Installation of NSCLDAQ for a containerized environment:

Summary of steps:

1. Install the singularity container system. On debian-like systems this is the package `singularity-container`

2. Obtain one of our debian containers. At present we have container images for Debian jessie and Debian buster.

3. Obtain and unwrap the tarball for an appropriate /usr/opt directory tree.

4. Create a start script for your container.

5. Test by running your start script.

Install the singularity container system.

How you do this depends on the system you are using. You may need to find the name of the package for your distribution or install from sources if a package does not exist (see https://sylabs.io/guides/3.0/user-guide/installation.html for installation details). For debian/ubuntu e.g. systems:

sudo apt-get install -y singularity-container

Will install the singularity software. Current versions of debian include singularity-container in their standard package repositories. Note that you may need to add the NeuroDebian repository to your system's repository list to make singularity-container visible and available. See https://neuro.debian.net/ and the section "Get NeuroDebian"

Obtain one of our debian containers.

We distribute container images using hub.docker.com. Singularity can then build a singularity image from a docker image on hub.docker.com. First let's create a directory in which all this stuff will live using mkdir. Then all of the commands in this section and the following section will be executed with your working directory set to that new empty directory.

The general form of the singularity command to build a container image from a docker hub reference is of the form:

singularity build image-name docker://owner/repository:version

The owner of our repositories is fribdaq. The repositories you'll want are frib-buster (Debian buster) and frib-jessie (Debian jessie). You'll want the most recent version of the desired image. Suppose

you want to build a buster image point a web browser at

https://hub.docker.com/r/fribdaq/frib-buster Click on "Tags" to view the versions. The top tag will be the version you want. Suppose that's V2.5 build a buster container by typing (in that new empty directory):

```
singularity build buster.img docker://fribdaq/frib-buster:v2.5
```

This can be done as an ordinary user. The following warnings are normal and won't affect the container:

```
WARNING: Building container as an unprivileged user. If you run this container as root

WARNING: it may be missing some functionality.
```

Obtain and unwrap a /usr/opt directory tree

These are maintained as tarballs in https://sourceforge.net/projects/nscldaq/files/for-containers/ The tarball for the jessie container is: usropt-jessie.tar.gz Due to limits on sourceforge distributable files, the buster /usr/opt is broken into a set of tarballs named usropt-buster-nnn.tar.gz where nnn is a number (e.g. usropt-buster-001.tar.gz

Download the appropriate tarball(s) and unwrap it/them into the your directory. usropt-jessie.tar.gz will create the directory tree: opt-jessie and the usropt-buster-nnn.tar.gz tarballs will create opt-buster. The order in which the usropt-buster-nnn.tar.gz tarballs are unwrapped is unimportant.

Create a start script

The important thing to keep in mind here is that when you start a singularity container, by default you'll have your home directory and the current working directory available to you. You can make other directories available via *binding* a binding specifies a host file or directory and optionally where that file or directory appears inside the container. So, if we're running a buster container, we want the opt-buster subdirectory to be bound into /usr/opt to match what to expect at the FRIB.

Here's a starting point for a start script.

```
#!/bin/bash

export SINGULARITY_SHELL=/bin/bash

# Modify the line below so that TOP_LEVEL is the directory you
#  created that holds opt-{buster,jessie}

TOP_LEVEL=/usr/opt        # Where we put all this stuff at FRIB.
```

```
# Modify the line below to point at your singularity image (e.g.
$TOP_LEVEL/buster.img)

IMAGE=$TOP_LEVEL/buster.img

# Modify the line below to point at your usropt tree (e.g.
$TOP_LEVEL/opt-buster)

USROPT=$TOP_LEVEL/opt-buster

TZ=`date +%Z` singularity shell \
     --bind $USROPT:/usr/opt  $IMAGE
```

Note that you can add additional --bind options The values after --bind are of the form:

host-file[:container-location]

Where host-file is some host file or directory and the optional container-location is where you want host-file to appear in your container environment. If container-location is missing, it appears in the same location e.g:

--bind /projects

binds /projects to /projects while

--bind /projects/fox:/foxproj

makes /projects/fox appear as /foxproj in the container.

The `TZ=`date +%Z`` sets the time zone environment to the time zone of the host system. This is necessary because we do not configure a time zone in the container environment as we don't a-prior know which time zone a host system is located in. This means that ordinarily the container environment would be providing times in UTC. The TZ environment variable gets propagated into the container where it informs the system time functions.

