# Install Ubuntu 16.04 in a virtual machine (or main machine)

The flight software you will be running for this class exercise has been configured and compiled to run under the Ubuntu 16.04 operating system. To more rapidly start using the software, we will create a virtual Ubuntu operating system environment on your machine.

We will use a virtual machine to make the running of this software easy in a known environment. In these examples, I am assuming that we will use VirtualBox, but any virtualization system can work.

1. Download VirtualBox
2. Download Ubuntu .iso
3. Install Ubuntu in VirtualBox
   1. In the VirtualBox GUI select new machine with the following parameters
      1. Name is CFS
      2. Type is Linux
      3. Version is Ubuntu (64 bit).
   2. Select defaults for memory (4GB recommended) and disk space (~50 GB recommended). Use defaults for everything else.
   3. Save the machine.
4. Update Settings
   1. elect the CFS machine you just created and then select “settings” from the main menu.
   2. Under “System”->Processor set the number of cores to run
   3. Under Display set the video memory (I use 32MB) and 1 monitor
   4. Under “Storage” load the Ubuntu .iso file in the virtual “CD-ROM drive”.
      1. In the ”Storage Tree” area under “Controller: IDE” select the icon of a disc.
      2. On the right “Attributes” pane, next to Optical Drive, select the little icon of a disc there, then “Choose Virtual optical Disk File” and select the Ubuntu .iso file.
      3. Hit OK.
5. Turn on the machine
   1. With the CFS machine selected, press the “start” button in Virtualbox.
   2. Select “Install Ubuntu” button.
   3. Select “Download updates while installing Ubuntu”
   4. Select “Erase disk and install Ubuntu” -> don’t worry, this is just the virtual machine. The hit “Install Now”, then Continue.
   5. Select timezone and keyboard.
   6. Select username: cfs
   7. Choose password: cfs\_password
   8. Hit “continue”
   9. Wait… eventually reboot. VirtualBox automatically removes the install medium so you can just hit enter when prompted.

# Install Tools

1. Open a terminal in Ubuntu
2. Go to your home directory (~/)
3. Install GIT
   1. **sudo apt-get install git**
4. Install some build libraries
   1. **sudo apt-get install g++-multilib**
5. Install CURL (needed for COSMOS)
   1. **sudo apt-get install curl**

# Clone the Project from GitHub

1. Clone the project GIT repo.
   1. **git clone** [**https://github.com/NasaDtn/spacesystems\_2018.git**](https://github.com/NasaDtn/spacesystems_2018.git)
2. Change into the new spacesystems\_2018 directory
   1. **cd spacesystems\_2018**

# Build and Configure the CFS Image

The NASA Goddard Space Flight Center GSFC) has released, open source, the Core Flight System (cFS) which comprises an Operating System Abstraction Layer (OSAL), an underlying execution infrastructure – the Core Flight Executive (cFE), and a series of applications that, together, make up a Command and Data Handling (C&DH) flight system. This step will compile the OSAL, cFE, and cFS in a docker container and run the software to make sure everything is working correctly.

1. Set up the CFS environment
   1. **cd cfs**
   2. **. ./setvars.sh (Note the first . is not a typo, it is . ./setvars.sh NOT ./setvars.sh)**
2. Build the CFS code
   1. Change to the build directory: **cd /build/cpu1**
   2. Configure the build: **make config**
   3. Build the OSAL, cFE, and CFS apps: **make**
   4. Wait while it all compiles. Look for “>>> DONE! <<<” at the end.
3. Make sure that CFS runs successfully on your platform.
   1. From the cpu1 directory (that you just built in) go to the exe folder: **cd exe**
   2. Run CFS: **sudo ./core-linux.bin --reset PO**
   3. Wait until you see the “Stop FLYWHEEL” message.
   4. Type CTRL-C to stop CFS and return to the command prompt.

You now have an out-of-the-box spacecraft Command and Data Handling flight software system!

# Install and configure the Ground Command and Control Software

Now that we have a flight system, we need some way to communicate with that system by issuing commands and receiving back telemetry. For this project we will use the COSMOS open-source ground system.

1. Change to your home directory
   1. **cd**
2. Run the Ubuntu Linux COSMOS install script from <http://cosmosrb.com/docs/installation/)>
   1. It should look something like:
      1. bash <(\curl -sSL https://raw.githubusercontent.com/BallAerospace/COSMOS/master/vendor/installers/linux\_mac/INSTALL\_COSMOS.sh)
   2. When prompted, select s (for sudo) at the first prompt.
   3. Select Y for installing rbEnv (this will take a while to build things)
   4. If asked to overwrite the executable, select y.
   5. Select Y for install and run cosmos demo
      1. Select OK when COSMOS window pops up
      2. Close the Launcher by clicking X on the launcher window.
3. Update your path
   1. **source ~/.bashrc**
4. Create a project demo (this will create a demo in ~/cfs\_demo)
   1. **cosmos demo cfs\_demo**
5. Replace the config directory from cfs\_demo with the config directory from the spacesystems area
   1. **rm -rf ~/cfs\_demo/config**
   2. **cp -R ~/spacesystems\_2018/config ~/cfs\_demo**
6. Start COSMOS
   1. **cd ~/cfs\_demo**
   2. **ruby Launcher**
   3. **Select “Update Project CRCs” from the splash screen then OK on the pop-up**
   4. **Select OK to close the splash screen**

