**Crazyflie 2.0 Project**

Prerequisite:

* Virtual Machine image from Bitcraze (<https://wiki.bitcraze.io/projects:virtualmachine:index#download>)

Instructions:

* Import the VM image into Oracle VirtualBox

*The VM comes pre-installed and pre-configured for development.*

*You can update the software for all the projects installed on the VM.*

*Note: Until April 2016, the latest updated version of the Crazyflie Client has been crashing on me.*

*The Eclipse IDE installed has all the firmware projects imported and ready for development or running/debugging.*

*The Crazyflie Client is the application that connects to the Crazyflie quadcopter to fly and configure it. (*[*https://wiki.bitcraze.io/doc:crazyflie:client:pycfclient:index*](https://wiki.bitcraze.io/doc:crazyflie:client:pycfclient:index)*)*

* Update the projects.

The project folders are located under /home/bitcraze/projects/

Don’t run the script on the Desktop to avoid updating the Crazyflie Client.

Instead:

* Open a terminal
* Type :    cd projects/<project name>
* Type :    git pull

Our work is focused on the *crazyflie-firmware* project that is responsible for the flight control system. This code runs on the STM32F405 MCU.

* Open Eclipse

The source code is located under the *crazyflie-firmware* project.

The *README.md* file gives a brief overview of the folder structure of the project.

The most interesting folders for us are:

<drivers> - contains software drivers for the hardware components on the Crazyflie

<modules> - contains firmware for flight control and other modules running

<lib> - contains the Operating System and other drivers of peripheral components of the STM32F405 MCU

<init> - contains main.c

All the projects are compiled using Makefiles

* In the Make View, expand the crazyflie-firmware project
* Select either *all* or *MAKE CLOAD* to compile the code to be loaded on the quadcopter, or

Select *MAKE DEBUG* to compile a debug version of the code.

* Click on the little hammer icon to compile the code.
* To load the compiled binary onto the quadcopter, follow the instructions in the *Crazyflie Client* documentation page.

Creating a simple program to activate LEDs and Motors:

* Open main.c
* Comment out the content of the file
* Copy and paste the following code into main.c

*#include "led.h"*

*#include "motors.h"*

*/\* ST includes \*/*

*#include "stm32fxxx.h"*

*volatile bool ledOn = true;*

*int main()*

*{*

*uint16\_t i = 0;*

*uint16\_t j = 0;*

*motorsInit();*

*ledInit();*

*//  ledSet(0, 1);*

*//  ledSet(1, 1);*

*while (1) {*

*while (j < 300) {*

*if (i == 65000) {*

*i = 0;*

*j++;*

*}*

*i++;*

*}*

*if (ledOn) {*

*ledSet(LED\_RED\_L, true);*

*ledSet(LED\_RED\_R, false);*

*motorsSetRatio(MOTOR\_M1, 6000);*

*motorsSetRatio(MOTOR\_M2, 6000);*

*motorsSetRatio(MOTOR\_M3, 6000);*

*motorsSetRatio(MOTOR\_M4, 6000);*

*} else {*

*ledSet(LED\_RED\_L, false);*

*ledSet(LED\_RED\_R, true);*

*motorsSetRatio(MOTOR\_M1, 0);*

*motorsSetRatio(MOTOR\_M2, 0);*

*motorsSetRatio(MOTOR\_M3, 0);*

*motorsSetRatio(MOTOR\_M4, 0);*

*}*

*ledOn = !ledOn;*

*j = 0;*

*}*

*//Should never reach this point!*

*while(1);*

*return 0;*

*}*

This simple software activates the motors of the quadcopter and makes an alternating flash of the LEDs.

* Compile the code by running *MAKE CLOAD* in the Make View.

**Flight Control (v. CDBoost)**

Prerequisites:

* Linux environment
* g++ 4.8 or higher
* Boost 1.57 library or higher (<http://www.boost.org/users/download/>)
* CDBoost (<https://gforge.inria.fr/projects/cdboost>), or available folder copy
* Quadcopter project folder

Setup:

* The default folder organization of the project needs requires that the Boost and CDBoost library folders be placed inside the Quadcopter project folder.

Compiling and Running the simulation:

* Open a terminal and go to the Quadcopter directory (we’ll refer to it as QDir)
* Type :    make all

This will compile the project and create a *controller* binary in QDir

* To run the simulation in the terminal,

Type :    ./controller  test/test\_cases/sensor\_input.txt  test/test\_cases/commander\_input.txt

Note: The first entry of each line in the input files is the time when the next event, for that line, should be triggered in the DEVS simulation. The format is a fraction “numerator/denominator”, thus 15/1 for example equates to 15 units of time.

The project folder organization is as such:

<atomic\_models> - contains the implementation of all the atomic models of the whole system;

<model\_generator> - contains the modelGenerator class that creates the top level model of the entire system. This includes creating couple models and creating the connections between the atomic models and atomic to coupled models;

Note: When connecting models, turn the atomic models into coupled models using atomicToCoupled() to prevent issues in the CDBoost framework that runs the simulation.

