How to Develop KIPR-Suite

By: Zachary Sasser
(Email: zsasser@kipr.org, Github: @Zacharyprime)

Can I break my controller this way?:
In short, not really.

If you mess up something in software, it can be instantly fixed by putting a new SD card (loaded with KIPR-OS) into your Wombat. Software can always be fixed.

However, there are “ways” to break hardware with software, for example running a program designed to overheat the processor. You can’t really do this by accident.

What does “Building KIPR-Suite” mean:
KIPR-Suite is the collection of software that makes up KIPR controllers. You can see many of them (and our other software) on our github page https://github.com/kipr

“Building” a project (collection of code) is the same thing as compiling it. There are some technical details that make the process different from compilation, but they aren’t important here.

KIPR Suite Software:
KIPR OS:
As of writing this, the OS a modified version of Debian for the Raspberry Pi. Usually when people say “KIPR OS” they mean “botui” which is the UI that you see on the screen (normally).

Botui:
This is the UI that you normally would see when the controller boots. This also handles sending the data you input in that UI to the necessary program or hardware.

Libwallaby:
This is the library that is compiled alongside your code in KISS IDE that handles the interaction between your code and the hardware. This is the program you will most likely be changing, if you are interested in this document.

Harrogate:
Harrogate is a complicated piece of software. It handles all of the KISS IDE as well as the Wifi Connections for the Wombat to name a few things. It is recommended not to change this unless you are an experienced Computer Scientist/Programmer.

Github Links:
KIPR Organization: https://github.com/kipr
Botui: https://github.com/kipr/botui
Libwallaby: https://github.com/kipr/libwallaby
Harrogate: https://github.com/kipr/harrogate

Update Manager: https://github.com/kipr/KIPR-Update

Warning Notice:

This document is to be used at your own risk and to your own competency. KIPR and Zachary Sasser are not responsible for any consequences of following these instructions. KIPR Staff is also not responsible to explain, engineer, or create any material for this purpose. The intent of this document is for students and teachers with the technical knowledge to access functionality that is not normally supported. Nothing in this document is officially supported by KIPR and is purely for educational and informational purposes. Some staff that are able, may assist you in your project, but as it is not part of the curriculum, that is out of that individual's willingness and ability.

If you do need help, contact me directly at zsasser@kipr.org or ztsasser@gmail.com and I will be happy to try and assist you. This is not an official service of KIPR or a guarantee.
(I just want to see you make cool stuff)

Navigating to Terminal:

1. Press CTRL+D to go to Desktop (Or click the "Hide UI" button)
2. Double Tap/Click on "Root Terminal"

(alternatively) Using SSH:

Windows:

1. Open the start menu or hit the Windows Key
2. Type "cmd" and hit enter
3. Connect to your Wombat via Ethernet or Wifi
4. Type "ssh pi@192.168.125.1"
    a. If you have done this with a Wallaby, you need to delete the file "C:/Users/<your username>/.ssh/known_hosts" to connect. (make sure "show hidden files" is enabled".
5. The password is "wallaby", you should now have access to a remote terminal.

Linux/MacOSx/Unix:

1. Navigate to a terminal (for unix, search terminal in the launchpad)
2. Type "sudo ssh pi@192.168.125.1" and enter password "wallaby"

What if I don’t know Linux?

This is not a Linux tutorial, ssh is just a useful tool for developing on the Wombat. The rest of this document will assume a small amount of Linux knowledge. If you are rusty on Linux or want to learn by doing this, here are some resources. If this doesn’t make sense, try looking online for Linux tutorials or your local bookstore/library for books (there are lots of them, Linux is old).

Linux Terminal Cheat-Sheet (online): https://www.tecmint.com/linux-commands-cheat-sheet/
Linux Terminal Cheat-Sheet (printable): https://files.fosswire.com/2007/08/fwunixref.pdf
Linux Terminal Handout (Zachary Sasser): External Link to Google Doc

Pulling from Github:

If you aren’t familiar with Github, here are a few resources. However, github is very useful so if you have the time and willingness, I suggest at least learning the basics. Github is a massive tool that nearly everyone in Computer Science uses. Github also sends representatives to GCER (Global Conference on Educational Robotics) fairly often, this may be a chance to learn more about it.

