside, which would make the device entirely mobile.

Figure 13 – This 12500 mA battery should give you 3-5 hours of mobile entertainment for gaming, watching movies, or listening to music

You can also easily place an XBox 360 wireless PC adapter in there together with the power bank. That way, you can use up to 4 XBox 360 controllers at the same time without having to add a new cable. This is awesome for controlling Kodi or EmulationStation without the need for a keyboard.

Conclusion

This was a fun and easy project. Some people are already enjoying this little console, stating that they are amazed by the idea and mobility you have thanks to the power bank. Since the VuShell has a lot of space, this project can have many different variations depending which extra accessories you want to put to use. You might even want to skip the speakers entirely and instead simply use the Stereo Boom Bonnet's headphone jack, which would allow you to play your games on a train ride, or if you're stuck on an airplane for several hours.

Although not the biggest screen, it's good enough to have a couple of friends sitting next to you to watch some movies on a field trip, or play some friendly or competitive games on one of the many emulators. Some will prefer the extra power of an XU4 to seriously play some games for the PSP, Dreamcast, or N64, while others are fine with some Nintendo, Super Nintendo, SEGA Genesis, or other classics on a C1. Placing it in the kitchen running Android on an C2 allows you to listen to your favorite music while cooking. Thanks to the touch screen everything you need is at the tip of your finger.

All in all, the options are nearly limitless, and it's very easy to do. Even children can build their own console. I encourage you to give it a try and comment on what you can do with such an all-in-one system.

How to Install ArchLinux With Full Disk Encryption on an ODROID-C2

October 1, 2017 By @YesDay Linux, ODROID-C2, Tutorial

Full Disk Encryption (FDE) protects our data against unauthorised access in case someone gains physical access to the storage media. In this article, I will describe how to install ArchLinux with Full Disk Encryption on ODROID-C2. The encryption method is LUKS with XTS key-size 512 bit (AES-256).

In a nutshell, Full Disk Encryption requires the following:

- Encrypting a partition and copying the root filesystem to it.
- The kernel to include the dm_crypt kernel module. In our case, this is already included by default, therefore we won't need to re-compile the kernel.
- The initramfs to include the dm_crypt kernel module and the cryptsetup binary. We use a tool called dracut to generate the required initramfs. Dracut supports the required functionality via the additional modules crypt and lvm.
- Passing the **dracut options for LUKS** to the initramfs via the bootargs property inside boot.ini. For example, say that in our case, we want the initramfs to unlock a LUKS volume with UUID ae51db2d-0890-4b1b-abc5-8c10f01da353 and load the root filesystem from the device mapper /dev/mapper/vg-root. To pass these dracut options we configure the following:

```
sudo nano /boot/boot.ini
setenv bootargs "rd.luks.uuid=ae51db2d-0890-4b1b-abc5-8c10f01da353 root=/dev/mapper/vg-root rootwait < leave the rest as is >"
```

Note
A lot of the steps throughout this document involve editing configuration files. To keep the words to the minimum, we use the above notation as a very concise way to describe such file editing steps. The above notation means:

- You need to edit the file /boot/boot.ini with root privileges (hence sudo nano /boot/boot.ini). Nano is the command line editor, however feel free to use another editor of your choice.
- Find the line starting with setenv bootargs and add or edit the configuration options rd.luks.uuid=ae51db2d-0890-4b1b-abc5-8c10f01da353 root=/dev/mapper/vg-root rootwait. Some files mentioned throughout this document might have the corresponding line being commented out or not present at all. If that's the case you will need to uncomment or append the line into the file, respectively.
- Leave the rest of the line after rootwait as is.

Additionally, for a headless setup, you will need to enable remote unlocking via SSH as described in "Remotely unlock the LUKS rootfs during boot using Dropbear sshd" article at **http://bit.ly/2g6qjDv**. Last but not least, if you prefer to use the described functionality out of the box, simply download the OS image at **http://bit.ly/2xR8LDe**. Either way, the current document will provide more technical details in regards to the underlying components and how they work together in a Full Disk Encryption environment.

Hardware requirements

- ODROID-C2
- A Linux box from which you will flash the OS image and interact with the ODROID-C2
- USB disk with at least 4GB capacity
- A microSD card or eMMC module with at least 4GB capacity
- (Optional) A USB-UART module kit for connecting with the ODROID-C2's serial console. Refer to the post at **http://bit.ly/2fM29BB** for instructions on how to connect along with explanation why the serial console is highly recommended in this case.

Flash the OS image and boot ODROID-C2

Flash the OS image to the USB disk by following the instructions from **http://bit.ly/2fGKEik.** Replace /dev/mmcblk0 in the following instructions with the device name for the microSD card as it appears on your computer. If mounted, unmount the partitions of the microSD card:

```
$ lsblk
$ umount /dev/mmcblk0p1
$ umount /dev/mmcblk0p2
```

Zero the beginning of the microSD card:

```
$ sudo dd if=/dev/zero bs=1M count=8
of=/dev/mmcblk0
$ sync
```

Using a tool like GParted, create an MBR/msdos partition table and two partitions on the microSD card:

- ext4 partition with 128M size
- lvm2 partition occupying the rest of the space (no need to format yet)

Next, copy the contents of the /boot directory from the USB disk into the first partition of the microSD card:

```
$ sudo cp -R /media/user/usb-disk/boot/*
/media/user/micro-sd-card-part1/
```

Create a symbolic link as a workaround for the hardcoded boot.ini path of the alarm/uboot-odroid-c2 (http://bit.ly/2xbEdPo):

```
$ cd /media/user/micro-sd-card-part1
$ sudo ln -s . boot
```

Then, flash the bootloader files:

```
$ sudo ./sd_fusing.sh /dev/mmcblk0
```

Determine the UUID of the USB disk:

```
$ sudo lsblk -o name,uuid,mountpoint
NAME UUID MOUNTPOINT
sdb
└─sdb1 2b53696c-2e8e-4e61-a164-1a7463fd3785
/media/user/usb-disk
```

Note that If there are duplicate UUIDs among the partitions of the USB disk and the microSD card, then remove the duplicates to avoid future conflicts:

```
$ sudo tune2fs /dev/sda2 -U $(uuidgen)
```

Configure the boot.ini to boot from the USB disk. To do so, use the UUID from the previous step to configure the boot.ini of the microSD card:

```
$ sudo nano /media/user/micro-sd-card-
part1/boot.ini
$ setenv bootargs "root=UUID=2b53696c-2e8e-
4e61-a164-1a7463fd3785 rootwait "
```

Unmount, run sync few times, and remove the microSD card and the USB disk from the Linux box. Plug the microSD card and the USB disk to the ODROID-C2, then boot the ODROID-C2 and connect to its serial console. If you need instructions on how to connect to the serial console, please refer to the article at http://bit.ly/2fM29BB.

If all goes, well you should boot into the USB disk. Note that if root=UUID=2b53696c-2e8e-4e61-a164-1a7463fd3785 doesn't work, then try root=/dev/sda1, root=/dev/sdb1 or whatever device name you see in the console prior to the failed boot (e.g,. [14.812393] sd 1:0:0:0: [sda] Attached SCSI removable disk). If you are still having issues try restarting a few times and/or repositioning the USB disk into a different USB port on the ODROID-C2. Don't worry if it seems to be giving you trouble, as you won't have to boot to the USB disk again after the first successful boot.

Next, verify that the root filesystem is mounted from the USB disk:

```
$ df -h
```

Change passwords

Change the passwords for the alarm and the root user. The default credentials are alarm/alarm and root/root.

```
$ passwd
$ su
$ passwd
```

Install required packages

```
$ su
$ pacman -Syu
$ pacman -S --needed sudo python git rsync
lvm2 cryptsetup
```

(Optional) Setup passwordless sudo for the user alarm:

```
$ echo 'alarm ALL=(ALL) NOPASSWD: ALL' >
/etc/sudoers.d/010_alarm-nopasswd
```

Install dracut
Install pacaur (http://bit.ly/2yEjAaY):

```
$ sudo pacman -S --needed base-devel cower
$ mkdir -p ~/.cache/pacaur && cd "$_"
$ cower -d pacaur
$ cd pacaur
$ makepkg -si --noconfirm --needed
```

Install dracut using the pacaur tool:

```
$ pacaur -S dracut
```

Verify the dracut installation by listing modules

```
$ dracut --list-modules
```

If the "pacaur -S dracut" command reports an error that aarch64 architecture is not supported by the package, then follow these steps to configure support for aarch64:

```
$ cd ~/.cache/pacaur/dracut/
$ nano PKGBUILD # replace `arch=("i686"
"x86_64")` with `arch=("aarch64")`
$ makepkg -si --noconfirm --needed
```

If the makepkg reports an error like dracut-046.tar ... FAILED (unknown public key 340F12141EA0994D), then type these commands and try again:

```
$ gpg --full-gen-key
$ gpg --recv-key 340F12141EA0994D
```

Refer to Makepkg signature checking for more details at http://bit.ly/2wuuBe6.

If the "gpg –full-gen-key" command reports the error Key generation failed: No pinentry, then follow the below steps to configure gpg as described at http://bit.ly/2yDAJBy and try again. The gpg-agent needs to know how to ask the user for the password:

```
$ nano ~/.gnupg/gpg-agent.conf
$ pinentry-program /usr/bin/pinentry-curses
$ gpg-connect-agent reloadagent /bye
```

If makepkg reports missing dependencies error, then upgrade the packages and try again.

```
$ sudo pacman -Syu
$ pacaur -Syua
```

Prepare the LUKS rootfs
Encrypt the second partition of the microSD card (see also Recommended options for LUKS at http://bit.ly/2yF15D2):

```
$ sudo cryptsetup -v -y -c aes-xts-plain64 -
s 512 -h sha512 -i 5000 --use-random
luksFormat /dev/mmcblk0p2
```

-v = verbose
-y = verify passphrase, ask twice, and complain if they don't match
-c = specify the cipher used
-s = specify the key size used
-h = specify the hash used
-i = number of milliseconds to spend passphrase processing (if using anything more than sha1, must be great than 1000)
–use-random = which random number generator to use
luksFormat = to initialize the partition and set a passphrase
/dev/mmcblk0p2 = the partition to encrypt

Unlock the LUKS device and mount it at /dev/mapper/lvm:

```
$ sudo cryptsetup luksOpen /dev/mmcblk0p2
lvm
```

Create primary volume, volume group, and logical volume:

```
$ sudo pvcreate /dev/mapper/lvm
$ sudo vgcreate vg /dev/mapper/lvm
$ sudo lvcreate -l 100%FREE -n root vg
```

Create the filesystem:

```
$ sudo mkfs.ext4 -O ^metadata_csum,^64bit
/dev/mapper/vg-root
```

Mount the new encrypted root volume (logical volume):

```
$ sudo mount /dev/mapper/vg-root /mnt
```

Copy the existing root volume to the new, encrypted root volume. With a 1.5GB installation, it completes in about 6 minutes on an average microSD:

```
$ sudo rsync -av
--exclude=/boot
--exclude=/mnt
--exclude=/proc
--exclude=/dev
--exclude=/sys
--exclude=/tmp
--exclude=/run
--exclude=/media
--exclude=/var/log
--exclude=/var/cache/pacman/pkg
--exclude=/usr/src/linux-headers*
--exclude=/home/*/.gvfs
```

```
--exclude=/home/*/.local/share/Trash
/ /mnt
```

If the SSH host keys are empty, remove them so that they will be regenerated the next time the sshd starts. This will prevent the memory leak issue as described at http://bit.ly/2xQxGqe.

```
$ sudo rm /mnt/etc/ssh/ssh_host*key*
```

Create some directories and mount the boot partition:

```
$ sudo mkdir -p /mnt/boot /mnt/mnt /mnt/proc
/mnt/dev /mnt/sys /mnt/tmp
$ sudo mount -t ext4 /dev/mmcblk0p1
/mnt/boot
```

Register the encrypted volume in crypttab

```
$ sudo bash -c 'echo lvm UUID=$(cryptsetup
luksUUID /dev/mmcblk0p2) none luks>>
/mnt/etc/crypttab'
```

Configure fstab:

```
$ sudo nano /mnt/etc/fstab
$ /dev/mapper/vg-root / ext4 errors=remount-
ro,noatime,discard 0 1
$ /dev/mmcblk0p1 /boot ext4 noatime,discard
0 2
Next, generate a new initramfs using dracut.
The following commands will add the dracut
modules crypt and lvm to the initramfs.
These modules will prompt for LUKS password
during boot and unlock the LUKS volume. Note
that the order of the modules is important:

$ sudo dracut --force --hostonly -a "crypt
lvm" /mnt/boot/initramfs-linux.img
```

Next, determine the LUKS UUID:

```
$ sudo cryptsetup luksUUID /dev/mmcblk0p2
470cc9eb-f36b-40a2-98d8-7fce3285bb89
```

Configure the rd.luks.uuid and root dracut options in bootargs. These will unlock the LUKS volume and load the rootfs from it during boot:

```
$ sudo nano /mnt/boot/boot.ini
$ setenv bootargs "rd.luks.uuid=470cc9eb-
f36b-40a2-98d8-7fce3285bb89
root=/dev/mapper/vg-root rootwait "
```

Note that in the above step, do NOT delete the rest of bootargs, essentially replace root=UUID=2b53696c-2e8e-4e61-a164-1a7463fd3785 with rd.luks.uuid=470cc9eb-f36b-40a2-98d8-7fce3285bb89 root=/dev/mapper/vg-root and leave the rest of bootargs untouched. Then, unmount and reboot into the LUKS rootfs:

```
$ sudo umount /mnt/boot
$ sudo umount /mnt
$ sudo reboot
```

If all goes well you will be prompted to enter the LUKS password during boot. Next, verify the LUKS rootfs:

```
df -h
output
Filesystem Size Used Avail Use% Mounted on
devtmpfs 714M 0 714M 0% /dev
tmpfs 859M 0 859M 0% /dev/shm
tmpfs 859M 8.3M 851M 1% /run
tmpfs 859M 0 859M 0% /sys/fs/cgroup
/dev/mapper/vg-root 1.7G 1.4G 256M 85% /
tmpfs 859M 0 859M 0% /tmp
/dev/mmcblk0p1 120M 26M 86M 23% /boot
tmpfs 172M 0 172M 0% /run/user/1000
```

Next, remotely unlock the LUKS rootfs during boot using Dropbear sshd. Replace 10.0.0.100 in the following instructions with the IP address assigned to the ODROID-C2 by your local DHCP server. Use the fing tool to find the assigned IP address (e.g. sudo fing 10.0.0.1/24). Then, make sure the SSH daemon is running:

```
$ sudo systemctl status sshd
$ journalctl -u sshd -n 100
```

If the above commands report that sshd fails with memory allocation error, then enter the following commands:

```
$ sudo rm /etc/ssh/ssh_host*key*
$ sudo systemctl start sshd
```

Refer to the article at http://bit.ly/2xQxGqe for more information about memory leaks in sshd.

Install and configure Dropbear
Install the dracut module crypt-ssh:

```
$ pacaur -S dracut-crypt-ssh-git
```

From your Linux box, copy the public SSH key to the appconf/dracut-crypt-ssh/authorized_keys file on the remote ODROID-C2 server:

```
$ cat ~/.ssh/*.pub | ssh alarm@10.0.0.100
'umask 077; mkdir -p appconf/dracut-crypt-
ssh; touch appconf/dracut-crypt-
ssh/authorized_keys; cat >>appconf/dracut-
crypt-ssh/authorized_keys'
```

Next, configure the crypt-ssh module:

```
$ sudo nano /etc/dracut.conf.d/crypt-
ssh.conf
$ dropbear_acl="/home/alarm/appconf/dracut-
crypt-ssh/authorized_keys"
```

Generate a new initramfs using dracut. The following commands will add the dracut modules network and crypt-ssh to the initramfs. Note that the order of the modules is important:

```
$ sudo dracut --force --hostonly -a "network
crypt lvm crypt-ssh" /boot/initramfs-
linux.img
```

Enable network access during boot by adding rd.neednet and ip dracut options to bootargs:

```
$ sudo nano /boot/boot.ini
setenv bootargs "rd.neednet=1
ip=10.0.0.100::10.0.0.1:255.255.255.0:archli
nux-luks-host:eth0:off
rd.luks.uuid=ae51db2d-0890-4b1b-abc5-
8c10f01da353 root=/dev/mapper/vg-root
rootwait "
```

If you prefer DHCP instead of static ip, simply replace with ip=dhcp. Refer to network documentation of dracut at http://bit.ly/2g6XCXk and dracut options at http://bit.ly/2yUBFT6 for more options (man dracut.cmdline). Reboot so that Dropbear starts, allowing for remote unlocking:

```
$ sudo reboot
```

From your Linux box, connect to the remote Dropbear SSH server running on the ODROID-C2:

```
$ ssh -p 222 root@10.0.0.100
```

Unlock the volume (asks you for the passphrase and sends it to console):

```
$ console_auth
Passphrase:
```

If unlocking the device succeeded, the initramfs will clean up itself and Dropbear terminates itself and your connection.