Test your script

install the script you create in the previous step (we normally put it in the same directory as the image but you can put it where you want). We normally name the script after the image it's going to run e.g. buster.sh Be sure to set it executable e.g. if you created /usr/opt for your containers;

```
install -m 0775 buster.sh /usr/opt/buster.sh
```

Now run the script you'll get some output

```
/usr/opt/buster.sh

Singularity: Invoking an interactive shell within container...
```

```
bash: /etc/bash_completion: No such file or directory
fox@flagtail:
```

if you poke around in /usr/opt now you should see, and be able to use the NSCL/FRIB software.

Notes on taking data.

Kernel module cannot be loaded from within the container. Furthermore the two servers that NSCLDAQ needs to run are typically started at system boot time and must run within a container as well. All of this can be tricky. The NSCLDAQ /usr/opt images currently provide a pair of drivers:

1. A fork of the Bit3/SBS/GE/ABACO driver for the fiber optic PCI/VME bus bridge.
2. A snapshot of the Broadcom PLX driver family needed for systems using the XIA Pixie16 boards. The Pixie16 board require the 9054 variant of these drivers. Furthermore, it was discovered that the driver and API are matched and detect mismatches. That is if you load a driver from a different version of the Broadcom package than the API used, this is detected and no XIA cards are found

Starting the NSCLDAQ servers

Two servers are must run if NSCLDAQ is to be used for data taking. These are not needed in systems that are only being used for offline analysis. They are only required for the NSCLDAQ data flow for online data taking. The first, DaqPortManager is responsible for allocating server ports from a pool of ports and supporting service discovery from among the assigned ports. The second, RingMaster, is responsible for managing ringbuffer client slot and setting up remote access.

In the containerized NSCLDAQ, these must be started within a container and must be from some relatively recent version of NSCLDAQ. In the example commands below we're going to assume you have the following shell script/environment definitions made:

Definition	**Purpose/Meaning**
USEROPT	Path to the /usr/opt image you want to use within the container (e.g. Jessie or Buster tree depending on the container you're using).
SINGULARITY_CONTAINER	Path to the container image you're going to run the servers within.
DAQPORTMANAGER	Path to the specific port manager you're using within the container filesystem. The server is `DaqPortManager` within the bin subdirectory of the installation of NSLDAQ you'll run the server from. This path should be within the container filesystem (e.g.

	/usr/opt/daq/11.3-023/bin/DaqPortManager)
DAQPORTMANAGERLOGFILE	Path within the container to a file that the port manager will log to.
DAQPORTMANAGERPIDFILE	Path within the container to a file that will contain the PID of the DAQ port manager (used to stop the server if needed).
RINGMASTER	Path within the container to the ring master server e.g. /usr/opt/daq/11.3-023/bin/RingMaster
RINGMASTERLOGFILE	Path within the container to where the ringmaster will write it's log file.

The following two script lines will start both of these servers. Note that there's no requirement that these servers run with root access.

```
nohup singularity exec --bind "${USROPT}:/usr/opt" --no-home  \
      "${SINGULARITY_CONTAINER}"  "${DAQPORTMANAGER}" \
     -log "${DAQPORTMANAGERLOGFILE}" \
     -pidfile "${DAQPORTMANAGERPIDFILE}"  \
     </dev/null >/dev/null 2>&1 &


nohup singularity exec --bind "${USROPT}:/usr/opt" --no-home  \
   "${SINGULARITY_CONTAINER}"  ${RINGMASTER} \
    -f /tmp/ringmaster.log \
    </dev/null >/dev/null  2>&1 &
```

These script lines are, in fact, taken from the init.d start script that's used to start NSCLDAQ at FRIB.

Kernel drivers

Kernel drivers are a thornier problem. Kernel drivers must:

- Be compiled in the host system.
- Be recompiled after kernel updates
- Be loaded in the host system.

Kernel updates can happen as a result of routine security updates and be ticking time bombs as the new kernel won't actually prevent the driver load until the next system reboot which can be days or even months following the update. Building a kernel module requires that appropriate support software be

installed on the host system (e.g. the Debian linux-headers pseudo package).

The dkms package attempts to address this problem by automating the recompilation of driver code on reboot. Using it, however is problematic because:

- It requires a specific source code organization not met by the Plx software and, therefore, each Broadcom release would require re-importation into the dkms package.
- It has happened in the past that dkms was not able to build a driver due to changes in
- kernel/driver interfaces that crept in as a result of a security update. Those failures are not reported by dkms and simply result in a non-functioning system that may be discovered at a later (usually more time-critical) time.
- Dkms driver packages are distributed via the distribution system of the host system (e.g. .deb's for Debian based systems and .rpm for RHEL based systems). This is problematic as the FRIB would need to be able to support a wide variety or kernel packaging systems.