Source 1: https://opensource.com/article/18/1/step-step-guide-git
Source 2: https://guides.github.com/activities/hello-world/

For this tutorial, we are going to be pulling Botui and adding the “Reflash” button that is actually going to be a feature on the Wombat after this is finished. We are going to start by downloading the source code for botui. Fair warning that the actual programming may be a bit complex, but I will do my best to explain it in detail.

To start, we are going to create a folder to develop inside to keep it separate from everything else. Then we will fill it with libwallaby and botui. In order to do this, we need to plug in the Wombat to an ethernet cable so that we can use the internet.

```
kipr@Lab-Dinosaur:~$ ssh pi@192.168.125.1
pi@192.168.125.1's password:

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
You have new mail.
Last login: Thu Sep  3 15:20:28 2020 from desktop-t0q640h

Wombat:~ $
```

```
Wombat:~ $ mkdir Dev
Wombat:~ $ ls
botguy.png              screenshots            wallaby_reset
botui_settings          splash.png             wallaby_reset_coproc
Desktop                 stm32flash             wallaby_set_id.sh
Dev                     task.sh                wallaby_set_serial.sh
flash_instructions.txt  updateMe.sh            wallaby_v7.bin
got2                    update-online          wallaby_v8.bin
harrogate               users.json             wifi_configurator.py
i2s                     wallaby_flash          Wombat.img
KIPR-Update             wallaby_get_id.sh      workspace.json
linux                   wallaby_get_serial.sh  zoobee_launcher.sh
logfile.txt             wallaby_init_gpio
py_compile.py           wallaby_rcd.sh
Wombat:~ $ cd Dev
Wombat:~/Dev $ git clone https://github.com/kipr/botui
Cloning into 'botui'...
remote: Enumerating objects: 7206, done.
remote: Total 7206 (delta 0), reused 0 (delta 0), pack-reused 7206
Receiving objects: 100% (7206/7206), 6.81 MiB | 811.00 KiB/s, done.
Resolving deltas: 100% (3275/3275), done.
Checking connectivity... done.
Wombat:~/Dev $ git clone https://github.com/kipr/libwallaby
Cloning into 'libwallaby'...
remote: Enumerating objects: 1244, done.
remote: Counting objects: 100% (1244/1244), done.
remote: Compressing objects: 100% (267/267), done.
remote: Total 7266 (delta 1052), reused 1131 (delta 973), pack-reused 6022
Receiving objects: 100% (7266/7266), 12.60 MiB | 1.00 MiB/s, done.
Resolving deltas: 100% (5475/5475), done.
Checking connectivity... done.
```

Adding the Reflash Button:

Often while working on a github project, it is good to have it’s page up so you can reference comments that other developers have made, view history of the file, and view other useful information. This makes it much easier to piece all the code together in your head.

We are going to add our button to the “Update” page (this may not be where you see the button when it is released to the public). To do this, we will have to edit a file in “Include”, “UI”, and “src”. “Include” handles what you’d expect, definitions of objects, functions, and variables. “src” is all the code that works in the background of the UI, processing the buttons to mean something in hardware. “UI” contains special .ui files that are for the Qt library that handles the Graphical User Interface (GUI). The Qt manual is found HERE.

The Include:

The file we are interested in depends on the page you would like to edit. In our case, we are going to be editing “WallabyUpdateWidget.h”. Inside the file you will see 3 public function definitions. These are associated with the current 3 buttons. We are going to add a function called “reflash ()”

```
#ifndef _WALLABYUPDATEWIDGET_H_
#define _WALLABYUPDATEWIDGET_H_

#include "StandardWidget.h"
#include <QString>
#include <QProcess>
#include <QDir>

namespace Ui
{
  class WallabyUpdateWidget;
}

class WallabyUpdateWidget : public StandardWidget
{
Q_OBJECT
public:
  WallabyUpdateWidget(Device *device, QWidget *parent = 0);
  ~WallabyUpdateWidget();

public slots:
  void update();
  void refresh();
  void ethernet();
  void reflash();  // This is our added function

private slots:
  void updateFinished(int exitCode, QProcess::ExitStatus exitStatus);

private:
  bool mountUsb(const QString device, const QDir dir);
  bool unmountUsb(const QString device);

  static const QString updateFileName;
  static const QDir mountDir;
  Ui::WallabyUpdateWidget *ui;
  QProcess *m_updateProc;
};
#endif
```

The src:

Inside of src, we will be working with “WallabyUpdateWidget.cpp”. The file itself is too large to display this way, the block of code below is just a chunk of the file. A function that I’ve previously made is Ethernet which handles the Online Update button. We will be reusing code from this to create the new function.

```
void WallabyUpdateWidget::ethernet(){
        if(QMessageBox::question(this, "Update?", QString("Is the ethernet
cable plugged into the controller?"), QMessageBox::Yes | QMessageBox::No)
!= QMessageBox::Yes)
      return;

      // Change UI to show output
      ui->updateConsole->setVisible(true);
      ui->selectionWidget->setVisible(false);
      ui->statusLabel->setText("Update progress:");

      // Run update process
      m_updateProc = new QProcess();
      m_updateProc->setProcessChannelMode(QProcess::MergedChannels);
      ui->updateConsole->setProcess(m_updateProc);
      connect(m_updateProc, SIGNAL(finished(int, QProcess::ExitStatus)),
SLOT(updateFinished(int, QProcess::ExitStatus)));

      m_updateProc->start("sh /home/pi/updateMe.sh");
}
```

The Modified Function:

```
void WallabyUpdateWidget::reflash(){
        // Change UI to show output
      ui->updateConsole->setVisible(true);
      ui->selectionWidget->setVisible(false);
      ui->statusLabel->setText("Update progress:");

      // Run update process
      m_updateProc = new QProcess();
      m_updateProc->setProcessChannelMode(QProcess::MergedChannels);
      ui->updateConsole->setProcess(m_updateProc);
      connect(m_updateProc, SIGNAL(finished(int, QProcess::ExitStatus)),
SLOT(updateFinished(int, QProcess::ExitStatus)));

      m_updateProc->start("sudo ./wallaby_flash"); //<--This is what
changed
}
```

What does any of that mean?:

To understand the entire function, you'll have to learn how to use Qt. All of the top code is basically creating a text window so that you can see what the console is outputting (like when you update your controller and it tells you where it's at). The only thing we've changed is removing the "are you sure" prompt and we changed what console command it runs. Running "sudo ./wallaby_flash" is the command to reflash the processor.

Attaching the button to our function:

We just have two more things to change before we can exit this file. We have to connect the button to our function. Then because update() disables all the other buttons, we need to make that happen to our button too.

```
WallabyUpdateWidget::WallabyUpdateWidget(Device *device, QWidget *parent)
  : StandardWidget(device, parent),
  ui(new Ui::WallabyUpdateWidget)
{
  ui->setupUi(this);
  performStandardSetup("Update");

  ui->updateConsole->setVisible(false);

  connect(ui->update, SIGNAL(clicked()), SLOT(update()));
  connect(ui->refresh, SIGNAL(clicked()), SLOT(refresh()));
  connect(ui->ethernet, SIGNAL(clicked()), SLOT(ethernet()));

  connect(ui->reflash, SIGNAL(clicked()), SLOT(reflash())); // <---
}
```

Note: "reflash" and "reflash()" are not the same. "ui->reflash" is the Qt button object, "reflash()" is our function for handling the button.

Fixing Context Errors:

Next we will need to make sure the buttons get disabled when we press “Update”. This is inside of the update() function. Not all of the function is shown below.

```
void WallabyUpdateWidget::update()
{
  // Get single selected item
  const QList<QListWidgetItem *> selected =
ui->updateList->selectedItems();
  if(selected.size() != 1)
    return;
  const QString selectedName = selected.at(0)->text();
  // Verify with user that they want to do the update
  if(QMessageBox::question(this, "Update?",
    QString("Are you sure you want to update using %1?").arg(selectedName),
    QMessageBox::Yes | QMessageBox::No) != QMessageBox::Yes)
      return;

  // Disable buttons
  ui->update->setEnabled(false);
  ui->refresh->setEnabled(false);
  ui->ethernet->setEnabled(false);

  ui->reflash->setEnabled(false); //<-- OUR CHANGE

  ...
    else {
      // Change UI to show output
      ui->updateConsole->setVisible(true);
      ui->selectionWidget->setVisible(false);
      ui->statusLabel->setText("Update progress:");
      // Run update process
      m_updateProc = new QProcess();
      m_updateProc->setProcessChannelMode(QProcess::MergedChannels);
      m_updateProc->setWorkingDirectory(subDir.absolutePath());
      ui->updateConsole->setProcess(m_updateProc);
      connect(m_updateProc, SIGNAL(finished(int, QProcess::ExitStatus)),
SLOT(updateFinished(int, QProcess::ExitStatus)));
      m_updateProc->start("sh", QStringList() <<
WallabyUpdateWidget::updateFileName);

      // Update script will reboot the controller
    }
  }
}
```

Note: This is optional, your button will work without this but it is bad practice.