You can also type “console_peek” which prints what’s on the console. There is also the unlock command, but we encountered an issue while testing as described at http://bit.ly/2fHB2nw.

Some use cases require feeding input automatically to the interactive command console_auth. From your Linux box, unlock the volume:

```
$ ssh -p 222 root@10.0.0.100 console_auth <
password-file
```

or

```
$ gpg2 --decrypt password-file.gpg | ssh -p
222 root@10.0.0.100 console_auth
For additional security, you might want to
only allow the execution of the command
console_auth and nothing else. To achieve
this, you need to configure the SSH key with
restricting options in the authorized_keys
file. From your Linux box, copy the public
SSH key, with restricting options, to the
appconf/dracut-crypt-ssh/authorized_keys
file on the remote ODROID-C2 server:
$ (printf 'command="console_auth",no-agent-
forwarding,no-port-forwarding,no-pty,no-X11-
forwarding ' && cat ~/.ssh/*.pub) | ssh
alarm@10.0.0.100 'umask 077; mkdir -p
appconf/dracut-crypt-ssh; touch
appconf/dracut-crypt-ssh/authorized_keys;
cat >appconf/dracut-crypt-
ssh/authorized_keys'
```

Refer to the Dropbear documentation for a full list of restricting options. Prior to continuing, it might be a good idea to create a copy of the initramfs:

```
$ sudo cp /boot/initramfs-linux.img
/boot/initramfs-linux.img-`date +%y%m%d-
%H%M%S`
```

In a headless setup, carefully examine the restricting options to avoid locking yourself out.

Finally, generate a new initramfs using dracut:

```
$ sudo dracut --force --hostonly -a "network
crypt lvm crypt-ssh" /boot/initramfs-
linux.img
```

In this case, you can unlock the volume interactively by simply typing the following command:

```
$ ssh -p 222 root@10.0.0.100
```

Note that when typing the above command, the console_auth command is automatically invoked on the remote server and immediately prompts for password, as if you just typed ssh -p 222 root@10.0.0.100 console_auth. While you type the password, it will be displayed on the screen in plain text. Therefore, you should avoid unlocking interactively when the access is restricted to the console_auth command. When you press enter you will be disconnected no matter whether the password was correct or not. Whereas with the non-restricted login (see http://bit.ly/2hHAGl0), you would only be disconnected if the password was correct, meaning that you would have feedback for whether the unlocking was successful or not. On the other hand, to unlock the volume using a password file, from your Linux box, type the following command:

```
$ ssh -p 222 root@10.0.0.100 < password-file
```

or

```
$ gpg2 --decrypt password-file.gpg | ssh -p
222 root@10.0.0.100
```

For comments, questions, or suggestions, please visit the original blog post at **http://bit.ly/2xMQE3I**.

References

ArchLinux dm-crypt/Encrypting an entire system (**http://bit.ly/2xPaybR**)

How to install Debian with Full Disk Encryption on ODROID-C2 (**http://bit.ly/2g6JtcF**)

I2C LCD Module: Using the TWI 1602 16×2 Serial LCD

October 1, 2017 By Miltiadis Melissas ODROID-C2, Tinkering

After doing so many IoT projects with my ODROID-C2 like the seismograph detector (http://bit.ly/2uWqas0), the wine cellar preserver, and notifier (http://bit.ly/2wch3Vb), the Gmail mechanical notifier (http://bit.ly/2wch3Vb) and many others, I was thinking about adding a low energy, low cost LCD screen for depicting any valuable information of all those electronic constructions for the sake of portability and readability. The I2C TWI 1602 16×2 Serial LCD Module Display for Arduino JD is the ideal solution for materializing all those specifications and much more.

This LCD Module Display communicates with an ODROID-C2 using the I2C protocol with just 4 wires. The I2C protocol is a multi-master, multi-slave, packet switched, single-ended, serial computer bus invented by Philips Semiconductor (now NXP Semiconductors). It is typically used for attaching lower-speed peripheral ICs to processors and microcontrollers in short-distance, intra-board communication (http://bit.ly/2qGiYP4). In the following lines, we describe how this connection can be materialized physically and programmatically. The language used is Python 2.7, and the program can be implemented easily into other projects as a module with minor modifications.

Hardware

You will need all of the usual ODROID-C2 accessories:

- ODROID-C2
- MicroSD card with the latest Ubuntu 16.04 provided by HardKernel (http://bit.ly/2rDOCfn)
- WiringPi library for controlling the GPIOs of an ODROID-C2 running on Ubuntu 16.04 (instructions from Hardkernel on how to install the library can be found at http://bit.ly/1NsrlU9)
- Keyboard
- Screen
- HDMI cable
- The keyboard, the screen, and the HDMI cable are optional because you can alternatively access your ODROID-C2 from your desktop computer via SSH
- Micro USB power or, even better, a power supply provided by Hardkernel (http://bit.ly/1X0bgdt)
- Optional: Power bank with UBEC (3A max, 5V) if you want to operate the device autonomously (see Figure 1). Hardkernel provides a better solution with UPS3 specifically designed for ODROID-C2. You can purchase the UPS3 from their store at this link: http://bit.ly/2l1rE25. The UPS3 is a good choice, as it gives the detector the ability to operate autonomously with greater stability and duration.
- Ethernet cable or usb wifi dongle
- The C Tinkering Kit on Ubuntu, which can be purchased from Hardkernel (http://bit.ly/1NsrlU9)
- I2C TWI 1602 16×2 Serial LCD Module Display for Arduino JD, which can be found from various places, such as eBay

For the wiring, please follow the schematic in Figure 1. There are 2 important wires for the communication: the SDA that provides the I2C serial data, and the SCL that provides the I2C serial clock. The SDA is on Pin 3 on the I2C LCD Display and is connected on GPIO Pin 3 of ODROID-C2. The SCL is on Pin 4 and is connected on GPIO Pin 5 of the ODROID-C2. For visual reference see the schematic in Figure 1 and Hardkernel's excellent 40-pin layout for ODROID-C2 (http:// bit.ly/2aXAlmt). These will help to make sure the wiring is correct. Now that we have our hardware ready, let's see how we can establish a communication between the ODROID-C2 and the I2C Serial LCD Display using the I2C protocol. The GPIO Pin 2 provides the VCC power, +5V, for the LCD Display and GPIO Pin 39 is of course the ground, GND.

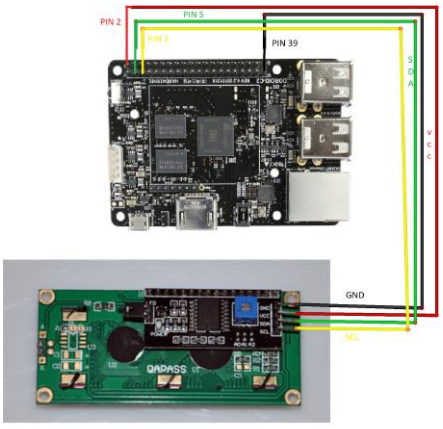


Figure 1 – wiring diagram

I2C communication

We will establish a connection between ODROID-C2 and the Serial LCD Display using the I2C protocol. The steps we

will follow here are almost identical with those presented on our previous article under the title "Seismograph Earthquake Detector: Measuring Seismic Acceleration using the ODROID-C2", published in ODROID magazine's July issue (http://bit.ly/2uWqas0). In that article, we described all the necessary steps necessary to establish communication between the ODROID-C2 and the MMA7455 accelerometer, which also uses I2C. We will repeat the same procedure here for the sake of the consistency and the integrity of this article.

All commands are entered in a terminal window or via SSH. First, you'll need to update ODROID-C2 to ensure all the latest packages are installed:

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ sudo apt-get dist-upgrade
```

Then you will need to reboot the ODROID-C2:

```
$ sudo reboot
```

You will need to install SMBus and I2C-Tools, since the LCD Module Display uses this protocol to communicate with the ODROID-C2. The System Management Bus, or SMBus, is a simple, single-ended, two-wire bus for lightweight communication. It is most commonly found in computer motherboards for communicating with the power source (http://bit.ly/2rAWhuU).

Once you have logged into your ODROID-C2 from the command line, run the following command to install Python-SMBus and I2C-Tools:

```
$ sudo apt-get install pythonsmbus
```

Set the ODROID-C2 to load the I2C driver:

```
$ modprobe ami-i2c
```

Set the ODROID-C2 to start I2C automatically at boot by editing /etc/modules:

```
$ sudo nano /etc/modules
```

Use your cursor keys to move to the last line, and add a new line with the following text:

```
$ i2c-dev
```

Press return, then add:

```
$ aml_i2c
```

Save your changes and exit the nano editor. To avoid having to run the I2C tools at root add the "ODROID" user to the I2C group:

```
$ sudo adduser Odroid 12c
```

Next reboot the ODROID-C2:

```
$ sudo reboot
```

Once your ODROID-C2 has been rebooted, you will have I2C support. You can check for connected I2C devices with the following command:

```
$ sudo i2cdetect -y -r 1
```

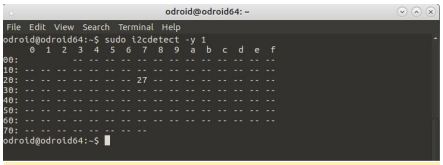


Figure 2 – Detected I2C devices using i2cdetect

If '27' is shown on line 20 under column 7, this means the LCD Display is communicating with the ODROID-C2 and working properly. More details may be found at http://bit.ly/2qCQM1s.

Python software

We will present the code in chunks, as we do always, in order to be better understood by our readers. The code is slightly modified from this the source here (http://bit.ly/2w2a957) and adopted for the needs of this project. The code is in Python and what it mainly does is to establish a connection between the ODROID-C2 and LCD Display by opening a I2C connection allowing 16 characters on two lines to be displayed. You can download the code here (http://bit.ly/2vzSMqd) and run it for immediate results, or if you don't want to retype all the code. First, import the necessary modules:

```
import smbus
import time

# Define device parameters
I2C_ADDR = 0x27 # I2C device address, if any
error,
 # change this address to 0x3f
LCD_WIDTH = 16 # Maximum characters per line

# Define device constants
LCD_CHR = 1 # Mode - Sending dataLCD_CMD = 0
# Mode - Sending command

LCD_LINE_1 = 0x80 # LCD RAM address for the
1st line
LCD_LINE_2 = 0xC0 # LCD RAM address for the
2nd line
LCD_LINE_3 = 0x94 # LCD RAM address for the
3rd line
LCD_LINE_4 = 0xD4 # LCD RAM address for the
4th line

LCD_BACKLIGHT = 0x08 # On
ENABLE = 0b00000100 # Enable bit

# Timing constants
E_PULSE = 0.0005
E_DELAY = 0.0005

#Open I2C interface
bus = smbus.SMBus(1) # Open I2C interface
for ODROID-C2

# Initialise display
def lcd_init():
 lcd_byte(0x33,LCD_CMD) # 110011 Initialise
 lcd_byte(0x32,LCD_CMD) # 110010 Initialise
 lcd_byte(0x06,LCD_CMD) # 000110 Cursor move
direction
 lcd_byte(0x0C,LCD_CMD) # 001100 Display
On,Cursor Off, Blink Off
 lcd_byte(0x28,LCD_CMD) # 101000 Data
length, number of lines, font size
 lcd_byte(0x01,LCD_CMD) # 000001 Clear
display
 time.sleep(E_DELAY)

# Send byte to data pins
 # (#bits = the data, #mode = 1 for data or
0 for command)
def lcd_byte(bits, mode):
 bits_high = mode | (bits & 0xF0) |
LCD_BACKLIGHT
 bits_low = mode | ((bits<<4) & 0xF0) |
LCD_BACKLIGHT

 bus.write_byte(I2C_ADDR, bits_high) # High
bits
 lcd_toggle_enable(bits_high)

 bus.write_byte(I2C_ADDR, bits_low) # Low
bits
 lcd_toggle_enable(bits_low)

# Toggle enable
def lcd_toggle_enable(bits):
 time.sleep(E_DELAY)
 bus.write_byte(I2C_ADDR, (bits | ENABLE))
 time.sleep(E_PULSE)
 bus.write_byte(I2C_ADDR,(bits & ~ENABLE))
 time.sleep(E_DELAY)

# Send string to display
def lcd_string(message,line):
 message = message.ljust(LCD_WIDTH," ")

 lcd_byte(line, LCD_CMD)

 for i in range(LCD_WIDTH):
 lcd_byte(ord(message[i]),LCD_CHR)

# Main program block, # Initialize display
def main():
 lcd_init()

# Send text to I2C TWI 1602 16x2 Serial LCD
Module Display
while True:

 lcd_string("***ODROID-C2***",LCD_LINE_1)
 lcd_string("ODROID-magazine ",LCD_LINE_2)

 time.sleep(3)

 lcd_string("***HardKernel***",LCD_LINE_1)
 lcd_string("*hardkernel.com*",LCD_LINE_2)

 time.sleep(3)

# Handling keyboard interrupts and exception
utility
if __name__ == '__main__':

 try:
 main()
 except KeyboardInterrupt:
 pass
 finally:
 lcd_byte(0x01, LCD_CMD)
```

Running the code

The above code can be written in any text editor. However, it's easier to do with a Python IDE, such as Python IDLE. The Python IDLE is accessible from the Mate desktop (Application -> Programming -> IDLE). As soon as we write the program, we can save it under any name, and finally run it as shown in Figure 3:

```
$ sudo python lcd16x2i2c.py
```

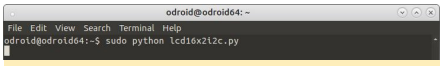


Figure 3 - output from python program

The messages are presented on the LCD module sequentially, 2 lines per time.

Figure 4 - LCD screen displaying a dual-line message

Conclusion

The "Drive I2C LCD screen with ODROID-C2" application can be implemented in any other project with minor modifications as a Python module. The only piece of code that has to be altered in order to change the lines of characters depicted on the LCD display are the following:

```
# Send text to I2C TWI 1602 16x2 Serial LCD
Module Display
while True:

 lcd_string("***ODROID-C2***",LCD_LINE_1)
 lcd_string("ODROID-magazine ",LCD_LINE_2)

 time.sleep(3)

 lcd_string("***HardKernel***",LCD_LINE_1)
 lcd_string("**hardkernel.com",LCD_LINE_2)

 time.sleep(3)
```

Feel free to make any changes to this code and add extra capabilities to any other projects that you might build.

GamODROID-C0: An ODROID-Based Portable Retro Gaming Console

October 1, 2017 By Julien Tiphaine Gaming, ODROID-C0, Tinkering

This article is about yet another homemade portable gaming console as a sequel to the first one that I built (http://bit.ly/2yFj4th). On the first build, I used an ODROID-W (pi clone) and a brand new GameBoy case. For this new project, I wanted something more powerful to run N64, Dreamcast and PSX games, but also some native Linux game. There are not a lot of low power consumption options with sufficient CPU+GPU for that, so I chose an ODROID-C0. Moreover, instead of using and transforming an existing case, I used a 3d printed one designed by myself with optimized dimensions and form factor. I want to thank the ODROID community (forum.ODROID.com), and in particular @meveric for his debian distribution and ODROID-optimized packages.

Components
Here is a list of all components I used for this build:

Main parts:

- ODROID-C0
- 8GB eMMC module
- 128 Gb MicroSD XC (SanDisk Ultra, XC I, class 10)
- 3.5" NTSC/PAL TFT Display (http://bit.ly/2yUyXgd)
- A 4x6cm prototype PCB board

Audio Parts :

- Stereo 2.8W Class D audio amp
- 2 PSP 2000/3000 speakers
- A cheap USB sound card with a small USB cable

Battery Parts:

- 2 LiPo batteries : Keeppower 16650 3.7v 2500mA protected
- 2 MOLEX connectors, 50079-8100
- 2 MOLEX receptacle, 51021-0200

Control parts :

- 12 soft tactile 8mm muttons (http://bit.ly/2xN8qDW)
- 4 tactile button 6mm switches (http://bit.ly/2xNmlcU)
- 2 PSP 1000 analog sticks
- 1 Analog multiplexer MC14051BCL

Cooling parts :

- 2 PS3 GPU copper heatsink
- 4 15x15mm copper heatsink
- Some 1mm Thermal Pad
- Some Silicon Thermal paste

Various other electronic parts:

- A 3mm blue led
- Some wires from an old IDE ribbon cable
- Some breadboard connection wires
- 3 resistors

Decoration parts:

- Some Nail polish templates for colors (black, yellow, red, green, blue)
- 200, 600 and 1200 sandpaper
- XTC 3D (http://bit.ly/2fG3l5O)
- White satin spray paint

Anticipated power consumption
The main sources of power drain are the ODROID-C0, the display and the audio system (soundcard and audio amp). Before starting, I measured the consumption of these 3 components:

- ODROID-C0 : 200-400 mAh depending on CPU and GPU usage
- Audio system : 310 mAh
- Display : 420 mAh

It's a total of 1130 mAh at 5v, so 5650 mAh / hour. The batteries I used are (at least) 3.7v x 5000 mA for a total of 18500 mA. The console should last more than 3h in all cases.

Why use a display with such poor resolution ?

There are several reasons for that: low power, 60 FPS, easy wiring, and it's blurry like old TV which makes cool hardware anti-aliasing.

Why use cylindrical batteries?

It's more a matter of space optimization regarding the capacity I wanted. Using a more classical flat battery would have forced me to make a case deeper than 2cm, although that was my first intention.

Why use a prototype board to mount additional components?

The goal was to easily mount all the components as one unique motherboard, and I can actually say, it was useful!

Why the need for an analog multiplexer?

The ODROID-C0 provides only 2 analog inputs, and one is already used to report the battery level. Thus, only 1 analog input was available for a total of 4 analog axis (2 thumb sticks with 2 directions each). The only way of reading 4 analog axis with one analog input was a multiplexer. And fortunately, the ODROID-C0 has enough digital pins to use 2 of them for analog channels switching.

Why use eMMC module vs. microSD?

The eMMC is much faster than a microSD. It allows the console to boot in a few seconds even with Xorg, a window manager, and Emulation Station with lots of games. I use the eMMC for the operating system, and the microSD for the games and video previews.

3D printed case

The console case has been modeled with Freecad. I designed it specifically for this project and the very specific size of the motherboard and all components. It was my first 3D model and first 3D print, so it may contains errors. However, the Freecad files are available on **GitHub (http://bit.ly/2fGJWRU)** and the STL files are freely **distributed on Thingverse (http://bit.ly/2xW9FAh).**

Figure 1 – Front internal view of the case. The black points are marks to make holes for skewing.

Figures 2 and 3 – Back internal view of the case. You can see batteries space at the bottom and some striations for CPU + GPU thermal dissipation.

The whole case is nearly the same as a Nintendo DS. It may not be obvious, but using the dimensions of a well known console allowed me to find good and cheap protection cases. As you can see on photos later, I used an NDS case to protect my GamODROID-C0, which I found for a few Euros.

To obtain a nice finish, I first used 600 and 1200 sandpaper on all parts. Then, I used a product called XTC-3D. It's awesome and give a nice brilliant finish, but it's still not a good enough finish for me. I used some 1200 sandpaper again before using a white satin aerosol painting. This gave me the finish that you see on photos below.

For small parts like buttons and the D-pad, I used some nail polish. It's very cheap and actually provided great brilliant finish. I finalized the buttons and D-pad with some transparent nail varnish to protect colors, since buttons are the most used part of the console.

Figure 4 – Hardware Assembly

My goal was to build a one-piece motherboard in order to make it more robust and easier to put inside the case. I also built small boards for buttons, D-pad and start/select buttons.

Display hack

The hack is roughly the same as the one I did for my Retroboy console (**http://bit.ly/2yFj4th**). However, there was some differences on the connector side : V-in and composite output was reversed this time. Figure 5 shows the original display, as found on the Adafruit website.

Figure 5 – 3.5" display

I first removed the white connector, then wired the V-in directly to the voltage regulator output and added two wires for powering through one of the ODROID 5V pin, as shown in Figure 6.

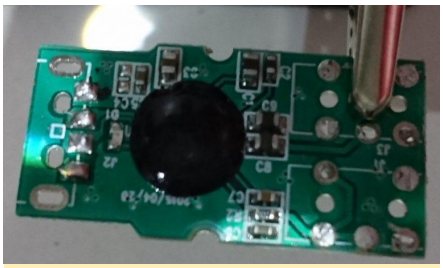
Figure 6 – Closeup of the ODROID 5V wiring

Sound card

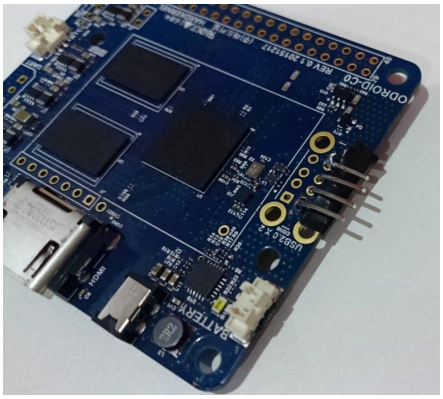
I chose a cheap USB sound card with a wire between the board and the USB connector. It was important because it was easier to unsolder.

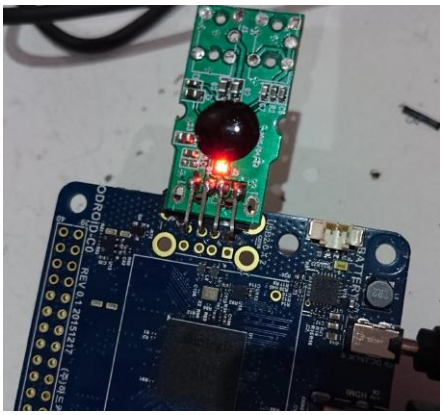
Figure 7 – USB sound card

I started to dismantle wires, connectors and then re-drilled the holes. I prepared the ODROID board by adding pins to the first USB connector, as shown in Figures 8 and 9. Finally, I soldered the sound card directly on the pins, as shown in Figure 10.



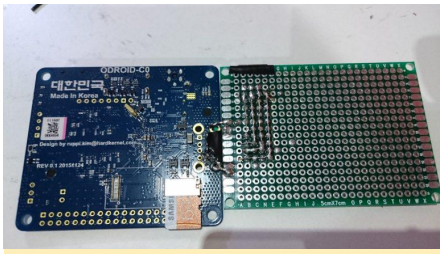
Figures 8, 9 and 10 – Modifications to the sound card



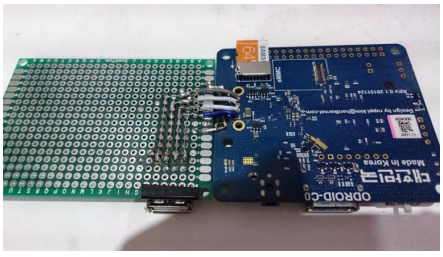


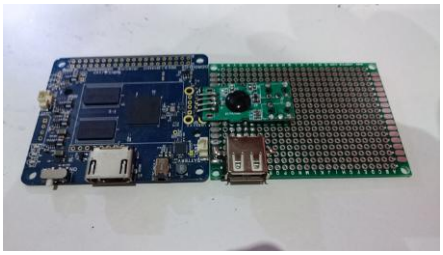
Extension board with USB port

I put the extension board just below the USB sound card. I first soldered a USB connector, then I wired it to the second ODROID USB connector through the extension board. Note that I also soldered the extension board to the ODROID motherboard to make the whole thing more robust.



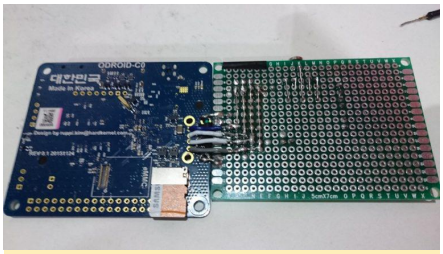
Figures 11, 12 and 13 – Attaching the extension board to the ODROID



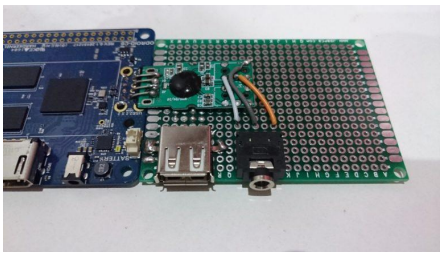


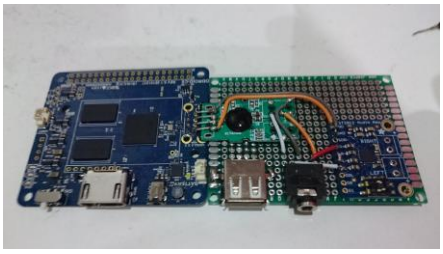
Finishing audio on the extension board

Having a sound card with analog output is nice, but a 3.5 audio jack and a good amp to drive the speakers is better, which was exactly the next step: wiring and soldering components on the extension board.



Figures 14, 15 and 16 – Wiring and soldering the components onto the extension board)





Analog multiplexer wiring

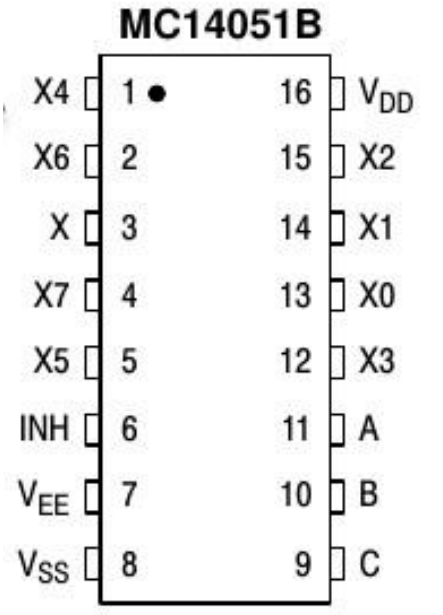
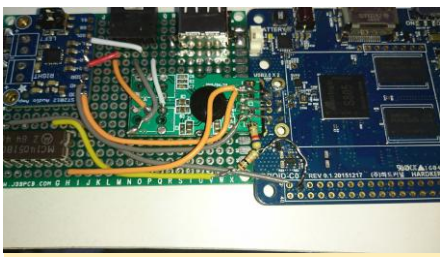
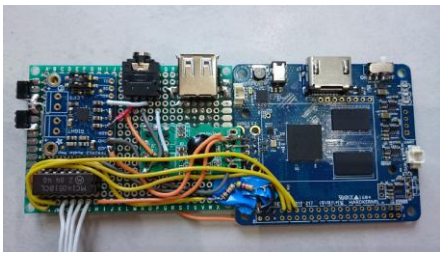


Figure 17 – Soldering diagram for the extension board

The soldering of this small piece started to add a lot of wires and ended up filling the extension board. I had to use the following: Vdd (Vin), Vss (ground), x (analog output), x0, x1, x2, x3 (analog inputs), A, B (digital switches). C was not needed as 2 switches were enough to switch the first 4 outputs. Vee and INH has been wired to ground. Note that I made a voltage divider bridge between x (output) and the analog input of the ODROID. This is because the PSP analog sticks and MC14051B operate in 5V whereas the ODROID-C0 analog input accept a maximum of 1.8v.



Figures 18 and 19 – Closeup of the analog multiplexer wiring



Volume buttons

You may have noticed on the previous photo that there were also 2 push buttons on one edge of the extension board. I wired them to GPIO pins to control audio volume, as shown in Figure 20.

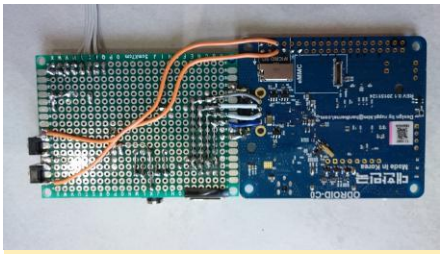


Figure 20 – Volume button wiring

Start/Select buttons

I used push buttons for start and select buttons. I mounted them on an small additional board together with a blue led for battery monitoring.

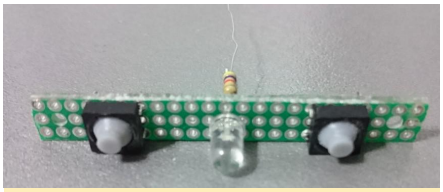


Figure 21 – Start and select button wiring

Batteries

As indicated before, I used a pair of protected cylindrical LiPo batteries. I wired them in parallel to get 5000 mA. I had to solder some wires directly onto the batteries and added a Molex connector to be able to connect the two wired batteries to the ODROID-C0 LiPo connector.

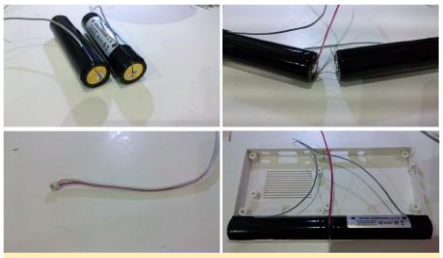
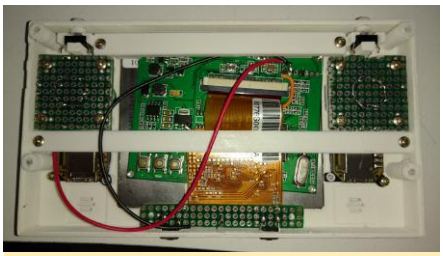


Figure 22 – Details of the battery wiring

Mounting the components

At this time, I had done everything on the hardware side. I started to mount in the front part of the case the display, analog sticks, D-pad, a-b-x-y boards and L1 + R1 buttons. The display is not glued but maintained with two traversal bars. As you will see in Figure 23, these bars allowed me to also block and drive all of the wires.



Figures 23 and 24 – Steps of the final assembly of the components inside the case

The next step for the front part of the case is adding speakers, start/select buttons board and wiring everything with a common ground. The final steps before closing are to add a heat-sink, putting L2+R2 buttons and the motherboard in the back part of the case, then soldering everything to GPIO. Note also the yellow wire which is the composite output of the ODROID that go to the display input 1.

Figures 25 and 26 – All of the components are fitted into the case before closing

Figures 27, 28 and 29 – The outside of the case after final assembly

Software

I created a script that constructed 80% of the system including a copy of specific configuration files. The other 20% are for ROMs and personal customization. If someone wants to do the same, it should be quite easy to adapt and re-run the script.

Before starting to comment the install script, here are the preparatory install steps that I did:

- Deployment of **@meveric's minimal Debian Jessie image on the eMMC (http://bit.ly/2yF2PML)**
- Created two partitions on the 128GB microSD: 4 Gb for save states and the rest for ROMs, which will be mounted at /mnt/states and /mnt/ressources). I did 2 partitions because I had the intention to create a read-only system except for states, but I finally kept a full read/write system.
- Created a GameODROID folder in /root and copied the install script and its dependencies

Installation Script

The installation script and all dependencies can be found on GitHub at **http://bit.ly/2fGJWRU**. It is organized with functions dedicated for each steps.

The first executed function creates custom mount points, copies custom fstab and activates tmpfs:

```
function fstab
{
 echo "fstab and filesystem"

 mkdir -p /mnt/states
 mkdir -p /mnt/ressources

 cp /root/GameODROID/fstab /etc/fstab

 sed -i "s/#RAMLOCK=yes/RAMLOCK=yes/"
/etc/default/tmpfs
 sed -i "s/#RAMSHM=yes/RAMSHM=yes/"
/etc/default/tmpfs
}
```

The custom fstab file allows to change mount options in order to optimize for speed (noatime, discard) and use a small tmpfs partition for /var/log:

```
tmpfs /var/log tmpfs
nodev,nosuid,noatime,size=20M 0 0
```

After this first function, the system is rebooted, then upgraded and rebooted again:

```
function uptodate
{
 echo "update"
 apt-get update
 apt-get upgrade
 apt-get dist-upgrade
}
```

The final step of this stage is to install all of the necessary base packages (function syspackages). There is nothing special here except for two things:

- evilwm : I had to use a window manager because some native games can't find the native screen resolution without it. I found that evilwm was a very good candidate for the console, since it is very light and invisible with default settings.
- Antimicro-ODROID : it's a very nice piece of software that I did not know about before. It allows me to map any keyboard and mouse event to the joypad.
- Python package evdev: used to configure reicast input
- I used an ODROID C1/C0 specific xorg configuration file supplied by @meveric (**http://bit.ly/2xaSonP**)

Games

This part correspond to the functions "emulators", "emulators_glupen64_meveric" and nativegames. Except for Dreamcast games for which I used reicase, all other emulators are part of Retroarch:

- pcsx-rearmed (PSX)
- fbalpha (CPS2)
- gambatte (Gameboy color)
- gpsp (Gameboy advance)
- mednafen-pce-fast (Pc-Engine + Cdrom)
- nestopia (Nes)
- picodrive (Sega 32X, SegaCD)
- pocketnes (Snes)
- genesis-plus-gx (GameGear, Genesis, MasterSystem)
- mednafen-ngp (Neogeo pocket color)

For native games, I selected those that were enjoyable with a gamepad and running correctly on the ODROID-C0 with a small screen:

- hurrican
- hcraft
- frogatto
- SuperMario War
- astromenace
- neverball
- shmupacabra
- aquaria
- Revolt
- Open JK3
- openjazz
- supertuxkart
- mars
- puzzlemoppet
- opentyrian
- pushover

Game launcher

This corresponds to the function "userinterface". Initially, I wanted to use Attract mode. Unfortunately, the implementation of GLES on ODROID-C0/C1 does not seem to include glBlendEquationSeparateOES() and glBlendFuncSeparateOES() functions, which are mandatory to compile libFSML, which in turn is mandatory to compile

Attract mode. Thus, I used the latest Emulation Station version with video preview support. Since I wanted to change the default splash screen with a custom one, I had to replace "splash_svg.cpp" file in "EmulationStation/data/converted". This file is a simple C array that contains the bytes of an SVG file. Despite the classical configuration of systems, I create a specific one that list two scripts to change the display: internal screen or HDMI (see the composite.sh and hdmi.sh scripts).

Specific tools

This correspond to the function "localtools". This is mainly to handle the custom GPIO gamepad. I had to wrote a small program in C that creates a gamepad through Linux's uinput and poll GPIO to generate events. I used polling instead of IRQ because the SoC does not have enough IRQ to handle all the buttons. I named this tool gpio_joypad and the source code is on GitHub at **http://bit.ly/2xaTdgp**. It also handles the analog multiplexer to get left and right analog thumb sticks values.

Boot config file

This correspond to the function "bootini". This function consists in copying a customized boot.ini file to the boot partition. The important changes I made are:

- Keeping only two video modes : cvbs480 (activated by default) and vga (commented out)
- Disabled cec and vpu
- Modified kernel arguments:
  - "cvbsmode=480cvbs" to get a 60Hz NTSC resolution instead of 50 Hz PAL
  - "max_freq=1824" to overclock the SoC (needed for N64 and Dreamcast emulators)
  - "quiet loglevel=3 rd.systemd.show_status=false udev.log-priority=3" to make the boot as quiet as possible

Initially, I wanted to display the splash screen early during the boot process. It is well documented on ODROID wiki, but unfortunately it works only for 720p resolutions.

Launch everything at start

This correspond to the function "startup". The automatic startup of X and Emulationstation at boot consisted in a custom tty1 service in systemd that launch agetty with autologin, a BASH profile that launch X when tty variable = tty1, and finally a xinitrc that start the window manager and Emulation Station.

/etc/systemd/system/getty@tty1.service.d/override.conf

```
[Service]

ExecStart=

ExecStart=-/sbin/agetty --autologin root --
noclear %I $TERM
The bash /root/.profile :
# ~/.profile: executed by Bourne-compatible
login shells.

if [ "$BASH" ]; then
 if [ -f ~/.bashrc ]; then
 . ~/.bashrc
 fi
fi

if [ "$(tty)" = "/dev/tty1" ] ; then
 /usr/local/bin/battery.sh &
 /usr/local/bin/gpio-joypad &
 startx -- -nocursor 2>&1 &
fi

mesg n
```

/root/.xinitrc

```
# a WM is needed some software are correctly
sized in full screen
# e.g : emulationstation, rvgl
evilwm & pid=$!

emulationstation.sh &

# this allows not to shutdown X when
emulation is killed
# We want that because we have to kill it
after gamelaunch
# else it does not reappear on screen
(SDL_Createwindow() does never end)
wait $pid
Note that the bash profile start the joypad
driver (gpio_joypad) and the battery
monitoring script (battery.sh) before
starting X.
The battery monitoring script is not very
accurate, but I dit not found any way to
make a better monitoring to switch on the
led on low battery or when charging:
#!/bin/bash

PIN=75
GPIO=/sys/class/gpio
ACCESS=$GPIO/gpio$PIN
LOWBAT=780
CHARGING=1020

if [ ! -d $ACCESS ] ; then
 echo $PIN > $GPIO/export
 echo out > $ACCESS/direction
 echo 0 > $ACCESS/value
fi

while true
do
 ADCVAL=$(cat /sys/class/saradc/saradc_ch0)
# echo "value : $ADCVAL"
 # charging
 if [ $ADCVAL -gt $CHARGING ]; then
 echo 1 > $ACCESS/value
 else
 # low bat
 if [ $ADCVAL -lt $LOWBAT ]; then
 echo 1 > $ACCESS/value
 sleep 1
 echo 0 > $ACCESS/value
 else
 echo 0 > $ACCESS/value
 fi
 fi

 sleep 2
done
```

Finalize and clean up

This correspond to the function "optimize_system". In this function, the BASH login message is hidden (to make the boot process as silent as possible) and packages cache is cleaned (apt-get clean). There are also two configuration files that are deployed. The custom journald.conf is here to write logs in ram instead of disk for better performance:

```
[Journal]
Storage=volatile
I also created a specific alsa configuration
file to add latency and buffers, so most
sound stutering are avoided for n64 and
dreamcast games:
pcm.!default {
 type plug
 slave.pcm "softvol"
 ttable.0.1 0.8
 ttable.1.0 0.8
}
pcm.dmixer {
 type dmix
 ipc_key 1024
 slave {
 pcm "hw:1,0"
 period_time 0
 period_size 2048
 buffer_size 65536
 rate 44100
 }
 bindings {
 0 0
 1 1
 }
}
pcm.dsnooper {
 type dsnoop
 ipc_key 1024
 slave {
 pcm "hw:1,0"
 channels 2
 period_time 0
 period_size 2048
 buffer_size 65536
 rate 44100
 }
 bindings {
 0 0
 1 1
 }
}
pcm.softvol {
 type softvol
 slave { pcm "dmixer" }
 control {
 name "Master"
 card 1
 }
}
ctl.!default {
 type hw
 card 1
}
ctl.softvol {
 type hw
 card 1
}
ctl.dmixer {
 type hw
 card 1
}
```

Global Retroarch configuration

Despite changing buttons and path, I had to adapt some videos parameters of retroarch (root/.config/retroarch/retroarch.cfg) to optimize performance and better suit the hardware:

```
video_refresh_rate = "59.950001"
video_monitor_index = "0"
video_fullscreen_x = "720"
video_fullscreen_y = "480"
video_vsync = "true"
video_threaded = "true"
video_force_aspect = "true"
```

Core-specific configuration
I also did some adjustments on a of the emulator cores:

Allowing 6 buttons for SegaCD and 32X:

```
picodrive_input1 = "6 button pad"
Changing glupen64 parameters to optimize
rendering on the ODROID SoC:
glupen64-cpucore = "dynamic_recompiler"
glupen64-rspmode = "HLE"
glupen64-43screensize = "320x240"
glupen64-BilinearMode = "standard"
Allowing PSX analog joypad support:
pcsx_rearmed_pad1type = "analog"
```

For the Dreamcast emulator, I used reicast-joyconfig (**http://bit.ly/2fLE1yH**) to generate the gamepad config and copied the resulting file to /root/.config/reicast/joy.conf. I also changed the fullscreen resolution to adapt it to the CVBS display:

```
[x11]
fullscreen = 1
height = 480
width = 720
```

Keyboard and mouse mapping for native games
Some native games work fine, but require either a mouse or a keyboard for special keys such as Esc, Enter, Space, Shift and the arrow keys. To map these keys to the console gamepad, I used antimicro. It's a very nice and easy-to-use program to map any mouse and keyboard key to any gamepad buttons.

Scraping videos
Emulation Station has an integrated scraper for game informations and pictures, but not for videos. Moreover, if video previews are supported depending on the chosen themes, they are played through VLC, which is not accelerated on the ODROID-C0/C1 SoC. The consequence is that 320×240@30 FPS in h.264 is the biggest playable size. I wrote and used a custom script available on GitHub at **http://bit.ly/2fGFSkU**, which parses the Emulation Station game folder and scrapes videos from **www.gamesdatabase.org**.

Lessons learned

- There is no way to correctly monitor the battery on an ODROID-C0
- With only a Mali 450 GPU, even with overclock, it is still too slow for a lot of N64 and Dreamcast games
- There are some crashes that seem to be related to the graphics driver, such as Emulation Station not exiting properly, and hurrican sometimes does not start with the correct resolution
- It is not possible to use a proper interrupt-based joypad driver, since there are not enough IRQs available on the SoC.
- There is a need for a window manager, otherwise fullscreen is not available for games and Emulation Station
- Reicast seems to emulate the GDRom noise, but I actually find it annoying

Figure 30 – Sega emulator running on the GamODROID-C0

You can check out the GamODROID-C0 in action at **https://youtu.be/3hxYhH7AFYU**. For comments, questions and suggestions, please visit the original blog post at **http://bit.ly/2khNDTz**.

Android Development: Android Content Provider

October 1, 2017 By Nanik Tolaram Android

Like any other operating system, Android internally needs to have persistence storage for storing system information. This data needs to be in persistent storage, as it will always need to refer to those data after every reboot to put the device in a particular state. User and device information like screen brightness, volume, accounts, calendar, etc will need to be stored somewhere. Android uses what is called Content Provider. Basically, it is a SQLite-backed persistent mechanism, or more simply known as a database. Most of the data is internally stored inside of several SQLite databases. In this article we will take a look at some of the content providers that are used internally by the operating system.

This article will look at some of the databases that are used internally by the operating system. A good starting point to learn more about content provider is to go to the Android Developer website from Google (**http://bit.ly/2hkvljq**).

What and where

Content providers are just normal Android applications that have a job to serve and process database requests from a client. Data from the internal content providers are stored inside /data/data folder as shown in Figure 1. We are interested in apps that has the following package format:

```
com.android.providers.< app_name >
```

Figure 1 list the internal Android content providers that are available.

```
u0_a0     u0_a0     4096  1970-07-21  20:28:00 com.android.providers.blockednumber
u0_a4     u0_a4     4096  1970-07-21  20:29:00 com.android.providers.calendar
u0_a0     u0_a0     4096  1970-07-21  20:29:00 com.android.providers.contacts
u0_a9     u0_a9     4096  2017-08-28  22:20:00 com.android.providers.downloads
u0_a9     u0_a9     4096  1970-07-21  20:29:00 com.android.providers.downloads.ui
u0_a9     u0_a9     4096  1970-07-21  20:29:00 com.android.providers.media
system    system    4096  1970-07-21  20:29:00 com.android.providers.settings
radio     radio     4096  1970-07-21  20:28:00 com.android.providers.telephony
u0_a0     u0_a0     4096  2017-08-28  21:07:00 com.android.providers.userdictionary
```

Figure 1 – Package content providers inside /data/data folder

Take, for example, the DownloadManager service that is provided by the Android SDK. This service allow apps to download file asynchronously. Internally, the framework uses this content provider to keep persistent information about a file status that is going to be downloaded. The follow SQL schema shows the declaration that is used internally to persist the downloaded file information.

```
CREATE TABLE android_metadata (locale TEXT);

CREATE TABLE downloads(_id INTEGER PRIMARY
KEY AUTOINCREMENT,uri TEXT, method INTEGER,
entity TEXT, no_integrity BOOLEAN, hint
TEXT, otaupdate BOOLEAN, _data TEXT,
mimetype TEXT, destination INTEGER,
no_system BOOLEAN, visibility INTEGER,
control INTEGER, status INTEGER, numfailed
INTEGER, lastmod BIGINT, notificationpackage
TEXT, notificationclass TEXT,
notificationextras TEXT, cookiedata TEXT,
useragent TEXT, referer TEXT, total_bytes
INTEGER, current_bytes INTEGER, etag TEXT,
uid INTEGER, otheruid INTEGER, title TEXT,
description TEXT, scanned BOOLEAN,
is_public_api INTEGER NOT NULL DEFAULT 0,
allow_roaming INTEGER NOT NULL DEFAULT 0,
allowed_network_types INTEGER NOT NULL
DEFAULT 0, is_visible_in_downloads_ui
INTEGER NOT NULL DEFAULT 1,
bypass_recommended_size_limit INTEGER NOT
NULL DEFAULT 0, mediaprovider_uri TEXT,
deleted BOOLEAN NOT NULL DEFAULT 0, errorMsg
TEXT, allow_metered INTEGER NOT NULL DEFAULT
1, allow_write BOOLEAN NOT NULL DEFAULT 0,
flags INTEGER NOT NULL DEFAULT 0);

CREATE TABLE request_headers(id INTEGER
PRIMARY KEY AUTOINCREMENT,download_id
INTEGER NOT NULL,header TEXT NOT NULL,value
TEXT NOT NULL);

CREATE TABLE android_metadata (locale TEXT);

CREATE TABLE downloads(_id INTEGER PRIMARY
KEY AUTOINCREMENT,uri TEXT, method INTEGER,
entity TEXT, no_integrity BOOLEAN, hint
TEXT, otaupdate BOOLEAN, _data TEXT,
mimetype TEXT, destination INTEGER,
no_system BOOLEAN, visibility INTEGER,
control INTEGER, status INTEGER, numfailed
INTEGER, lastmod BIGINT, notificationpackage
TEXT, notificationclass TEXT,
notificationextras TEXT, cookiedata TEXT,
useragent TEXT, referer TEXT, total_bytes
INTEGER, current_bytes INTEGER, etag TEXT,
uid INTEGER, otheruid INTEGER, title TEXT,
description TEXT, scanned BOOLEAN,
is_public_api INTEGER NOT NULL DEFAULT 0,
allow_roaming INTEGER NOT NULL DEFAULT 0,
allowed_network_types INTEGER NOT NULL
DEFAULT 0, is_visible_in_downloads_ui
INTEGER NOT NULL DEFAULT 1,
bypass_recommended_size_limit INTEGER NOT
```

```
NULL DEFAULT 0, mediaprovider_uri TEXT,
deleted BOOLEAN NOT NULL DEFAULT 0, errorMsg
TEXT, allow_metered INTEGER NOT NULL DEFAULT
1, allow_write BOOLEAN NOT NULL DEFAULT 0,
flags INTEGER NOT NULL DEFAULT 0);

CREATE TABLE request_headers(id INTEGER
PRIMARY KEY AUTOINCREMENT,download_id
INTEGER NOT NULL,header TEXT NOT NULL,value
TEXT NOT NULL);
```

The following code block shows an example of a data is stored for the downloaded file:

```
1|https://www.gstatic.com/android/config_upd
ate/08202014-
metadata.txt|0|||||/data/user/0/com.android.
providers.downloads/cache/08202014-
metadata.txt|text/plain|2||2||200|0||com.goo
gle.android.configupdater|||||0|||||0820201
4-metadata.txt|||1|1|-1|0|0||0||1|0|0

2|http://www.gstatic.com/android/config_upda
te/07252017-sms-
blacklist.metadata.txt|0|||||/data/user/0/co
m.android.providers.downloads/cache/07252017
-sms-
blacklist.metadata.txt|text/plain|2||2||200|
0||com.google.android.configupdater||||||385
|385|||07252017-sms-
blacklist.metadata.”txt|||1|1|-1|0|0||0||1|0
|0
```

Content provider declaration

The content providers provided by the operating system, which are not all made available to a user application, normally have the following declaration in their AndroidManifest.xml:

Listing 1 – Settings content provider