# Commanding Flight Software

1. In a new terminal window start CFS
   1. Remember run . ./setvars.sh
2. Start COSMOS
   1. Remember, ruby Launcher from within your cfs\_demo directory
3. Open the Command And Telemetry Server
4. Open the Command Sender (select OK to pop-up)
5. Open the Telemetry Viewer
   1. Select show screen next to the WHE entry
6. Enable TO Output
   1. Select the TO\_OUTPUT\_ENABLE command from the Command Sender
   2. Set the last argument (current a long string) to the value “127.0.0.1”
   3. Send the command
   4. Watch telemetry stream in on the telemetry viewer screen

# Refreshing from GitHub

Sometimes, I will publish new bits of code, updated instructions, and sample software. To refresh your local software, perform the following:

1. Go to your main directory that you cloned software into
   1. **cd ~/spacesystems\_2018**
2. “pull” new software into this area
   1. **git pull**
3. If any COSMOS config files changed, make sure to copy the config directory into your ground station directory.
   1. **cp -R ~/spacesystems\_2018/config ~/cfs\_demo/config**

That’s it. Your directories will now be refreshed with the latest set of information. Remember, if any of the COSMOS config files have changed,

# Adding Watchpoints and Actions

In class, we talked about autonomy in space systems being represented as an IF…..THEN condition. The way this is represented in the CFS flight software is with an application called the “Limit Checker” (LC). We will not be going into the full feature set of the LC app because we don’t need to in order to accomplish this practicum. So what follows is a useful but simplified version of how this all works.

In the LC app, the IF….THEN statement is formulated as:

IF Conditions THEN Actions

Where **Conditions** is a series of logical statements such as :

(Condition1 AND Condition2) OR (Condition 3)

And **Actions** is a command to run when the conditions evaluate to TRUE, such as:

SET\_CAP\_B\_ACTIVE

So, to use the limit check, we need to:

1. Define the individual conditions (Condition1, Condition2, etc…)
   1. The LC calls these conditions “watchpoints”
   2. A watchpoint is a data value, a threshold, and how long the threshold has to be violated in order for the condition to be considered active.
   3. In watchpoint terminology, if the threshold is violated the watchpoint is considered to have “failed”
2. Define the IF…THEN statement that matches conditions and actions.
   1. The LC calls these “action points”
   2. There is a logic expression that incorporates watchpoints
   3. There is a command to run when the logical expression is active.

Why separate watchpoints and action points? Flexibility and avoiding redundancy. For example, we can easily define 3 action points using combinations of just 2 watch points:

W1: The condition where “capacitor A charge > 97%”

W2: The condition where “capacitor B charge > 97%”

We can then define 3 action points:

IF (W1) THEN (SET\_CAP\_B\_ACTIVE)

IF (W2) THEN (SET\_CAP\_A\_ACTIVE)

IF ((W1) AND (W2)) THEN (PANIC)

## Adding Watchpoints

The LC App keeps watchpoints in a “table” file. The table is fully populated with entries (all but 1 of which are listed as “LC\_WATCH\_NOT\_USED” and labeled #0, #1, #2, and so on). You will edit unused entries to make your own watchpoints.

I have added a simple watchpoint which says “When Capacitor A Charge >= 3” and added it as watchpoint #0.

To edit this watchpoint, and add your own, follow these instructions:

* Change directory to where LC keeps the watchpoint table
  + **cd ~/spacesystems\_2018/cfs/apps/lc/fsw/tables**
* Edit the watchpoint table
  + **gedit lc\_def\_wdt.c**
* Verify that watchpoint #0 exists and is labelled “WHE Example”. If not, you need to exit gedit, pull the latest software from GITHUB (with the command below), and re-open in gedit.
  + **git pull**
* Update items #0, #1, #2, etc… with the watchpoints that you want. With the following cavats:
  + .DataType will ALWAYS be **LC\_DATA\_BYTE**.
  + .OperatorID will be the mathematical operator to use (See Table 1)
    - In watchpoint #0 we use LC\_OPER\_GE which is the comparison “greater than or equal to” (>=).
  + .MessageID will ALWAYS be **WHE\_HK\_TLM\_MID**
  + .WatchpointOffset will be the offset of the telemetry point (See Table 2)
    - In watchpoint #0, offset 14 is “Capacitor A charge”
  + .BitMask is ALWAYS **LC\_NO\_BITMASK**
  + .CustomFuncArgument is ALWAYS **0**
  + .ResultAgeWhenStale is ALWAYS **0**
  + .ComparisonValue.Unsigned32is the value you want to compare to.
    - In watchpoint #0 this is 3 (meaning 3% charge)
* When you are finished editing, save your file and exit.