The UI File:

Finally, we will edit the UI file. The .ui file is like an HTML file where it defines the layout of the page. All the graphical parts are handled by Qt based on this file. Qt Creator is a useful tool for editing these files so you don't have to figure out how UI files work, but for this tutorial we will be editing the raw files with a text editor. The file we are interested in is "WallabyUpdateWidget.ui" (notice a pattern?).

Because making a button from scratch would be complicated, we are just going to copy the ethernet button and make a few changes. It is recommended you go view the entire file to understand how it works, it is too large to be put into this format.

```
<item>
         <spacer name="verticalSpacer">
          <property name="orientation">
           <enum>Qt::Vertical</enum>
          </property>
          <property name="sizeHint" stdset="0">
           <size>
            <width>20</width>
            <height>40</height>
           </size>
          </property>
         </spacer>
        </item>
        <item>
         <widget class="QPushButton" name="ethernet">
          <property name="font">
           <font>
            <pointsize>33</pointsize>
           </font>
          </property>
          <property name="text">
           <string>Online Update</string>
          </property>
         </widget>
        </item>
       </layout>
      </item>
     </layout>
    </widget>
   </item>
```

Note: If you are unfamiliar with markup languages, a container is defined with a open bracket <container name> and then closed with a closing bracket </container name>

To make our button, we are going to place it right above the Online Update button. This way we can just copy and paste and change some text. Feel free to play around with the different properties. Qt Creator will show you many more things you can edit and play with.

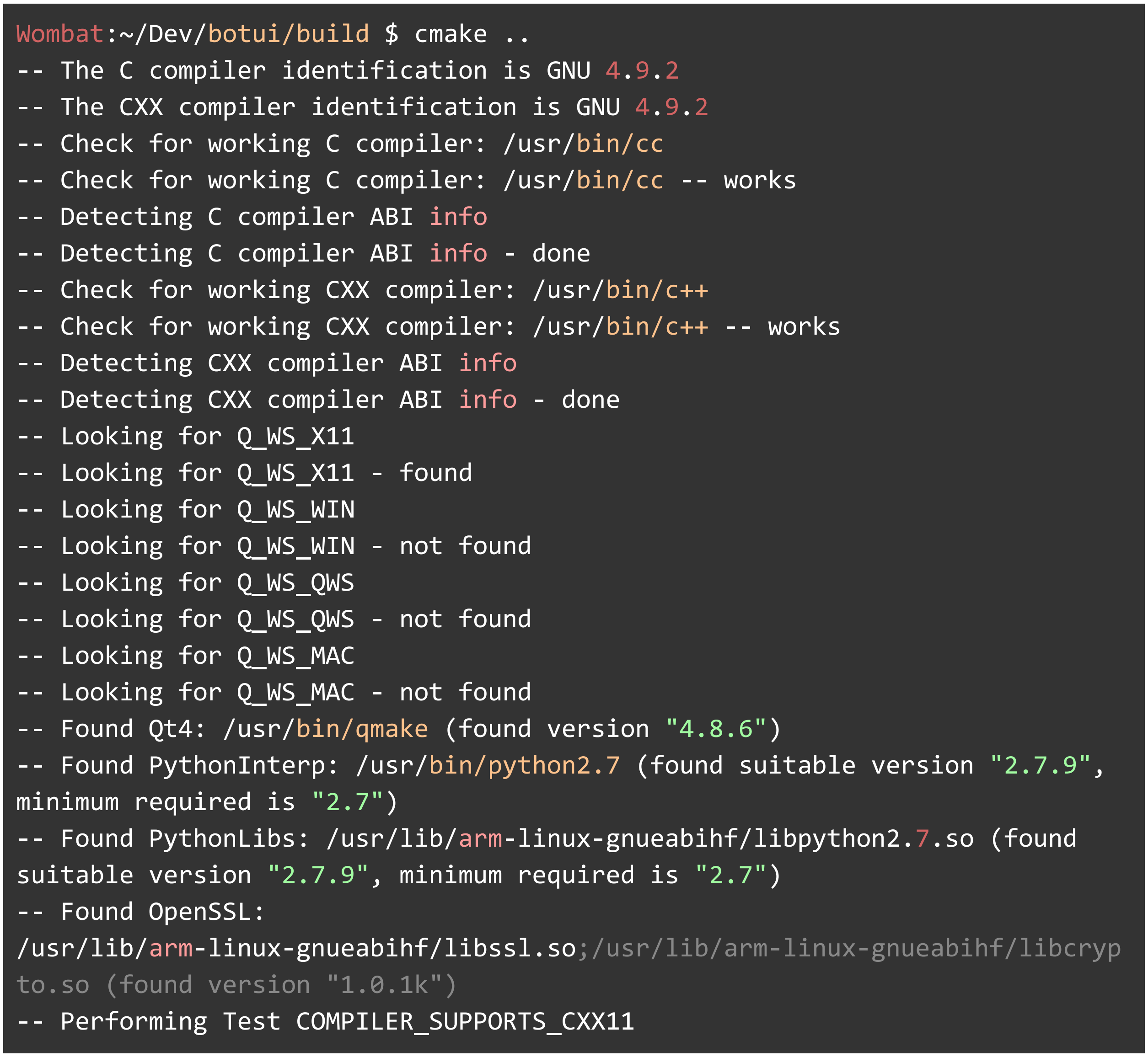
```
         <item>
          <spacer name="verticalSpacer">
           <property name="orientation">
            <enum>Qt::Vertical</enum>
           </property>
           <property name="sizeHint" stdset="0">
            <size>
             <width>20</width>
             <height>40</height>
            </size>
           </property>
          </spacer>
         </item>

         <item>
          <widget class="QPushButton" name="reflash">
           <property name="font">
            <font>
             <pointsize>33</pointsize>
            </font>
           </property>
           <property name="text">
            <string>Reflash Processor</string>
           </property>
          </widget>
         </item>

         <item>
          <widget class="QPushButton" name="ethernet">
           <property name="font">
            <font>
             <pointsize>33</pointsize>
            </font>
           </property>
           <property name="text">
            <string>Online Update</string>
           </property>
          </widget>
         </item>
        </layout> ...
```