<data\_structures> - contains a Message class that was defined to transmit structures containing the different types of data and commands between the atomic models. The design of this class is specific to the atomic models of this Quadcopter system;

<vendor> - contains a selection of utilities that are useful for Real-Time version of the model simulation;

<test> - contains unit tests for all the atomic models in the system;

<cdboost> - contains the library for the simulation engine that runs DEVS-based models;

<boost\_1\_57> - contains the Boost C++ library

Understanding the CDBoost engine:

The following is good reading materials for understand PDEVS, and how CDBoost (and subsequently ECDBoost) implementations work:

* Niyonkuru, D; Wainer, G, “Towards a DEVS-based Operating System”
* Niyonkuru, D; Wainer, G; Dalle, O; Vicino, D, “Sequential PDEVS Architecture”
* Niyonkuru, D; “Bare-Metal Kernels for DEVS Model Execution in Embedded Systems”

**Flight Control (v. ECDBoost)**

Prerequisites (building on the previous requirements):

* Replace CDBoost with pdevslib which is a Real-Time version of CDBoost, thus making it ECDBoost.

In this library, some classes have slight variations to the original CDBoost version, and there’s the introduction of the *driver* and *port* classes, and the change of *runner* to *erunner* and *input\_stream* to *event\_stream* classes.

Setup:

Note: ECDBoost references the *Time* (eTime.h) and *Message* (eMessage.h) classes found in <vendor> folder as opposed to the use of *BRITime* and *<data\_structures>/Message* classes that are used in the original Quadcopter system design.

The ECDBoost *driver* class has a dependency on the *Message* (eMessage.h) format. This can be easily bypassed by commenting the referencing code, as it is currently only used for logging and isn’t integral to the code execution. This way, we can keep using the message format from *<data\_structures>/Message* class.

The *Time* class calls driver interface functions from hwTime.h/cpp which is intended to interface with a MCU’s hardware timer. To bypass any blockers, before targeting the system to a hardware platform, references in Time to hwTime functions can be commented. And to solve the issue of continuing to advance the simulation time, a hack can be made inside Time::currentTime() by replacing the current code with:

*Time tmp (simStartTime);*

*simStartTime = simStartTime + Time(0,0,0,1);*

*return tmp;*

Note to change the type of *simStartTime* to from *RTime* to *Time*.

This simulates an advancement in time by 1msec at each call of Time::currentTime().

With the switch to ECDBoost, the top level model needs to introduce port connections between port objects and models from the main system. If no ports are required in the sytem design, the port connections are set to empty initializer lists in the construction of an erunner object (see declaration of erunner object in main.cpp).

Compiling and Running the simulation:

* Before being able to run the simulation on ECDBoost, you need to change the declaration of *runner* object in main.cpp to an *erunner* object.
* Compile and run the code the same way as instructed in the previous section.

**An Integrated Solution (Crazyflie firmware + Simple DEVS flight control + ECDBoost)**

Prerequisites:

* Virtual Machine image from Bitcraze (<https://wiki.bitcraze.io/projects:virtualmachine:index#download>)
* Boost **1.55** library (<http://www.boost.org/users/download/>)
* Quadcopter project folder:
  + Real-Time project version that uses **pdevslib** (ECDBoost) (ref. previous section)
  + Include pdevslib in the Quadcopter project folder

Setup:

* Open the VM
* Go to the *crazyflie-firmware* project directory (we’ll refer to it as CF\_Dir).
* Copy the Quadcopter project folder in CF\_Dir

Notes:

1. The integration of the Quadcopter project (QProj) with the crazyflie-firmware (CF\_Proj) creates a mixture of C++ and C code. This entails that the compilation of the whole solution will require a compiler that supports both C and C++.
2. The mix of C and C++ code raises the question about whether to make the main() program C++ or C. C++ is an extension of C. It supports C syntax. However, the reverse is relation is complicated. It’s easier to make main() a C++ program that makes external calls to C code rather than the opposite.
3. When making function calls to any of the CF\_Proj drivers from a QProj module (e.g. main(), atomic models, driver interface) remember to declare the functions in the QProj module as extern “C”. Also, the extern declaration should precede the function’s global declaration through the header file inclusion.
4. We want to create a solution that runs the Quadcopter’s DEVS-based flight control system with ECDBoost (bare-metal kernel) on the Crazyflie 2.0 (which involves activating hardware components on the drone such as rotors and LEDs).
5. We want to **exclude** any calls to the CF\_Proj RTOS (located in <lib> folder) that schedules applications in the CF\_Proj.
6. The QProj is intended to replace the CF\_proj flight control (located in the <modules> folder).
7. The easiest solution for 2.2 and 2.3 above, is to replace CF\_Proj’s main.c with QProj’s main.cpp. The reason behind that, is that CF\_Proj’s main() is the starting point where the RTOS and the flight control system are initialized. Hence, taking out those initialization calls easily accomplishes 2.2 and 2.3.
8. To compile the new hybrid solution, we need to make some modifications to CF\_Proj’s Makefile
9. In the *Build Configuration* section, under *#Crazyflie*, in the condition *ifeq($(F405), 1)*, add the path to the <vendor> folder in QProj, to the VPATH variable (e.g. “quadcopter/vendor”)
10. Add a new section, above the variable assignment of *OBJ*:

############### E-DEVS RT-files configuration ################

RT\_OBJ = eTime.o hwTime.o

1. Add $(RT\_OBJ) to the assignment line of *OBJ*.
2. Under Compilation configuration
   1. Add: CPP = $(CROSS\_COMPILE)g++
   2. Change the value of LD to : $(CROSS\_COMPILE)g++

We are defining a new variable to use to compile C++ files.