You now have defined some watchpoints in the software! Now, to go make action points!

## Adding Action Points

The LC App keeps watchpoints in a “table” file. The table is fully populated with entries (all but 1 of which are listed as “LC\_ACTION\_NOT\_USED” and labeled #0, #1, #2, and so on). You will edit unused entries to make your own action points.

I have added a simple actionpoint which says “discharge capacitor A if watchpoint 0 triggers”. Recall from above that we had set watchpoint 0 to trigger when capacitor A charge was > 3%. I used action point #1 (AP1) for this to show that there is no hard-coding of action points IDs and watch point IDs. AP1 can use WP0 and so on…

To edit my actionpoint, and add your own, follow these instructions:

* Make sure you are still in the tables directory:
  + **cd ~/spacesystems\_2018/cfs/apps/lc/fsw/tables**
* Edit the actionpoint table
  + **gedit lc\_def\_adt.c**
* Verify that actionpoint #1 exists and is labelled “WHE Discharge Capacitor A”. If not, you need to exit gedit, pull the latest software from GITHUB (with the command below), and re-open in gedit.
  + **git pull**
* Update items #0, #1, #2, etc… with the actionpoints that you want. With the following cavats:
  + .DefaultState will ALWAYS be **LC\_APSTATE\_ACTIVE**.
  + .MaxPassiveEvents will ALWAYS be 0.
  + .MaxPassFailEvents will ALWAYS be 0.
  + .MaxFailPassEvents will ALWAYS be 0.
  + .RTSId will be the action to take (See Table 3)
    - For my example, the action is to discharge capacitor A so we use WHE\_CAP\_A\_DISCHARGE\_CC.
  + .MaxFailsBeforeRTS will be the number of seconds the conditions should persist before running the action. LC uses the term “fail” to mean “the RPN equation says to run the action”, which can be a bit confusing at first.
    - This MUST be a value > 0
    - For my action, I wanted the action to run the second it was detected, so I used “1”.
    - If you wanted to wait until the next second, you could use the value “2” and so on.
  + .EventType will ALWAYS be **CFE\_EVS\_INFORMATION**
  + .EventID should be 1000 + your action point ID
    - In my example, my action point is #1, so I used 1001.
  + .EventText is the text you want displayed to the user on the CFS screen.
    - In my example, I used “Discharge Capacitor A”
  + .RPNEquation is the RPN equation that describes the conditions to determine whether to run the action.
    - The last item MUST be LC\_RPN\_EQUAL.
    - If you simply want to run the action if a watchpoint triggers, you just put the number of the watchpoint.
      * For my example, I want to run the action if watch point 0 triggers, so I just put 0 there.
    - You can use combinations of watchpoints using logical operators (See Table 4).
    - It is possible that your rules NEVER need a complex expression and ALWAYS just take the action whenever the watchpoint triggers. If you feel you need something more complex, read the next section on RPN notation.
* When you are finished editing, save your file and exit.

Now you have action points!

# Re-build the flight software!

Just like before, go to the correct directory and build the software.

1. Build the CFS code
   1. Change to the CFS directory: **cd ~/spacesystems\_2018/cfs**
   2. Set environment variables: **. ./setvar.sh**
   3. Change to the build directory: **cd build/cpu1**
   4. Configure the build: **make config**
   5. Build the OSAL, cFE, and CFS apps: **make**
   6. Wait while it all compiles. Look for “>>> DONE! <<<” at the end.
2. Make sure that CFS runs successfully on your platform.
   1. From the cpu1 directory (that you just built in) go to the exe folder: **cd exe**
   2. Run CFS: **sudo ./core-linux.bin –reset PO**
   3. Wait until you see the “Stop FLYWHEEL” message.
   4. Type CTRL-C to stop CFS and return to the command prompt.

## Postfix (Reverse Polish) Notation

Everyone has probably seen Facebook quizzes relating to order-of-operations such as:

3+5\*6 = ?

Now, we know that multiplication comes before addition, so the answer here is 33 (not 48). One way to clarify order-of-operations is to use parenthesis so as to not make mistakes:

3+(5\*6) = ?

This notation is called INFIX notation because the operators (+, \*) are ”in order” with their operands. Doing this is great for people to read, but actually takes up a lot of memory and processing space on embedded computers. So, instead, we use POSTFIX notation (operators come AFTER their operands). So, we could write our original equation in postfix as:

5 6 \* 3 + = ?