Building the Package (compiling):

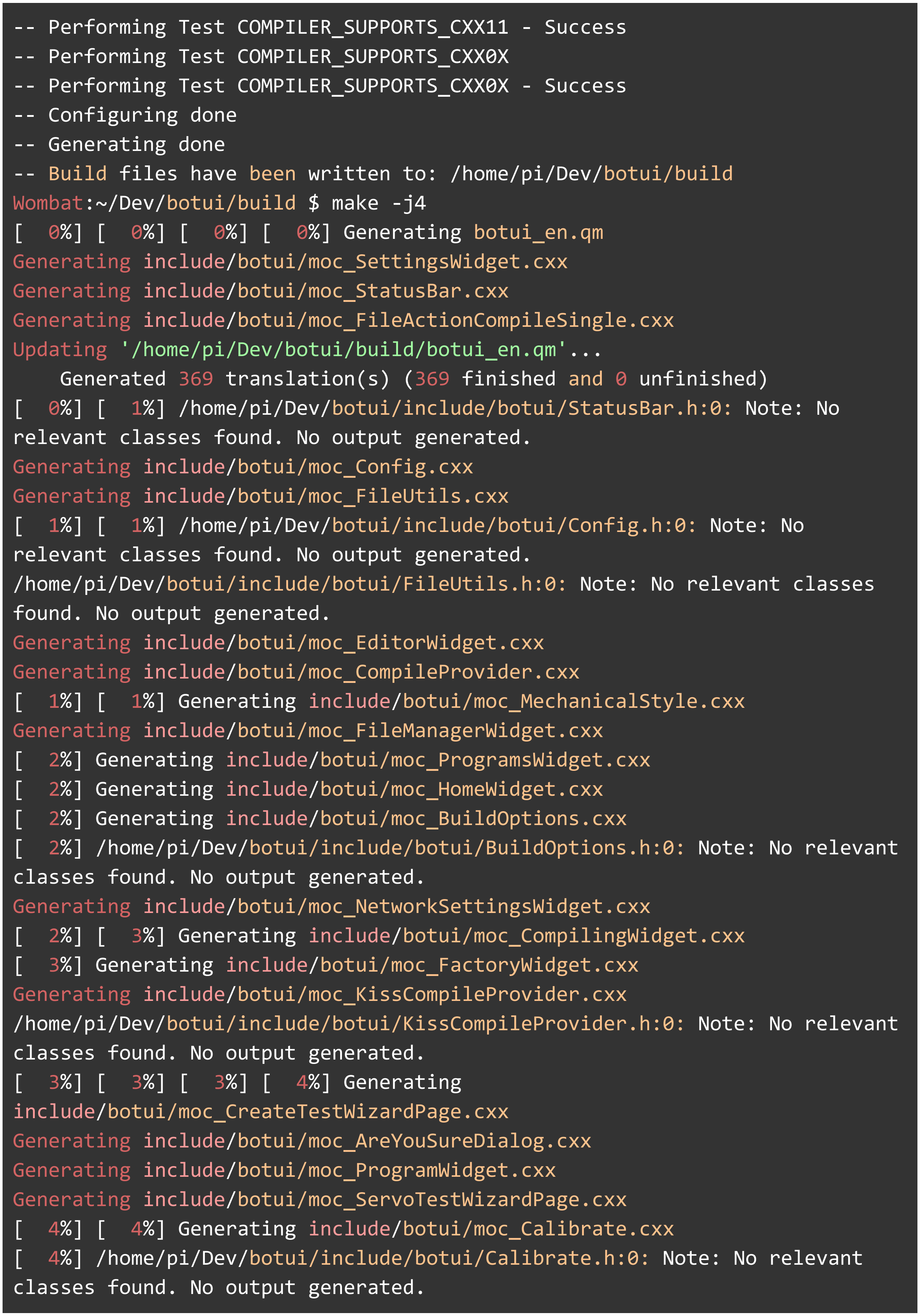
To build our package, we are going to be using cmake and make, then packaging it with dpkg which will allow us to distribute it to others. To do this we will make a folder called “build” in the main botui folder.