```
< manifest
xmlns:android="http://schemas.android.com/ap
k/res/android"
package="com.android.providers.settings"
coreApp="true"
android:sharedUserId="android.uid.system">
    < application
android:allowClearUserData="false"
android:label="@string/app_label"
android:process="system"
android:backupAgent="SettingsBackupAgent"
android:killAfterRestore="false"
android:icon="@mipmap/ic_launcher_settings"
android:defaultToDeviceProtectedStorage="tru
e" android:directBootAware="true" >
        < provider
android:name="SettingsProvider"
android:authorities="settings"
android:multiprocess="false"
android:exported="true"
android:singleUser="true"
android:initOrder="100" />
    < /application >
< /manifest >
```

Listing 1 is the AndroidManifest.xml for the Settings application which is stored under the package com.android.providers.settings. Another example can be seen in Listing-2 which shows the declaration for Contacts Provider used to stored contacts information:

Listing 2 – Contacts content provider

```
< manifest
xmlns:android="http://schemas.android.com/ap
k/res/android"
package="com.android.providers.contacts"
android:sharedUserId="android.uid.shared"
android:sharedUserLabel="@string/sharedUserL
abel" >
    < uses-permission
android:name="android.permission.BIND_DIRECT
ORY_SEARCH" />
    < uses-permission
android:name="android.permission.GET_ACCOUNT
S" />
     .....
     .....
     .....
    < permission
android:name="android.permission.SEND_CALL_L
OG_CHANGE" android:label="Broadcast that a
change happened to the call log."
android:protectionLevel="signature|system"
/>

        < provider
android:name="ContactsProvider2"
android:authorities="contacts;com.android.co
ntacts"
android:label="@string/provider_label" …...
…... />
                .....
                .....
                .....
        < /provider >
```

The following table lists some of the content providers that exist inside Android version 7.1.2:

Description	Package Name	Source Location
CalendarProvider	com.android.providers.calendar	
ContactsProvider	com.android.providers.contacts	
DownloadProvider	com.android.providers.downloads com.android.providers.downloads.ui	
MediaProvider	com.android.providers.media	
SettingsProvider	com.android.providers.settings	frameworks/base/packages/SettingsProvider/src/com/android/providers/settings/SettingsProvider.java
TelephonyProvider	com.android.providers.telephony	packages/providers/TelephonyProvider/src/com/android/providers/telephony/TelephonyProvider.java
UserDictionaryProvider	com.android.providers.userdictionary	
BlockedNumberCall	com.android.providers.blockednumber	
PartnerbookmarksProvider		packages/providers/PartnerBookmarksProvider/src/com/android/providers/partnerbookmarks/PartnerBookmarksProvider.java
EmailProvider		packages/apps/Email/provider_src/com/android/email/provider/EmailProvider.java
LauncherProvider		packages/apps/Launcher3/src/com/android/launcher3/LauncherProvider.java
CellBroadcastReceiver		packages/apps/CellBroadcastReceiver/src/com/android/cellbroadcastreceiver/CellBroadcastContentProvider.java
WearPackageIconProvider.java		packages/apps/PackageInstaller/src/com/android/packageinstaller/wear/WearPackageIconProvider.java
GalleryProvider		packages/apps/Gallery2/src/com/android/gallery3d/provider/GalleryProvider.java
DeskClock		packages/apps/DeskClock/src/com/android/deskclock/provider/ClockProvider.java
SearchRecentSuggestionsProvider		frameworks/base/core/java/android/content/SearchRecentSuggestionsProvider.java
RecentsProvider		frameworks/base/packages/DocumentsUI/src/com/android/documentsui/RecentsProvider.java
MtpDocumentsProvider		frameworks/base/packages/MtpDocumentsProvider/src/com/android/mtp/MtpDocumentsProvider.java
ExternalStorageProvider		frameworks/base/packages/ExternalStorageProvider/src/com/android/externalstorage/ExternalStorageProvider.java
BugreportStorageProvider		frameworks/base/packages/Shell/src/com/android/shell/BugreportStorageProvider.java

ODROID-MC1 Parallel Programming: Getting Started

October 1, 2017 By Andy Yuen ODROID-MC1

This guide is not meant to teach you how to write parallel programs on the ODROID-MC1. It is meant to provide you with an environment ready for experimenting with MPJ Express, a reference implementation of the mpiJava 1.2 API. An MPJ Express parallel program that generates Mandelbrot images has been provided for you to run on any machine or cluster that has the the Java SDK installed: ARM or INTEL. If there is sufficient interest expressed for information on MPJ Express programming, we can write a tutorial for a future edition of the magazine.

Why parallel programming?

Parallel programming or computing is a form of computation in which many independent calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved at the same time. In short, its aim includes:

- Increase overall speed,
- Process huge amount of data,
- Solve problems in real time, and
- Solve problems in due time

Why now?

Many people argue whether **Moore's Law** still holds. Moore's law is the observation that the number of transistors in a dense integrated circuit doubles approximately every two years (some say 18 months). Moore's Law is named after Gordon E. Moore, the co-founder of INTEL and Fairchild Semiconductor. It is this continuous advancement of integrated circuit technology that has brought us from the original 4.77 megahertz PC to the current multi-gigahertz processors. The processor architecture has also changed a lot with multiple execution pipelines, out-of-order execution, caching, etc.

Assuming Moore's Law still applies, we are still faced with big problems in improving our single CPU performance:

- **The Power Wall:** Power = C * Vdd2 * Frequency

We cannot scale transistor count and frequency without reducing Vdd (supply voltage). Voltage scaling has already stalled.

- **The Complexity Wall:** Debugging and verifying large OOO (Out-Of-Order) cores is expensive (100s of engineers for 3-5 years). Caches are easier to design but can only help so much.

As an example of the power (frequency) wall, it has been reported that:

- **E5640 Xeon (4 cores @ 2.66 GHz) has a power envelope of 95 watts**
- **L5630 Xeon (4 Cores @ 2.13 GHz) has a power envelope of 40 watts**

This implies an increase of 137% electrical power for an increase of 24% of CPU power. At this rate, it is not going to scale.

Enter multi-core design. A multi-core processor implements multiprocessing in a single physical package. Instead of cranking up the frequency to achieve higher performance, more cores are put in a processor so that programs can be executed in parallel to gain performance. These days, all INTEL processors are multicore. Even the processors used in mobile phone are all multi-core processors.

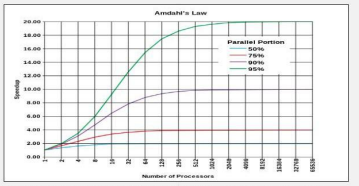
Limitations on performance gains

How much improvement can I expect for my application to gain running on a multi-core processor? The answer is that it depends. You application may not have any performance gain at all if it has not been designed to take advantage of multi-core capability. Even if it does, it still depends on the nature of your program and the algorithm it is using.

Amdahl's law states that if P is the proportion of a program that can be made parallel, and (1−P) is the proportion that cannot be parallelised, then the maximum speedup that can be achieved by using N processors is:

- **1/[(1-P) + (P/N)]**

The speedup in relation to the number of cores or processors at specific values of P is shown in the graph below.



Amdahl's law states that if P is the proportion of a program that can be made parallel, and (1−P) is the proportion that cannot be parallelised, then the maximum speedup that can be achieved by using N processors = 1/[(1-P) + (P/N)]

Figure 1 – Amdahl's Law

This gives you some perspective on how much performance you may be able to gain by writing your program to take advantage of parallelism instead of having unreal expectations.

Why do parallel programming in Java?

Some of the advantages of writing parallel programs in Java include:

- Write once, run anywhere,
- Large pool of software developers,
- Object Oriented (OO) programming abstractions,
- Compile time and runtime checking of code,
- Automatic garbage collection,
- Supports multi-threading in language, and
- Rich collection of libraries

Java supported multi-threading since its inception, so what is new? Java multithreading uses the Shared Memory Model, meaning that it cannot be scaled to use multiple machines.

A Distributed Memory Model refers to a multiprocessor computer system, such as an ODROID-MC1, in which each processor has its own private memory. Computational tasks can only operate on local data, and if remote data is required, the computational task must communicate with one or more remote processors. In contrast, a Shared Memory multiprocessor offers a single memory space used by all processors. Processors do not have to be aware where data resides, except that there may be performance penalties, and that race conditions are to be avoided.

The MPJ Express message passing library

MPJ Express is a reference implementation of the mpiJava 1.2 API, which is the Java equivalent of the MPI 1.1 specification. It allows application developers to write and execute parallel applications for multicore processors and compute clusters using either a multicore configuration (shared memory model) or a cluster configuration (distributed memory model) respectively. The latter also supports a hybrid approach to run parallel programs on a cluster of multicore machines such as the ODROID-MC1.

All the software dependencies have already been installed on the SD card image I provided. My mpj-example project on Github My mpj-example project on Github has also been cloned and compiled. The resultant jar file and a dependent file have been copied to the ~/mpj_rundir directory where you can try out in either multicore or cluster mode. All MPJ Express documentations can be found in the $MPJ_HOME/doc directory.

Fractal Generation using MPJ Express

The mpj_example project is a Mandelbrot generator. Mandelbrot set images are made by sampling complex numbers and determining for each number whether the result tends towards infinity when the iteration of a particular mathematical operation is performed. The real and imaginary parts of each number are converted into image coordinates for a pixel coloured according to how rapidly the sequence diverges, if at all. My MPJ Express parallel program assigns each available core to compute one vertical slice of the Mandelbrot set image at a time. Consequently, the more cores are available, the more work can be performed in parallel. Mandelbrot images at specific coordinates are shown in the following images.

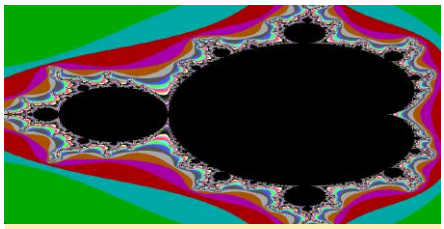


Figure 2.1 – mandelbrot1: (-0.5, 0.0)

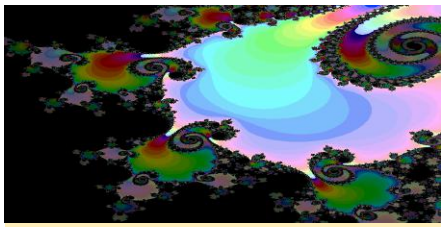


Figure 2.2 – mandelbrot2: (-0.7615134027775, 0.0794865972225)

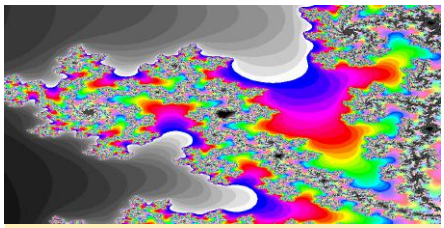


Figure 2.3 – mandelbrot3: (0.1015, -.633)

These Mandelbrot images are generated using the following commands on a single machine, the master node, using a multicore configuration. From the master command prompt, issue the following commands:

```
cd ~/mpj_rundir

mpjrun.sh -np 1 -cp ./mpj-example-1.0.jar com.kardinia.mpj.ColourMandelbrot -.5 0 2
colourMap.txt mandelbrot1.png

mpjrun.sh -np 1 -cp ./mpj-example-1.0.jar com.kardinia.mpj.ColourMandelbrot
-0.7615134027775 0.0794865972225 0.0032285925920 colourMap.txt mandelbrot2.png

mpjrun.sh -np 1 -cp ./mpj-example-1.0.jar com.kardinia.mpj.ColourMandelbrot 0.1015 -.633
0.01 colourMap.txt mandelbrot3.png
```

Figure 3 – Mandelbrot commands

You can rerun the above command with -np values between 1 and 8 inclusive to see the difference in performance by varying the number of cores used for Mandelbrot generation. Remember that the XU4 has 4 little A7 and 4 big A15 cores.

The parameters after com.kardinia.mpj.ColourMandelbrot are:

- parameter 1: starting x coordinate
- parameter 2: starting y coordinate
- parameter 3: step size
- parameter 4: color map for mapping number of iterations to a particular colour
- parameter 5: filename to save the generated mandelbrot

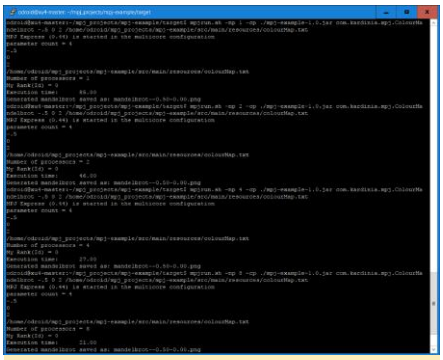


Figure 4 – A screenshot of running Mandelbrot Generator in multicore mode.

To run the Mandelbrot Generator in cluster mode, follow the instructions below:

A text file named "machines" which contains the hostnames of every node in you ODROID-MC1 cluster on separate lines is required. The machines file that is in the ~/mpj_rundir contains the following 4 lines:

```
xu4-master
xu4-node1
xu4-node2
xu4-node3
```

To start the MPJ daemon on each node, issue the command below once from the master node to start a MPJ daemon on each node:

```
$ mpjboot machines
```

Then issue the following commands from the master node:

```
mpjrun.sh -np 4 -dev hybdev -cp ./mpj-example-1.0.jar
com.kardinia.mpj.ColourMandelbrot -.5 0 2 colourMap.txt mandelbrot1.png

mpjrun.sh -np 4 -dev hybdev -cp ./mpj-example-1.0.jar
com.kardinia.mpj.ColourMandelbrot -0.7615134027775 0.0794865972225
0.0032285925920 colourMap.txt mandelbrot2.png

mpjrun.sh -np 4 -dev hybdev -cp ./mpj-example-1.0.jar
com.kardinia.mpj.ColourMandelbrot 0.1015 -.633 0.01 colourMap.txt
mandelbrot3.png
```

Figure 5 – Master node commands

Again, you can vary the number after -np between 4 and 32 as there are a total of 32 cores in your ODROID-MC1 cluster. The screenshot below shows running the above commands in cluster mode.

```
mpjrun.sh -np 4 -dev hybdev -cp ./mpj-example-1.0.jar
com.kardinia.mpj.ColourMandelbrot -.5 0 2 colourMap.txt mandelbrot1.png

mpjrun.sh -np 4 -dev hybdev -cp ./mpj-example-1.0.jar
com.kardinia.mpj.ColourMandelbrot -0.7615134027775 0.0794865972225
0.0032285925920 colourMap.txt mandelbrot2.png

mpjrun.sh -np 4 -dev hybdev -cp ./mpj-example-1.0.jar
com.kardinia.mpj.ColourMandelbrot 0.1015 -.633 0.01 colourMap.txt
mandelbrot3.png
```

Figure 6 – Running Mandelbrot Generator in cluster mode

When you are done with experimenting with the cluster mode, issue the following command from the master to terminate all the MPJ daemons started earlier:

```
$ cd ~/mpj_rundir
$ mpjhalt machines
```

Performance on the ODROID-MC1

The performance of running the Mandelbrot Generator on the ODROID-MC1 in both multicore and cluster mode is summarised in the line graph below. For comparison, I also ran it on a VM with 4 cores assigned to it on an old INTEL I7 quad core machine. Figure 7 is a screenshot of the generator running in the VM.

Figure 7 – Running Mandelbrot Generator in the virtual machine

The performance of running on INTEL is also shown in the same graph. The vertical axis is the time in seconds taken to generate the Mandelbrot at coordinate -0.5, 0.0. The horizontal axis is the number of cores used.

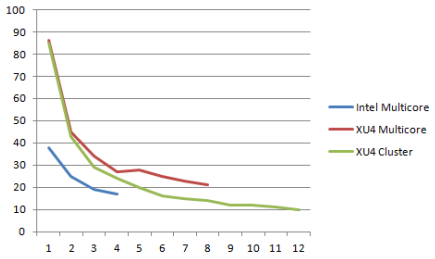


Figure 8 – Graph of the execution time of the Mandelbrot Generator as the number of cores increases

Graphing the data differently gives the performance increase factor as the number of cores increases.

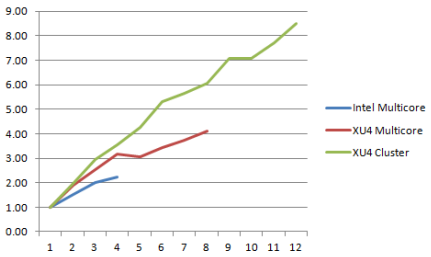


Figure 9 – Graph of performance increase of the Mandelbrot Generator as the number of cores increases

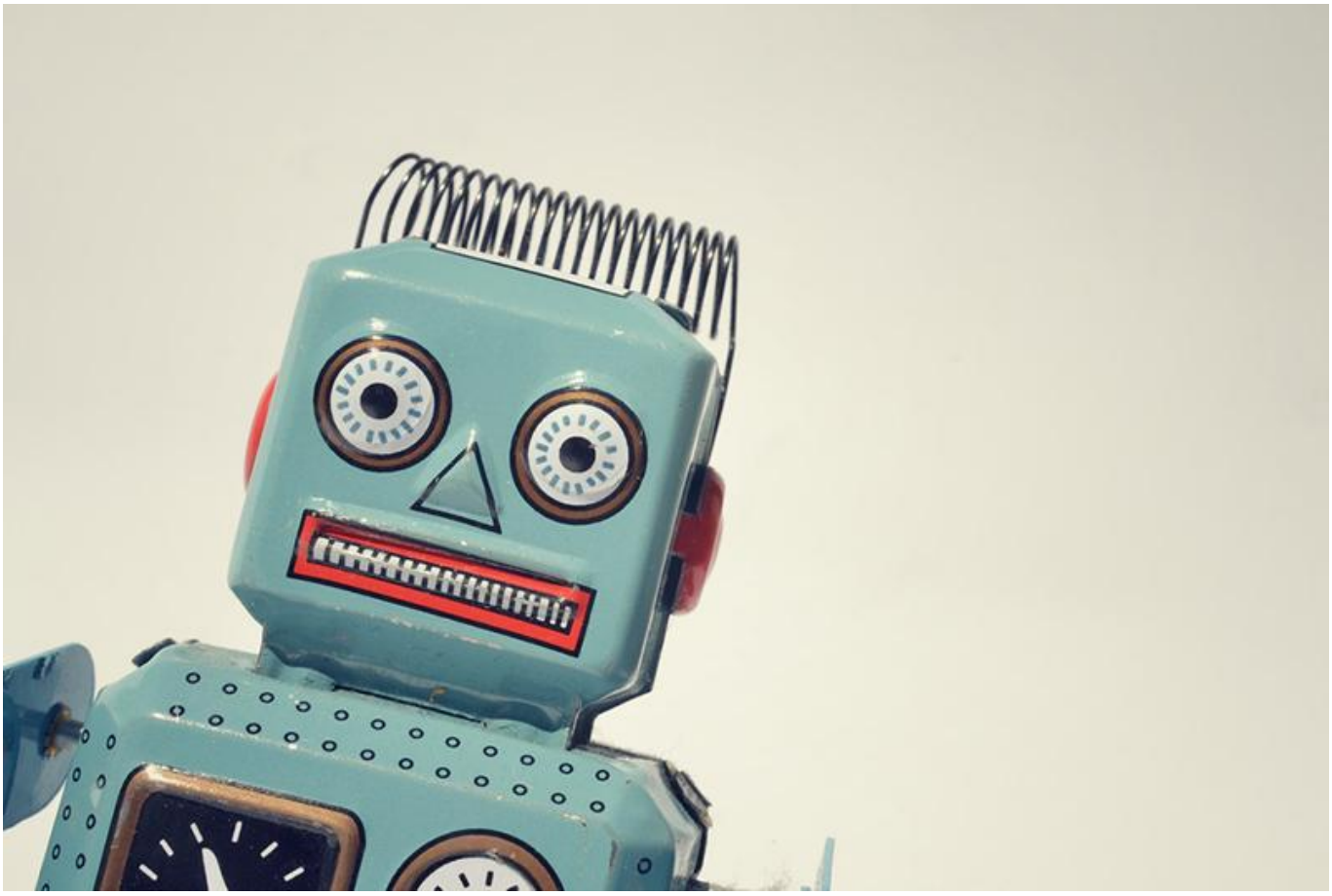
I observed that when a node was using all 4 big cores or all 8 cores, the current used was between 2.0 and 2.5 amps. My cheap power supply was not able to supply enough current when all 4 XU4s on the ODROID-MC1 were running all cores at 100% utilization. This is the reason why I only measured the performance for the cluster configuration up to 12 cores. Another interesting observation was that in multicore mode on a single XU4, the most gain occurred when all 4 big cores were being used. Adding the little cores did not improve performance by that much. Even for cluster mode, the performance gain tapered off as the number of cores increased due to Amdahl's law as the master had to spend the same amount of time combining the generated partial images into a complete image and it took a finite amount of time to transfer the partial images via the network.

Conclusion

I hope my two getting started guides in the ODROID Magazine have given you some ideas of using your ODROID-MC1 as a Docker swarm cluster and also as a Compute cluster for parallel programming. What you can do with it is limited only by your imagination. Let us know if you are interested in additional information regarding using MPJ Express. We can create additional tutorials. In the meantime, enjoy and keep exploring the capabilities of your ODROID-MC1.

Home Assistant: Scripts for Customization

October 1, 2017 By Tutorial



In this article, we will delve deeper still into Home Assistant customization, creating our own scripts to collect data from remote sensors and other control devices. We will also look into various ways to communicate with the remote sensors.

Getting remote temperature data

Let's assume you have this problem: you have several temperature sensors such as the DS1820 around your house connected to various ODROIDs and you want to send the data to Home Assistant, which runs on a single device. You'll need to decide on a data transport protocol and write some script to collect the temperature readings and pass it to Home Assistant.

Let's analyze some approaches:

- Polling over HTTP
- Pushing over Home Assistant API
- Pushing over MQTT

Polling over HTTP

If you're used to web development, you're probably used to CGI (Common Gateway Interface), the oldest way to generate dynamic content using a web server (http://bit.ly/2jNVkjT). Basically, you upload a script on the server, regardless of language, which is called by the web server, serving the script's output back to the client. Obviously, you first need to install an HTTP server on your remote host and activate CGI support. We'll use Apache 2.4:

```
$ sudo apt-get install apache2
$ sudo a2enmod cgi
```

The default configuration maps the /cgi-bin/ URL to /usr/lib/cgi-bin on your file system. Any executable scripts you place here can be called by the web server. Let's assume that you can get the temperature data on the remote host with these shell commands:

```
$ cat /sys/devices/w1_bus_master1/28-05168661eaff/w1_slave
c6 01 4b 46 7f ff 0c 10 bd : crc=bd YES
c6 01 4b 46 7f ff 0c 10 bd t=28375
```

In the output above, the first line validates the reading of the value (if the CRC matches), and the second line returns the value in milli-celsius. We will create two scripts (don't forget to mark them as executable) to illustrate the code in two different languages: BASH and Python. The files will be stored in /usr/lib/cgi-bin/temperature.sh and /usr/lib/cgi-bin/temperature.py.

```
#!/bin/bash

filename='/sys/devices/w1_bus_master1/28-05168661eaff/w1_slave'
valid=0

echo "Content-Type: text/plain"
echo

# read line by line, parse each line
while read -r line
do
    if [[ $line =~ crc=.*YES ]]; then
        # the CRC is valid. Continue processing
        valid=1
        continue
    fi
    if [[ "$valid" == "1" ]] && [[ $line =~ t=[0-9]+ ]]; then
        # extract the temperature value
        rawtemperature=`echo "$line" | cut -d "=" -f 2`
        # convert to degrees celsius and keep 1 digit of accuracy
        echo "scale=1;$rawtemperature/1000" | bc
    fi
#read line by line from $filename
done < "$filename"
```

```
#!/bin/bash


filename='/sys/devices/w1_bus_master1/28-05168661eaff/w1_slave'
valid=0

echo "Content-Type: text/plain"
echo

# read line by line, parse each line
while read -r line
do

  if [[ $line =~ crc=.*YES ]]; then
    # the CRC is valid. Continue processing
    valid=1
    continue
  fi
  if [[ "$valid" == "1" ]] && [[ $line =~ t=[0-9]+ ]]; then
    # extract the temperature value
    rawtemperature=`echo "$line" | cut -d "=" -f 2`
    # convert to degrees celsius and keep 1 digit of accuracy
    echo "scale=1;$rawtemperature/1000" | bc
  fi
#read line by line from $filename
done < "$filename"
```

Figure 1a There are two ways of reading the same temperature, here in bash

```
#!/usr/bin/python
import re

filename = '/sys/devices/w1_bus_master1/28-05168661eaff/w1_slave'
valid = False

print "Content-Type: text/plain"
print ""

# execute the command and parse each line of output
with open(filename) as f:
    for line in f:

        if re.search('crc=.*YES', line):
            # the CRC is valid. Continue processing
            valid = True
            continue

        if valid and re.search('t=[0-9]+', line):
            # extract the temperature value
            temperature = re.search('t=([0-9]+)', line)
            # convert to degrees celsius and keep 1 digit of accuracy
            output = "%.1f" % (float(temperature.group(1))/1000.0)
            print output
```

Figure 1b - And here in Python

Let's analyze the scripts a bit. **Both scripts** start with a shebang line which tells the caller which interpreter to use to run the script (line 1). Next, we define two variables to point to the file to be read (line 4) and a variable to remember if the reading is valid or not (line 5). On lines 7 and 8 we print the HTTP headers. The CGI script has to return HTTP headers on the first lines, separated by a blank line from the rest of the output. The web server needs at least the Content-Type header to process the request. If you omit this, you will get an HTTP 500 error. On line 11 we begin reading the lines from the file in order to parse each one. We look for a valid CRC with a regular expression on line 14, and if it is correct, we set valid to true. On line 19, if the CRC is true and the line contains a temperature, we extract the raw temperature (line 21) and convert it to celsius, with one digit of accuracy (line 23), and print it to standard output. In order to access the data, you could use any HTTP client, like wget, as shown in Figure 2.

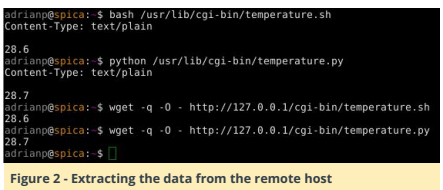


Figure 2 - Extracting the data from the remote host

There might be slight differences in the output returned because of different rounding methods used, or by variations in the time the query is made, which can cause the sensor data to fluctuate.

For security purposes, you can enable HTTP Basic Authentication in your server's config. You'll need SSL/HTTPS with valid certificates in order to protect yourself from somebody sniffing your traffic, but that goes beyond the scope of this article. You can read more about those **here** and **here**.

In order to add the sensor to Home Assistant we can use the REST sensor inside **configuration.yaml** :

```
sensor:
 ...
 - platform: rest
   resource: http://192.168.1.13/cgi-bin/temperature.sh
   name: Temperature REST Bash
   unit_of_measurement: C
 - platform: rest
   resource: http://192.168.1.13/cgi-bin/temperature.py
   name: Temperature REST Python
  unit_of_measurement: C
```

You can get the code **here** and **here**.

Pros for this method:

- It's easy to implement if you've done web development
- On Home Assistant restart new data is polled

Cons for this method:

- Using a web server exposes you to possible vulnerabilities
- The web server may use a lot of resources in comparison to what it needs to do

Pushing over HA API

A different approach that doesn't involve a web server is to push sensor data to Home Assistant from the remote system. We can use a **Template Sensor** to hold and present the data. In order to do this, you can have the script in Figure 3 called periodically with cron on the remote system.

```
#!/bin/bash

filename='/sys/devices/w1_bus_master1/28-05168661eaff/w1_slave'
homeassistantip='192.168.1.9'
haport=8123
api_password='odroid'
sensor_name='sensor.temperature_via_api'
valid=0

# read line by line, parse each line
while read -r line
do

    if [[ $line =~ crc=.*YES ]]; then
        # the CRC is valid. Continue processing
        valid=1
        continue
    fi
    if [[ "$valid" == "1" ]] && [[ $line =~ t=[0-9]+ ]]; then
        # extract the temperature value
        rawtemperature=`echo "$line" | cut -d "=" -f 2`
        # convert to degrees celsius and keep 1 digit of accuracy
        temperature=`echo "scale=1;$rawtemperature/1000" | bc`
        # push the data to the Home Assistant entity via the API
        curl -X POST -H "x-ha-access: $api_password" -H "Content-Type: application/json"
        --data "{"state": "$temperature"}"
http://$homeassistantip:$haport/api/states/$sensor_name
    fi
#read line by line from $filename
done < "$filename"
```

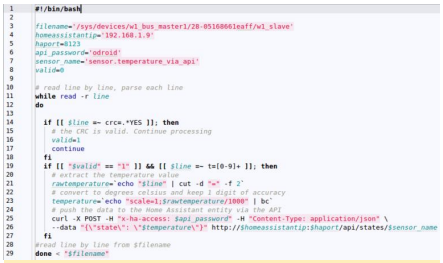


Figure 3 - Pushing data via the HA API

As you can see, the code is similar to the previous example, except that at line 25 it uses **Home Assistant REST API** to submit the temperature reading. The REST API requires you to send the Home Assistant API Key inside of a HTTP header, and the data you want changed needs to be in a JSON payload in the POST request. The URL you post to is your Home Assistant instance /api/states/sensor.name. To enable this and submit data every 5 minutes, add the following cron entry:

```
$ crontab -e
*/5 * * * * /bin/bash /path/to/script/temperature-HA-API.sh > /dev/null 2>&1
```

The Home Assistant configuration looks like this:

```
sensor:
…
- platform: template
   sensors:
     temperature_via_api:
       value_template: '{{ states.sensor.temperature_via_api.state }}'
       friendly_name: Temperature via API
       unit_of_measurement: C
```

The template sensor is usually used to extract data from other Home Assistant entities, and in this case we use it to extract data from itself. This trick prevents it from deleting the state data after an external update. Before you set the temperature, the sensor state will be blank. After cron executes the script the first time, you will get temperature data. You can get the code from **here**

Pros for this method:

- You control when data is pushed
- Resource use is very low

Cons for this method:

- Your script needs to have your Home Assistant secret password in clear

- When Home Assistant is restarted, the sensor will not have any value until the first update

Pushing over MQTT

The MQTT protocol is a machine to machine protocol designed for efficiency (and low power environments) and has been discussed already in previous ODROID Magazine articles. The way it works is that a central server called a broker relays messages for clients that subscribe to a common topic. Think of a topic as something like an IRC channel where clients connect and send each other specific messages.

Home Assistant has a built-in MQTT Broker, but in my tests I found it unreliable, so I used a dedicated broker called Mosquitto. It can be installed on the same system as Home Assistant, or on a different system. To install it, follow these steps:

```
$ sudo apt-get install mosquitto mosquitto-clients
$ sudo systemctl enable mosquitto
```

MQTT version 3.11 supports authentication, so you should set up a username and password that is shared by broker and clients and, optionally, SSL encryption. In my setup I used user-password authentication, and added an 'ODROID' user:

```
$ sudo mosquitto_passwd -c /etc/mosquitto/passwd ODROID
$ sudo vi /etc/mosquitto/conf.d/default.conf
allow_anonymous false
password_file /etc/mosquitto/passwd
```

You can enable general MQTT support in Home Assistant by adding a MQTT platform in configuration.yaml (remember that the mqtt_password parameter is defined in secrets.yaml instead):

```
mqtt:
  broker: 127.0.0.1
  port: 1883
  client_id: home-assistant
  keepalive: 60
  username: ODROID
  password: !secret mqtt_password
```

In order to push temperature data to Home Assistant our script will need the Paho-MQTT Python library. In order to parse configuration data we'll need the python-yaml library as well:

```
$ sudo apt-get install python-pip python-yaml
$ sudo pip install paho-mqtt
```

The script runs as a daemon, performing periodic temperature readings in the background and sending changes via MQTT. The code which reads the actual temperature (line 40) is the same as in Figure 1b and is not shown in Figure 4 for brevity. The only change is that instead of printing the temperature, it returns it as a string.

The code begins by importing a few helper modules, defining functions to parse the YAML configuration into a dictionary. Reading the temperature and execution begins at line 57. A new MQTT client object is defined and initialized with the necessary details to access the MQTT broker. On line 61, there is a background thread started by the loop_start() call which ensures that the client remains connected to the MQTT broker. Without it, the connection would time out and you would need to reconnect manually. More information about the MQTT API in Python is available here. On line 65, there is a loop that reads temperature data, compares it with the last temperature read, and if there is a change, publishes an MQTT message to the broker with the new temperature. Then the code sleeps for a while before the next reading. When publishing data to the broker (on line 71), you need to specify the MQTT topic, the value being sent, and also if this data should be persistent or not. Persistent data is convenient, because you can get the last temperature reading from MQTT when you start Home Assistant and read the temperature for the first time. You can get the full code from here.