The approach we took at the FRIB was differs from dkms and allows for a centralized set of driver sources. Given the build systems for the btp driver (BIT3/SBS/GE/ABACO bus bridge) and Plx driver famiy (XIA Pixie16 cards), we take slightly different approaches with both devices. However in both cases we:

1. Detect via our init.d script that we don't yet have a driver for the running system.
2. Build the drivers for the running system emailing the stdout and stderr of that build to an appropriate party (in our case our trouble ticketing system) for review.
3. Load the appropriate driver.

If you do not need one or the other of these drivers, feel free to ignore the corresponding section below.

Btp driver

qqThe build structure for this driver makes it easy to maintain a single source tree and build drivers for appropriate kernel versions. First we construct a filename for the kernel module we'll need to load. This the target off the module build qualified by the running kernel version:

```
KERNEL_VERSION="`uname -r`"
KERNEL_SOURCE_DIR="/usr/src/linux-headers-${KERNEL_VERSION}"
DRIVERBUILD_EMAIL=whereemailsgo@yourhost.yourdomain #Emails here.

BTPDRIVERDIR="${DRIVER_TOPDIR}/btp"
BTPDRIVER="${BTPDRIVERDIR}/btp-${KERNEL_VERSION}.ko"
```

The DRIVER_TOPDIR has been defined to be the top directory for all of this driver stuff. In our world

the btp subdirectory has stuff in the sbs/driver/ subdirectory of the NSCLDAQ source tree.

Testing for the need to build and building:

```
if test ! -e "${BTPDRIVER}"
   then
      rm -f "${BTPDRIVERDIR}/dd/*.o"  "${BTPDRIVERDIR}/dd/*.ko"
      (cd "${BTPDRIVERDIR}/dd"; \
       make -C "${KERNEL_SOURCE_DIR}" SUBDIRS=`pwd` BTDRIVER=`pwd`/..) \
          2>&1 | mail\
             -s"Driver build for btp-${KERNEL_VERSION} on`hostname`"   \
             "${DRIVERBUILD_EMAIL}"
        cp "${BTPDRIVERDIR}/dd/btp.ko"  "${BTPDRIVER}"
   fi
```

The SBS build infrastructure does not have a good clean target to its Makefile hence the rm. Note the cp command to copy the output of the build to a driver name that's qualified by the kernel version. Loading the driver looks like this:

```
insmod "${BTPDRIVER}"
"${BTPDRIVERDIR}/dd/mkbtp" 5
```

Plx driver

The Plx driver is actually a family of drivers. Our /usr/opt tarballs actually contain a full Plx driver release including sources. In the Plx ecosystem, the drivers are actually a directory tree and the script buildalldrivers builds all of the drivers for the entire Plx chipset family. What we therefore do is maintain a driver directory tree for each kernel version we've seen. To do this we've

1. Built a script that copies the base driver tree to a kernel qualified driver tree.
2. Modified the Plx_load script so that it loads from the kernel qualified tree rather than the base tree.

We define several symbols that are common with the btp driver build and show them again here so you don't need to read that section of the driver.

```
KERNEL_VERSION="`uname -r`"
KERNEL_SOURCE_DIR="/usr/src/linux-headers-${KERNEL_VERSION}"
DRIVERBUILD_EMAIL=whereemailsgo@yourhost.yourdomain #Emails here.
```

We have a directory tree pointed to by the symbol DRIVER_TOPDIR (not shown here) which has subdirectories btp for the bpt driver and plx for the entire plx software tree. This results in the definitions:

```
PLX_SDK_DIR="${DRIVER_TOPDIR}/plx"
PLXDRIVERDIR="${PLX_SDK_DIR}/Driver-${KERNEL_VERSION}"
```

If necessary on boot a new driver is built:

```
if test ! -d "${PLXDRIVERDIR}"
   then
      (cd "${PLX_SDK_DIR}"; ./mkdrivertree)
      (cd "${PLXDRIVERDIR}" ; \
             PLX_SDK_DIR=${PLX_SDK_DIR} \
      ${PLXDRIVERDIR}/buildalldrivers clean; \
      PLX_SDK_DIR=${PLX_SDK_DIR} ${PLXDRIVERDIR}/buildalldrivers) \
        2>&1 |\
        mail -s \
         "Build of PLX Driver family for ${KERNEL_VERSION} on `hostname`"\
         "${DRIVERBUILD_EMAIL}
   fi
```

Note the mkdrivertree script we added to the Plx directory tree that builds a kernel-specific driver tree.

The ${PLX_SDK_DIR}/Bin/Plx_load script has already been modified by us to load the driver from the tree associated with the running kernel.

Note it's important to use the Plx distribution that's included with the /usr/opt tarball as all of the XIA support code has been built against the API that corresponds to the version of the driver in that tree. We use the same Plx distribution for all of our /usr/opt tarballs so several container ecosystems can co-exist on the same host seamlessly.