We are also changing the variable that defines the linker, from using gcc linker that only supports C files, to g++ linker than can support both C and C++ files.

1. Add a new section underneath:

############### E-DEVS section ################

DEVSDIR = <relative path to pdevs folder in pdevslib>

*(e.g. quadcopter/pdevslib/boost/simulation/pdevs)*

INC\_BOOST = -I<path to Boost 1.55 lib>

*(e.g. -I/home/bitcraze/Downloads/boost\_1\_55\_0)*

INC\_PDEVS = -I<path to pdevslib folder> *(e.g. -Iquadcopter/pdevslib)* -I$(DEVSDIR) -I$(DEVSDIR)/basic\_models

INC\_DEVSMODELS = -I<**paths** to all folders containing DEVS models: atomic, ports>

*(e.g. -Iquadcopter/atomic\_models –Iquadcopter/devs\_ports -Iquadcopter/model\_generator)*

INC\_DEPENDENCIES = -I<**paths** to all folders containing utility classes used by DEVS models>

*(e.g. -Iquadcopter/vendor -Iquadcopter/data\_structures)*

###############################################

1. In the following variable assignment section for INCLUDES,

Add: INCLUDES+= $(INC\_DEPENDENCIES) $(INC\_BOOST) $(INC\_PDEVS) $(INC\_DEVSMODELS)

1. After the conditional statement under *#Flags required by ST Library*
   1. Add: STFLAGS += -DTARGET\_M4 -DTARGET\_CORTEX\_M -DTARGET\_STM -DTARGET\_STM32F4 -DTOOLCHAIN\_GCC\_ARM -DTOOLCHAIN\_GCC -D\_\_CORTEX\_M4 -DARM\_MATH\_CM4
2. Below, where there are variable assignments to CFLAGS
   1. Add:

# Compiler flags to generate dependency files:

CFLAGS += -MD -MP -MF $(BIN)/dep/$(@).d -MQ $(@)

#Permits to remove un-used functions and global variables from output file

CFLAGS += -fno-exceptions -ffunction-sections -fdata-sections

CFLAGS += -fno-common -fmessage-length=0 -Wunused-parameter -Wunused-local-typedefs -Wextra #-Wreorder

CFLAGS += -fomit-frame-pointer –fpermissive

1. Change the initial assignment statement of LDFLAGS to:
   1. LDFLAGS = $(PROCESSOR) -Wl,--gc-sections --specs=rdimon.specs -Wl,-Map=$(PROG).map,--cref -lstdc++ -lsupc++ -lm -lc -lgcc –lnosys
2. Lastly, we need to add a make rule for the C++ files in *CF\_Dir/scripts/target.mk* file, between the rules for *CC\_COMMAND* and *LD\_COMMAND*:

CPP\_COMMAND=$(CPP) $(CFLAGS) -g -c -std=c++11 $< -o $(BIN)/$@

CPP\_COMMAND\_SILENT=" CPP $@"

.cpp.o:

@$(if $(QUIET), ,echo $(CPP\_COMMAND$(VERBOSE)) )

@$(CPP\_COMMAND)

### Building a Simple DEVS-based flight control

1. To get ourselves going, we can start with a simple system design made up of 1 atomic model (*flightDEVS*) for the flight control, an input port (*CommandPort*) and 4 output ports (*MotorPort[1-4]*).
2. Refer to the attached folder containing the program files needed.

The project folder organization is the same as that of the original QProj with the addition of:

<devs\_ports> - contains the definition of the port classes. The CommandPort simulates incoming thrust values by using an array of inputs. The MotionSensorPort class is currently out of use.

The MotorPort class calls the motor driver of the CF\_Proj to output the thrust level to each motor.

<vendor> - contains a new utility file called util.c that defines functions that turn on the LEDs. Those functions call the LEDs driver of *CF\_Proj*. As such, we need to add “util.o” to the *RT\_OBJ* variable assignment in the *CF\_Proj* Makefile.

This flight system uses the *Message* (eMessage\_s.h) format.

main.cpp defines the top level model *ControlUnit*, the atomic model *flightDEVS*, the ports *CommandPort* and *MotorPort[1-4]*, and the simulation *erunner root*.

1. Swap the attached Quadcopter project with the original one copied into *CF\_Proj*, and swap the new provided main.cpp with the *CF\_Proj* main.c
2. From a terminal run *make all DEBUG=0 CFLOAD=1* from *CF\_Dir*.
3. Load the compiled binary onto the *CF2* using the *Crazyflie Client*.