```
#!/usr/bin/python
import paho.mqtt.client as mqtt
import re
import time
import sys
import yaml

# Prerequisites:
# * pip: sudo apt-get install python-pip
# * paho-mqtt: pip install paho-mqtt
# * python-yaml: sudo apt-get install python-yaml

# Configuration file goes in /etc/temperature-mqtt-agent.yaml and should contain your mqtt broker details

# For startup copy temperature-mqtt-agent.service to /etc/systemd/system/
# Startup is done via systemd with
#  sudo systemctl enable temperature-mqtt-agent
#  sudo systemctl start temperature-mqtt-agent

filename = '/sys/devices/w1_bus_master1/28-05168661eaff/w1_slave'
valid = False
oldValue = 0

""" Parse and load the configuration file to get MQTT credentials """

conf = {}

def parseConfig():
    global conf
    with open("/etc/temperature-mqtt-agent.yaml", 'r') as stream:
        try:
            conf = yaml.load(stream)
        except yaml.YAMLError as exc:
            print(exc)
            print("Unable to parse configuration file /etc/temperature-mqtt-agent.yaml")
            sys.exit(1)

""" Read temperature from sysfs and return it as a string """

    def readTemperature():
        with open(filename) as f:
            for line in f:
                if re.search('crc=.*YES', line):
                    # the CRC is valid. Continue processing
                    valid = True
                    continue
                if valid and re.search('t=[0-9]+', line):
                    # extract the temperature value
                    temperature = re.search('t=([0-9]+)', line)
                    # convert to degrees celsius and keep 1 digit of accuracy
                    output = "%.1f" % (float(temperature.group(1)) / 1000.0)
                    # print("Temperature is "+str(output))
                    return output

""" Initialize the MQTT object and connect to the server """
parseConfig()
client = mqtt.Client()
if conf['mqttUser'] and conf['mqttPass']:

client.username_pw_set(username=conf['mqttUser'], password=conf['mqttPass'])
    client.connect(conf['mqttServer'], conf['mqttPort'], 60)
    client.loop_start()

""" Do an infinite loop reading temperatures and sending them via MQTT """

    while (True):
        newValue = readTemperature()
        # publish the output value via MQTT if the value has changed
        if oldValue != newValue:
            print("Temperature changed from %f to %f" % (float(oldValue), float(newValue)))
            sys.stdout.flush()

client.publish(conf['mqttTopic'], newValue, 0, conf['mqttPersistent'])
            oldValue = newValue
        # sleep for a while
        # print("Sleeping...")
        time.sleep(conf['sleep'])
```

Figure 4 - Sending temperature data via MQTT

The script will also need a configuration file where it keeps MQTT credentials, located at /etc/temperature-mqtt-agent.yaml:

```
mqttServer: 192.168.1.9
mqttPort: 1883
mqttUser: ODROID
mqttPass: ODROID
mqttTopic: ha/kids_room/temperature
mqttPersistent: True
sleep: 10
```

There's also a systemd startup script to start your script on every boot. Copy it to /etc/systemd/system:

```
$ cat /etc/systemd/system/temperature-mqtt-agent.service
[Unit]
Description=Temperature MQTT Agent
After=network.target
[Service]
ExecStart=/usr/local/bin/temperature-mqtt-agent.py
Type=simple
Restart=always
RestartSec=5<
[Install]
WantedBy=multi-user.target
```

To enable it at startup, run the following commands:

```
$ sudo systemctl enable temperature-mqtt-agent.service
$ sudo systemctl start temperature-mqtt-agent.service
```

On the Home Assistant side of things, we need to define an **MQTT sensor** with the following configuration:

```
sensor:
...
 - platform: mqtt
    state_topic: 'ha/kids_room/temperature'
    name: 'Temperature via MQTT'
    unit_of_measurement: C
```

Pros for this method:

- Resource use is low
- Standard API with low overhead designed for machine-to-machine communication

Cons for this method:

- The remote system needs to have the MQTT password in the clear
- When Home Assistant is restarted, the sensor will not have any value until the first update unless the MQTT Persistence option is used

Now that you've seen several examples of getting data into Home Assistant, you will have to choose what is best for your setup. From now on I will go with MQTT because, even if it seems more difficult in the beginning, it scales better with more complex tasks.

Controlling a Smart TV with a custom component

Here's a new problem that we want to solve. Let's collect the current channel number, program name, and TV state from a Samsung TV running **SamyGO firmware**. The TV exposes this information via a REST API which can be installed on the TV from **here**. The API sends back information in JSON format about the current state of the TV. It can inject remote control codes and can also send back screenshots with what's currently on. The call and results for the current information look like this:

```
$ wget -O - "http://tv-ip:1080/cgi-bin/samygo-web-api.cgi?challenge=oyd4uIz5WWAkWPo5MzfxBFraI05C3FDorSPE7xiMLCVAQ40a&action=CHANNELINFO"
```

{"source":"TV (0)", "pvr_status":"NONE", "powerstate":"Normal", "tv_mode":"Cable (1)", "volume":"9", "channel_number":"45", "channel_name":"Nat Geo HD", "program_name":"Disaster planet", "resolution":"1920x1080", "error":false}

In theory, we could configure REST sensors to make the query above and use templating to preserve only the desired information, like this:

```
sensor:
...
  - platform: rest
  resource: http://tv-ip:1080/cgi-bin/samygo-web-api.cgi?challenge=oyd4uIz5WWAkWPo5MzfxBFraI05C3FDorSPE7xiMLCVAQ40a&action=CHANNELINFO
  method: GET
  value_template: '{{ value_json.channel_name }}'
  name: TV Channel Name
```

But the problem is that, in order to get all the information in different sensors, you need to make the same query, discard a lot of data, and keep only what you need for that particular sensor. This is inefficient, and in this case, it won't work because, in order to obtain and expose this information, the web API running on the TV injects various libraries into running processes on the TV to hijack some function calls and obtain the data **here**. The injection step is critical, and doing multiple injections at the same time could cause the process to crash, which would lock up your TV. This is why the web API serializes the queries and won't respond to a query before the previous one is done, but this could result in timeouts.

What is needed in this case is for the sensor component to store all of the JSON data and have template sensors to extract the needed data and present it. In order to do this, we need a custom component, derived from the REST sensor which acts just like the REST sensor, but when it receives JSON data it stores that data as attributes of the entity instead of discarding them.

Custom components live in the ~homeassistant/.homeassistant/custom_components directory and preserve the structure of regular components (meaning our sensor would live in the sensor subdirectory). They are loaded at Home Assistant startup before configuration is parsed. Figure 5 shows the differences between the REST sensor and the new custom JsonRest sensor.

```
--- /srv/homeassistant/lib64/python3.5/site-packages/homeassistant/components/sensor/rest.py
+++ /home/homeassistant/.homeassistant/custom_components/sensor/jsonrest.py    2017-06-29 15:5
@@ -1,2 +1,5 @@
-"""
-Support for RESTful API sensors.
+"""
+Support for RESTful API sensors.
+
+Modified to parse a JSON reply and store data as attributes
 """
@@ -7,7 +10,7 @@
 import voluptuous as vol
 import json
 import requests
-from homeassistant.components.sensor import PLATFORM_SCHEMA
+from homeassistant.components.sensor import PLATFORM_SCHEMA, DOMAIN
 from homeassistant.const import (CONF_NAME, CONF_METHOD, CONF_PAYLOAD,
                                  CONF_RESOURCE, CONF_VALUE_TEMPLATE,
                                  CONF_VERIFY_SSL, CONF_USERNAME,
@@ -24,7 +27,7 @@
 DEFAULT_METHOD = 'GET'
 DEFAULT_NAME = 'REST Sensor'
 DEFAULT_NAME = 'JSON REST Sensor'
 DEFAULT_VERIFY_SSL = True
@@ -65,3 +68,3 @@
             auth = None
-        rest = RestData(method, resource, auth, headers, payload, verify_ssl)
+        rest = JSONRestData(method, resource, auth, headers, payload, verify_ssl)
         rest.update()
@@ -72,6 +75,6 @@

-        add_devices([RestSensor(hass, rest, name, unit, value_template)])
+        add_devices([JSONRestSensor(hass, rest, name, unit, value_template)])

-class RestSensor(Entity):
+class JSONRestSensor(Entity):
     """Implementation of a REST sensor."""
@@ -83,2 +86,3 @@
         self._name = name
+        self._attributes = {}
         self._state = STATE_UNKNOWN
@@ -102,3 +106,3 @@
         return self._state

-    def update(self):
+    def update(self):
@@ -115,9 +119,22 @@
         self._state = value

+        """ Parse the return text as JSON and save the json as an attribute. """
+        try:
+            self._attributes = json.loads(value)
+        except json.JSONDecodeError:
+            self._attributes = {}
+            pass

+    @property
+    def state_attributes(self):
+        """Return the attributes of the entity.

+        Provide the parsed JSON data (if any).
+        """
+        return self._attributes


-class RestData(object):
+class JSONRestData(object):
     """Class for handling the data retrieval."""
```

Figure 5 - Changes to store and expose attributes

In order to understand the changes made, you should follow the custom components guide http://bit.ly/2fvc1PT. The code makes some name changes in the module's classes to prevent collisions with the REST component, and initializes and manages a list of attributes that are parsed from the JSON input. These will show up as attributes in the States view. The new component name is JsonRest, the same as the filename.

To install the JsonRest component, you can follow these steps:

```
mkdir -p ~homeassistant/.homeassistant/custom_components/sensor/
wget -O ~homeassistant/.homeassistant/custom_components/sensor/jsonrest.py https://raw.githubusercontent.com/mad-ady/home-assistant-customizations/master/custom_components/sensor/jsonrest.py
```

To configure the new component, once it's stored in the custom_components/sensor directory, we can use this configuration to poll the TV every 5 minutes:

```
sensor:
…
 - platform: jsonrest
    resource: http://tv-ip:1080/cgi-bin/samygo-web-api.cgi?challenge=oyd4uIz5WWAkWPo5MzfxBFraI05C3FDorSPE7xiMLCVAQ40a&action=CHANNELINFO
    method: GET
    name: TV Living ChannelInfo
    scan_interval: '00:05'
 - platform: template
    sensors:
     tv_living_powerstate:
        value_template: '{{ states.sensor.tv_living_channelinfo.attributes.power_state }}'
        friendly_name: TV Living Power
     tv_living_channel_number:
        value_template: '{{ states.sensor.tv_living_channelinfo.attributes.channel_number }}'
        friendly_name: TV Living Channel Number
      tv_living_channel_name:
        value_template: '{{ states.sensor.tv_living_channelinfo.attributes.channel_name }}'
        friendly_name: TV Living Channel Name
      tv_living_program_name:
        value_template: '{{ states.sensor.tv_living_channelinfo.attributes.program_name }}'
        friendly_name: TV Living Program Name
```

Now only the JsonRest component will poll the TV for information, and the template sensors extract the needed data from the attributes, reducing the load on the TV.

Since the TV web API allows the capture of screenshots, let's add that to Home Assistant as well (to keep an eye on what the kids are watching). The API returns a JPEG image each time you ask with URL parameter action=SNAPSHOT. You can use a **Generic IP Camera component**:

```
camera 2:
  platform: generic
  name: TV Living Image
  still_image_url: http://tv-ip:1080/cgi-bin/samygo-web-api.cgi?challenge=oyd4uIz5WWAkWPo5MzfxBFraI05C3FDorSPE7xiMLCVAQ40a&action=SNAPSHOT
```

The TV web API also allows you to send remote control actions, which can be modelled through the **Restful Command component**:

```
rest_command:
  tv_living_power_on:
    url: !secret samygo_tv_living_power_on
  tv_living_power_off:
    url: !secret samygo_tv_living_power_off
```

After a bit of grouping, the polished end result may be viewed **here**. A link to the configuration is available **here**, and an example for the secrets file is **here**. You can find the code and configuration on the **GitHub page**.

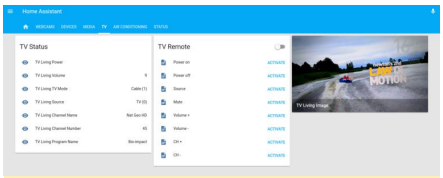
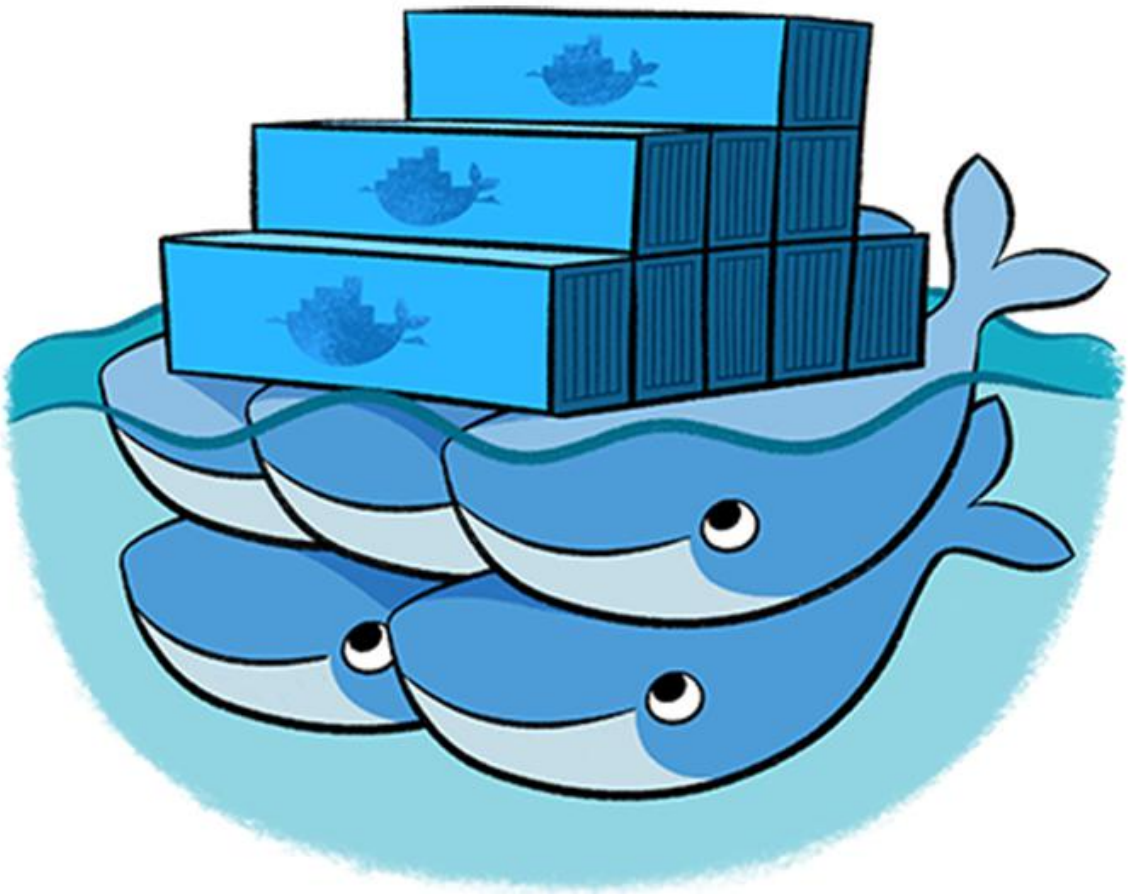


Figure 6 - Keeping an eye on the TV

ODROID-MC1 Docker Swarm: Getting Started Guide

October 1, 2017 By Andy Yuen Docker, Tutorial, ODROID-MC1



The staff at Hardkernel built a big cluster computing setup for testing the stability of Kernel 4.9. The cluster consisted of 200 ODROID-XU4's (i.e, with a net total of 1600 CPU cores and 400GB of RAM), as shown in Figure 1.

Figure 1 – A cluster of 200 ODROID-XU4 devices

The experience obtained with this exercise led them to the idea of building an affordable and yet powerful personal cluster, out of which was born the ODROID-MC1. ODROID-MC1 stands for My Cluster One. It consists of 4 stackable units, each with a specially designed Single Board Computer (SBC) based on the Samsung Exynos 5422 octa-core processor. It is compatible with the ODROID-XU4 series SBC, and is mounted on an aluminum case. These cases (which also incorporates an integrated heatsink) are stacked with a fan attached on the back-end, to ensure adequate cooling.

The ODROID-MC1 circuit board is a trimmed version of that used in the ODROID-HC1 (Home Cloud One) Network Attached Storage (NAS), with the SATA adapter removed. The ODROID-HC1 circuit board, in turn, is a redesigned ODROID-XU4 with the HDMI connector, eMMC connector, USB 3.0 hub, power button and, slide switch removed.

Key features of the ODROID-MC1 include:

- Samsung Exynos 5422 Cortex-A15 2Ghz and Cortex-A7 Octa core CPUs
- 2Gbyte LPDDR3 RAM PoP stacked
- Gigabit Ethernet port
- USB 2.0 Host
- UHS-1 micro-SD card slot for boot media
- Linux server OS images based on modern Kernel 4.9 LTS

The ODROID-MC1 comes assembled and ready to use as a personal cluster for learning as well as for doing useful work. In Part 1 of this series on the ODROID-MC1, I will be describing how to use it as a Docker Swarm cluster. In Part

2, I shall describe how to develop parallel programs to run on the ODROID-MC1 cluster.

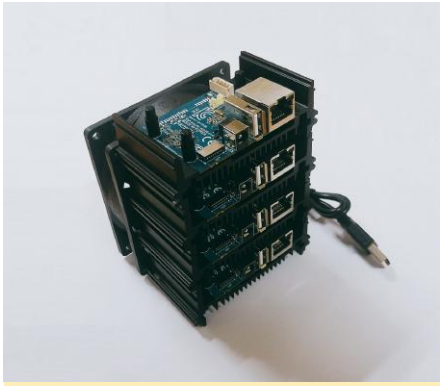


Figure 2 – The ODROID-MC1 makes an excellent swarm device

To set up the MC1 cluster, you need the following in addition to the MC1 hardware:

- 1 x Gigabit switch with at least 5 ports
- 5 x Ethernet cables
- 4 x SD cards (at least 8GB in capacity)
- 4 x power adapters for the MC1 computers

Setting Up the OS on Each Computer on the Cluster

The most tedious part in setting up the ODROID-MC1 cluster is to install an Operating System (OS) and software packages needed for running and managing the docker swarm on each compute node. To expedite the process, you can download an SD card image with everything almost ready to use at https://oph.mdrjr.net/MrDreamBot/. I say "almost",' because there are still a few steps you have to do to make everything work. The SD card has logins 'root' and 'odroid' already set up. The password for both logins is "odroid:.

The swarm we are building consists of 1 master and 3 worker nodes. For discussion purposes, assume they use the following host names and IP addresses. Of course you can change them to suit your environment. All nodes in the swam should have static IP address like so:

```
xu4-master – 192.168.1.80
xu4-node1 – 192.168.1.81
xu4-node1 – 192.168.1.82
xu4-node1 – 192.168.1.83
```

To start the setup process, you need to connect your PC and one ODROID-MC1 node at a time to a Gigabit switch which has a connection to your home router (for access to the Internet). The image is configured to use dynamically allocated IP address using DHCP from your router. You have to login using SSH to configure each node to use a static IP address instead. There are other configuration parameters you need to change as well.

The setup process assumes that you have some Linux command line knowledge to carry out the following steps:

- Write OS image to your SD card – Copy the SD card image to 4 x 8GB Class 10 SD cards. If you use bigger capacity SD cards, you have to resize the filesystem on each SD card to take up all space on your SD card. The easiest way to do this is to mount the SD card on a Linux machine and use gparted (www.gparted.com) to resize it. That is the method I used for my SD cards. Insert an SD card in one of the MC1 computers.
- Initiate an SSH session from your PC to the ODROID-MC1 node as root. Its IP address can be found in your home router. Skip the next step if you are setting up the master node.
- Change the host name by editing the /etc/hostname file, to change xu4-master to xu4-nodeX where X is either 1, 2 or 3 depending on which worker node you are setting up.
- Configure a static IP address by editing the /etc/network/interfaces, by removing the "#" in front of the highlighted section and replacing the IP address 192.168.1.80 with the IP address (in your home network subnet) to which you want to assign the node you are setting up.
- Update the /etc/hosts file such that each ODROID-MC1 node entry has the correct name and IP address.
- Test the changes – Reboot the node to see if you can SSH into it using the new IP address you assigned to it. If so, you have successfully set up that node. If not, double check the changes described above to make sure there are no typos.
- Set up the next worker node – Repeat Steps 2 through 7 until all the nodes have been set up.

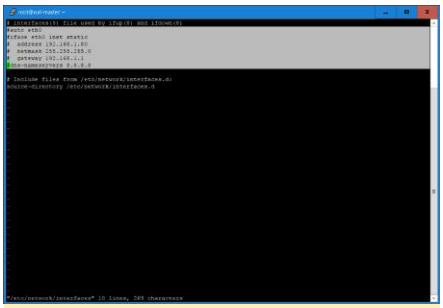


Figure 3 – Listing the Docker Swarm interfaces

For experienced Linux users, an alternate way to do the above is to mount each SD card on your Linux system and edit those files directly on the SD card.

After you have set up your cluster, ssh into xu4-master as user "odroid", password "odroid". From the master, you can SSH to all the worker nodes without using password as the nodes in the cluster have been set up with key-based authentication. Do the same for "root" by either using the "sudo -s" command, or by using SSH to establish a connection as the root user into the xu4-master node, then using SSH to connect to all of the worker nodes.

Setting Up Docker Swarm

A node is a Docker host participating in a swarm. A manager node is where you submit a service definition and it schedules the service to run as tasks on worker nodes. Worker nodes receive and execute tasks scheduled by a manager node. A manager node, by default, is also a worker node unless explicitly configured not to execute tasks. Multiple master and worker nodes can be set up in a swarm to provide High Availability (HA).

To bring up swarm mode, issue the following command on the manager:

```
$ docker swarm init --advertise-addr
192.168.1.80
```

which returns:

```
swarm initialized: current node
(8jw6y313hmt3vfa1fme1dinro) is now a manager
```

Run the following command to add a worker to this swarm on each node:

```
$ docker swarm join --token SWMTKN-1-
1q385ckmw7owbj2zfno04dmidb62iqg2devd7yvae5wv
uohc11-at5g1ad4f24fck4cutsqhnw06
192.168.1.80:2377
```

To make the other nodes join the cluster, issue the "docker swarm join"'command above on each node. This can be done using the parallel-ssh to issue the command once from the manager and executed on each node. Figure 4 is a screenshot of running "docker ps" command using parallel-ssh.

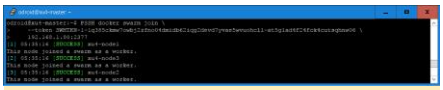


Figure 4 – Running the "docker ps" command using parallel-ssh

Now we have a Docker swarm up and running.

Testing the Swarm

To help visualize what is going on in the swarm, we can use the Docker Swarm Visualizer image (visualizer-arm). To deploy it as a service, issue the following command from the manager command prompt:

```
$ docker service create --name=dsv --
publish=8080:8080/tcp --
constraint=node.role==manager --
mount=type=bind,src=/var/run/docker.sock,dst
=/var/run/docker.sock alexellis2/visualizer-
arm
```

Note that the ODROID-XU4 is ARMv7-based, i.e., it is a 32 bit system, unlike the ODROID-C2 which is ARMv8-based, and 64 bit. Consequently, the Docker images used in the following commands are different from those used in my Docker examples for the ODROID-C2.

Point your browser at the master by visiting http://192.168.1.80:8080, or you can point your browser to any of the nodes in the swarm. Observe the changes reported by the visualizer when deploying the httpd service using my 32 bit httpd busybox mdreambot image at http://dockr.ly/2wWPCNP. My image is started using the command:

```
$ docker service create --replicas 3 --name
httpd -p 80:80 mdreambot/arm32-busybox-httpd
```

Figure 5 shows a Docker Swarm Visualizer displaying the nodes on which the service replicas are run, illustrating the declarative service model used by swarm mode.

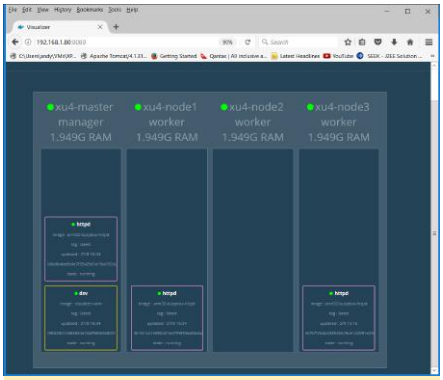


Figure 5 – Docker Swarm Visualizer shows the nodes on which the service replicas are run

Use the following curl command to test the load balancing feature of docker swarm:

```
$ curl http://192.168.1.80/cgi-bin/lbtest
```

Figure 6 is a screenshot of the curl commands output, which reconfirms that each request has been directed to a different node.

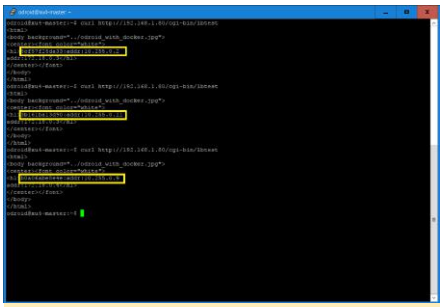


Figure 6 – Docker performs automatic load balancing

For a test of self-healing, I stopped the running httpd container on xu4-master, and another httpd container was spun up on another node to replace the one I just stopped as can be seen in the screenshot below. This is because when we started the service, we specified "replica=3" and the Docker swarm will maintain the desired number of replicas. This is called desired state reconciliation.

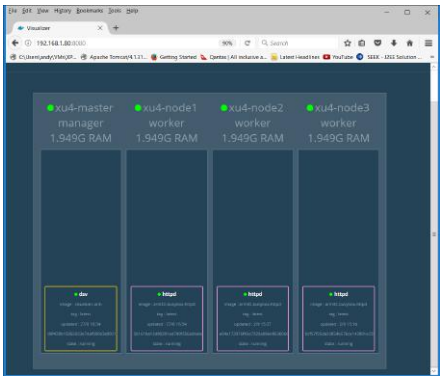


Figure 7 – Docker supports self-healing state reconciliation

Conclusion

The Docker swarm mode is now fully functional on your ODROID-MC1 cluster. Feel free to experiment with it. I hope this guide achieves its objective in getting you started running docker swarm on the ODROID-MC1. For more information on Docker swarm mode, please refer to my other ODROID Magazine articles on the subject.

Meet An ODROIDian: Brian Kim, Hardkernel Engineer

October 1, 2017 By Brian Kim Meet an ODROIDian

Figure 1 – Brian Kim at Google headquarters in California

Figure 2 – Brian Kim's family: Younger Sister, Mother, Father, Older Sister and Nephews

Please tell us a little about yourself.

I'm 36 years old and live in Seoul, South Korea. I'm the Research Engineer of Hardkernel Co., Ltd. My main job for Hardkernel is maintaining the open source software like u-boot, Linux Kernel, WiringPi2-Python and Buildroot. I modify the open source software and add some routines in order to support ODROIDs. Hardkernel provides technical support in the ODROID forums at http://forum.odroid.com, and my boss will assign software issues reported in the forums to me. Although it is sometimes stressful when it is a complex issue or not a software issue, it is enjoyable to have technical discussions with the ODROID users.

I have not only a Bachelor's degree in Information and Communication from Youngsan University (South Korea), but also a Master's degree in Mobile Communication Engineering from Sungkyunkwan University (South Korea). I was not a good student when I was in high school, but I studied very hard in college and achieved a 4.5/4.5 GPA in one semester. I learned overall background knowledge of Computer Science in college and graduate school. I studied CMT (Concurrent Multipath Transfer) using SCTP in a master's degree and wrote a paper about it.

I started my career as a software developer. My first serious project was the medical information system software using Delphi in 2005. Our team designed medical database associate with personal information database, and we wrote Object Pascal source code for the software. I enjoyed developing software using various programming languages and libraries such as Visual Basic, MFC, Win32, Qt, PHP, ASP, JSP, C, C++ and Java.

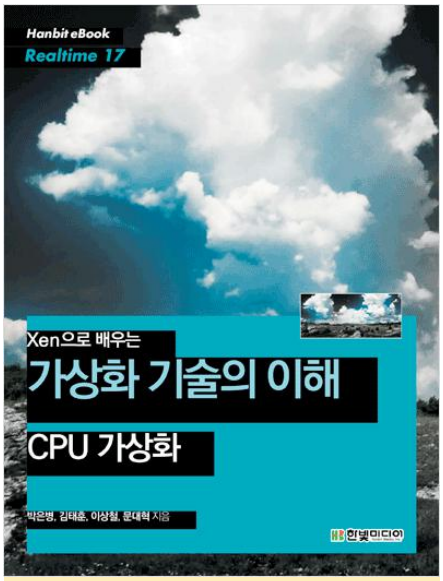


Figure 3 – One of Brian's toy projects, Only Debugger

Even after graduating from college, I still had a thirst for learning. So, I moved to Seoul in order to take Embedded Professional course at a private academy. I learned Embedded system, ARM architecture, Linux kernel, Network programming and RTOS in this course work from 2005 to 2006. I was very influenced by Unix philosophy at that time. After I finished the course, I developed POS (Point Of Sale) and IP set-top box software as a part-time job in 2007. I joined WIZnet as my first full-time job in 2008.

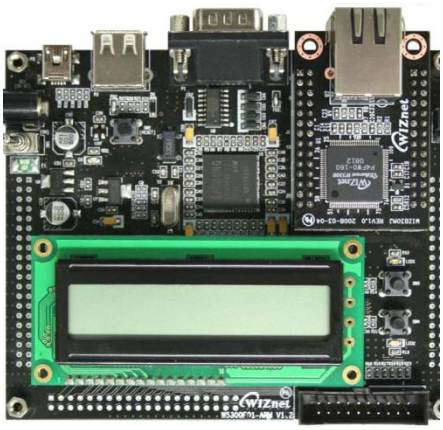


Figure 4 – Brian's first commercial product, W5300E01-ARM

WIZnet is a fabulous company that designs network chips embedded with hardwired TCP/IP stack. My first job in WIZnet is developing Linux network driver for WIZnet chipset. I worked hard and finished the project in three months. After that, I developed ARM embedded board included WIZnet chip called W5300E01-ARM, which was my first commercial product. The modified network driver I developed is included in the mainline Linux kernel source code. Besides that, I developed a Serial-to-Ethernet gateway module and gave technical support.

I participated in an open source software analysis study group every Saturday in 2007 (Linux Kernel) and 2011 (Xen Hypervisor). Our study group analyzed the source code in detail until it was fully understood, which we were passionate about. After finishing the study about a year, we wrote articles and books about what we learned. The source code of open source software is my textbook, and open source software developers are my teachers even now.

Figure 5 – Brian's first computer, an IBM XT

How did you get started with computers?

When I was 8 year old, I got started with computers with an IBM XT. Although it was my cousin's computer, I frequently used the computer to play DOS games. I remember the old gossip at that time, which was that "640KB is enough." When I was 10, my father gave me a 386 PC as a birthday present, and I started PC communications using a 2400bps modem.

Whom do you admire in the world of technology?

Linus Torvalds, since he made the Linux kernel and Git. Linux and Git changed the software world.

How did you decide to work at Hardkernel?

The most important factor was what I will do for Hardkernel. The responsibilities in the job post seemed interesting to me.

How do you use your ODROIDs at home?

I enjoy making interesting things with ODROIDs. Some of my projects can be found in ODROID magazine, such as Ambilight, Rear View Camera and ODROID Arcade Box. In the South Korean office, we use ODROIDs as private servers and automatic fish feeders. The cryptocurrency miner using 200 ODROID-XU4 devices was also an interesting project. I created and used a voice light switch using ODROID-C2 and Google Cloud Speech API at home.

Which ODROID is your favorite and why?

The ODROID-C2 is my favorite, because I'm one of ODROID-C2 main developers, and it has 64bit ARM architecture. Although I'm maintaining all of the ODROIDS currently on sale (ODROID-C1+, ODROID-C2 and ODROID-XU4), I joined after the XU4 and C1+ were developed.

What innovations do you see in future Hardkernel products?

I think that keeping our current position in the SBC market as high-performance devices, and trying to enter the low-end server market are good for survival.

What hobbies and interests do you have apart from computer?

I enjoy travel, computer games, snowboarding, wakeboarding, scuba diving and triathlon (swimming, cycling and marathon). I completed a Triathlon Olympic course last year. I went to Busan from Seoul by bike during the summer vacation this year. The distance is about 325 miles (523 KM). I will challenge myself with a full course marathon next month.

Figure 6 – Brian's hobbies are snowboarding, scuba diving, triathlon, traveling and cycling

What advice do you have for someone wanting to learn more about programming?

Read a good source code from open source software. Write lots of high quality code as much as you can.