Aside from making us sound like Yoda when we say that out loud, POSTFIX gets rid of the need for parenthesis – order of operation is encoded in the expression now. The process of evaluating goes something like this:

Step 1: Grab 5, then grab 6, then grab the operator \*

Step 2: Multiple 5 \* 6 to get 30

Step 3: Replace 5 6 \* with its result 30, so instead of “5 6 \* 3 +” we have “30 3 +”

Step 4: Grab 30, then grab 3, then grab the operator +

Step 5: Add 30 and 3 to get 33

Step 6: Replace 30 3 + with its result 33, so instead of “30 3 +” we have 33

Step 7: If all we have is a single number left, that must be the answer. 33!

POSTFIX is usually called “reverse polish notation” or RPN. The Wikipedia page for “Reverse Polish Notation” is excellent if you want to learn more.

CFS uses POSTFIX (RPN) notation when evaluating watch points. In this case a watchpoint is considered “TRUE” if it has triggered and “FALSE” if it has not triggered. As simple TRUE/FALSE statement, action points are not adding and multiplying, they use logical operators such as AND, OR, XOR, NOT, and EQUALS.

So, let’s say we want an action to be run if watchpoint0 (WP0) triggers AND watchpoint 1(WP1) has not triggered OR whenever watchpoint 3 (WP3) triggers.

Using INFIX we would write:

(WP0 AND !(WP1)) OR WP3

In POSTFIX we would write:

WP1 ! WP0 AND WP3 OR

And in the ADT table watchpoints are represented by a simple number, and the expression must ALWAYS end with LC\_RPN\_EQUAL so, we would represent the above as:

1, LC\_RPN\_NOT, 0, LC\_RPN\_AND, 3, LC\_RPN\_OR, LC\_RPN\_EQUAL

(There are plenty of infix to postfix online converters to use if this gives you a headache).

# Appendix A – Watch and Action Point Table Constants

Table - LC Watchpoint Operators.

|  |  |
| --- | --- |
| Operator | Value in Watch Table |
| < | LC\_OPER\_LT |
| <= | LC\_OPER\_LE |
| != | LC\_OPER\_NE |
| == | LC\_OPER\_EQ |
| >= | LC\_OPER\_GE |
| > | LC\_OPER\_GT |

Table - WHE Telemetry points, offsets, and values.

|  |  |  |
| --- | --- | --- |
| Telemetry Data Point | OFFSET | Values |
| Capacitor A charge | 14 | 0-255. Represents % of safe charge. |
| Capacitor A State | 15 | 0: Charging. 1: Discharging. 2:Leaking. 3:Broken |
| Capacitor B Charge | 16 | 0-255. Represents % of safe charge. |
| Capacitor B State | 17 | 0: Charging. 1: Discharging. 2:Leaking. 3:Broken |
| SBC State | 18 | 0: Off. 1: Powered. 2: Observing. 3:Error |
| TEMP as integer | 19 | 0-255. Temperature as an integer (truncated) e.g., 15.2 will be reported at 15. 15.9 will also be reported at 15. |
| Louver State | 20 | 0: Closed. 1: Open |
| Heater State | 21 | 0: Off. 1: On |
| Active Capacitor | 22 | 0: A. 1: B. |
| Damage State | 23 | 0: None. 1: Minor. 2: Major |

Table - Actions for WHE.

|  |  |
| --- | --- |
| WHE\_NOOP\_CC | Perform a no-op (useless, really) |
| WHE\_CAP\_A\_ACTIVE\_CC | Make capacitor A the active capacitor |
| WHE\_CAP\_A\_DISCHARGE\_CC | Discharge capacitor A |
| WHE\_CAP\_B\_ACTIVE\_CC | Make capacitor B the active capacitor |
| WHE\_CAP\_B\_DISCHARGE\_CC | Discharge capacitor B |
| WHE\_OBS\_START\_CC | Start an observations |
| WHE\_OBS\_STOP\_CC | Stop an observation |
| WHE\_POWER\_SBC\_CC | Turn on power to the SBC. |
| WHE\_THERM\_HTR\_OFF\_CC | Turn off the heater |
| WHE\_THERM\_HTR\_ON\_CC | Turn on the heater |
| WHE\_THERM\_LOUVER\_CLOSE\_CC | Close the Louver |
| WHE\_THERM\_LOUVER\_OPEN\_CC | Open the Louver |
| WHE\_TLM\_RESET\_CNTS\_CC | Reset telemetry counts |

Table - Action Point RPN Equation Operators

|  |  |
| --- | --- |
| LC\_RPN\_AND | Logical AND |
| LC\_RPN\_OR | Logical OR |
| LC\_RPN\_XOR | Logical Exclusive OR |
| LC\_RPN\_NOT | Negation |
| LC\_RPN\_EQUAL | Equality |