Then we will use “cmake ..” to generate build files for the project. This is essentially going through our project and finding what files are part of our project and need to be compiled. Make will then use that data to compile all of the files. Adding -j4 forces make to use all 4 cores of the Raspberry Pi processor. In order to show what your output should look like, most of the important output is shown below.

1. mkdir build && cd build
2. cmake ..
3. make -j4



```
Wombat:~/Dev/botui/build $ cmake ..
-- The C compiler identification is GNU 4.9.2
-- The CXX compiler identification is GNU 4.9.2
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Looking for Q_WS_X11
-- Looking for Q_WS_X11 - found
-- Looking for Q_WS_WIN
-- Looking for Q_WS_WIN - not found
-- Looking for Q_WS_QWS
-- Looking for Q_WS_QWS - not found
-- Looking for Q_WS_MAC
-- Looking for Q_WS_MAC - not found
-- Found Qt4: /usr/bin/qmake (found version "4.8.6")
-- Found PythonInterp: /usr/bin/python2.7 (found suitable version "2.7.9",
minimum required is "2.7")
-- Found PythonLibs: /usr/lib/arm-linux-gnueabihf/libpython2.7.so (found
suitable version "2.7.9", minimum required is "2.7")
-- Found OpenSSL:
/usr/lib/arm-linux-gnueabihf/libssl.so;/usr/lib/arm-linux-gnueabihf/libcryp
to.so (found version "1.0.1k")
-- Performing Test COMPILER_SUPPORTS_CXX11
```



```
-- Performing Test COMPILER_SUPPORTS_CXX11 - Success
-- Performing Test COMPILER_SUPPORTS_CXX0X
-- Performing Test COMPILER_SUPPORTS_CXX0X - Success
-- Configuring done
-- Generating done
-- Build files have been written to: /home/pi/Dev/botui/build
Wombat:~/Dev/botui/build $ make -j4
[  0%] [  0%] [  0%] [  0%] Generating botui_en.qm
Generating include/botui/moc_SettingsWidget.cxx
Generating include/botui/moc_StatusBar.cxx
Generating include/botui/moc_FileActionCompileSingle.cxx
Updating '/home/pi/Dev/botui/build/botui_en.qm'...
    Generated 369 translation(s) (369 finished and 0 unfinished)
[  0%] [  1%] /home/pi/Dev/botui/include/botui/StatusBar.h:0: Note: No
relevant classes found. No output generated.
Generating include/botui/moc_Config.cxx
Generating include/botui/moc_FileUtils.cxx
[  1%] [  1%] /home/pi/Dev/botui/include/botui/Config.h:0: Note: No
relevant classes found. No output generated.
/home/pi/Dev/botui/include/botui/FileUtils.h:0: Note: No relevant classes
found. No output generated.
Generating include/botui/moc_EditorWidget.cxx
Generating include/botui/moc_CompileProvider.cxx
[  1%] [  1%] Generating include/botui/moc_MechanicalStyle.cxx
Generating include/botui/moc_FileManagerWidget.cxx
[  2%] Generating include/botui/moc_ProgramsWidget.cxx
[  2%] Generating include/botui/moc_HomeWidget.cxx
[  2%] Generating include/botui/moc_BuildOptions.cxx
[  2%] /home/pi/Dev/botui/include/botui/BuildOptions.h:0: Note: No relevant
classes found. No output generated.
Generating include/botui/moc_NetworkSettingsWidget.cxx
[  2%] [  3%] Generating include/botui/moc_CompilingWidget.cxx
[  3%] Generating include/botui/moc_FactoryWidget.cxx
Generating include/botui/moc_KissCompileProvider.cxx
/home/pi/Dev/botui/include/botui/KissCompileProvider.h:0: Note: No relevant
classes found. No output generated.
[  3%] [  3%] [  3%] [  4%] Generating
include/botui/moc_CreateTestWizardPage.cxx
Generating include/botui/moc_AreYouSureDialog.cxx
Generating include/botui/moc_ProgramWidget.cxx
Generating include/botui/moc_ServoTestWizardPage.cxx
[  4%] [  4%] Generating include/botui/moc_Calibrate.cxx
[  4%] /home/pi/Dev/botui/include/botui/Calibrate.h:0: Note: No relevant
classes found. No output generated.
```

```
... (skipped lines)
[ 48%] Building CXX object CMakeFiles/botui.dir/src/MechanicalStyle.cpp.o
[ 48%] Building CXX object
CMakeFiles/botui.dir/src/ServoTestWizardPage.cpp.o
/home/pi/Dev/botui/src/CameraWidget.cpp: In member function 'void
CameraWidget::setChannelConfig(const Config&, int)':
/home/pi/Dev/botui/src/CameraWidget.cpp:35:37: warning: comparison between
signed and unsigned integer expressions [-Wsign-compare]
   if(m_camDevice->channels().size() <= channelNum)
                                     ^
/home/pi/Dev/botui/src/CameraWidget.cpp: In member function 'void
CameraWidget::update()':
/home/pi/Dev/botui/src/CameraWidget.cpp:98:36: warning: comparison between
signed and unsigned integer expressions [-Wsign-compare]
         for(int objNum = 0; objNum < objs->size(); ++objNum) {
                                    ^
[ 48%] Building CXX object CMakeFiles/botui.dir/src/QHsvPicker.cpp.o
[ 49%] Building CXX object CMakeFiles/botui.dir/src/Clipboard.cpp.o
[ 49%] Building CXX object CMakeFiles/botui.dir/src/ProgramArgsWidget.cpp.o
[ 49%] Building CXX object CMakeFiles/botui.dir/src/SystemUtils.cpp.o
[ 49%] Building CXX object CMakeFiles/botui.dir/src/NetworkItemModel.cpp.o
[ 50%] Building CXX object CMakeFiles/botui.dir/src/main.cpp.o
 ...
[ 99%] Building CXX object CMakeFiles/botui.dir/qrc_fonts.cxx.o
[100%] Building CXX object CMakeFiles/botui.dir/qrc_target.cxx.o
Linking CXX executable ../deploy/botui
make[2]: warning:  Clock skew detected.  Your build may be incomplete.
[100%] Built target botui
Wombat:~/Dev/botui/build $
```

Packaging (making it into a .deb):

1. Go back to the Dev folder, so that you are just outside of botui
2. Make a tar.gz file of the botui folder: tar -zcvf botui.tar.gz botui
3. Copy that tar.gz into the file: cp botui.tar.gz botui
4. Move into the directory: cd botui
5. Run this (inside the botui folder): dpkg-buildpackage -b -rfakeroot -us -uc
6. The resulting .deb will be outside of the botui folder (in Dev). The name of the file does not reflect the actual version but will probably say “botui_25.8-1_armhf.deb”.

The output(s) of the packaging process are all too massive to fit into this document. The program will essentially recompile everything into a form that can be used by other linux machines. This can take some time so be patient. This is how we produce the .deb files for github.com/kipr/KIPR-update which handles the online updates and USB stick updates.