EPICS Exercises using EtherCAT

Note: A test control box will be available, for use with an EtherCAT station. The station consists of a BECKHOFF EK1100 coupler connected to an ES3104 Analogue Input terminal, an EL4134 Analogue Output terminal, an ES1014 Digital Input terminal and an EL2024 Digital Output terminal. Details of EtherCAT I/O addressing are given in the examples below.

For each question, we want to end up with:

1. A new database file (".db") file (e.g. *exerciseApp/Db/exercise1.db*)
2. A new GUI screen (".edl") file (e.g. *exerciseApp/opi/edl/exercise1.edl*)
3. A new IOC startup script (".src") file (e.g. *iocBoot/iocexercise/stexercise1.src*)
4. A new bash script to start the GUI (e.g. *exerciseApp/opi/edl/startgui1*)

At the end of the course, the goal is to end up with a complete set of these files for each question. That way, you can go back to any of the IOC's you have developed, at any time.

Note: It is **very important** to develop and keep these files in the correct directory within the development area file system structure, otherwise things will get very confused, very quickly!

After developing the files, you need to type "make" to generate the version which you will run.
From the top level of the application directory:

- To run the IOC, type: ./bin/linux-x86_64/stexercise1.boot (for Exercise 1)
- To run the GUI, type: ./data/startgui1 (for Exercise 1)

Exercise 1 – EDM (Graphical User Interface)

1. Edit the supplied startup IOC script (called *stexercise1.src* in the *iocBoot/iocexercise* directory) to run an EPICS IOC on your workstation. This loads the database *exercise1.db*, the source of which is provided in your *exerciseApp/Db* directory. Define the same, unique (to you), prefix to substitute for *$(user)* in the dbLoadRecords command. We suggest you use your Fed ID as the prefix. This will ensure that the names of the EPICS records in your database are unique. The prefix string *$(user)* is defined *without* the final '**:**' separator. After editing the IOC startup script, type "make".
2. Now start the IOC: *./bin/linux-x86_64/stexercise1.boot* from the top of the application tree.
3. Use the IOC console command "dbl" to list the names of the records in your loaded database.
4. The database file, *exercise1.db*, includes the following records (substitute your own prefix for *$(user)*):
    a. An MBBO record, called *$(user):freqMenu*, which allows the user to select one of four predefined frequencies for a sine wave.
    b. An AO record, called *$(user):offset*, which allows the user to define a fixed additive offset to the value of the sine wave.
    c. An AO record, called *$(user):amplitude*, which allows the user to define a multiplicative factor by which to change the amplitude of the sine wave.
    d. A CALC record, called *$(user):function*, which produces the output value of the sine wave every 100ms.
5. Using EDM, create a graphical user interface file: *exerciseApp/opi/edl/exercise1.edl* which enables interaction with the above records. The frequency should be selectable using a Menu Button object. The offset and amplitude records should be connected to Text Control objects and the output from the sinewave record should be displayed in a Stripchart object (check the "CA Time" option next to the Display PV entry). Type "make".
6. Edit the supplied GUI startup bash script: *exerciseApp/opi/edl/startgui1* to set the "*user=*" macro in here. It should be set to the same as the macro used for the IOC startup script. Type "make".
7. Start the GUI: : *./data/startgui1* from the top of the application tree.

Exercise 2 – Simple Record Processing

Inside the *exerciseApp/Db* directory, create a simple database, *exercise2.db*, using Visual DCT. When starting VDCT on a new database, you must specify the command like this:
vdct exercise2.db ../../dbd/exercise.dbd from inside: *exerciseApp/Db*

Your new database should contain the following 3 pairs of records:

1. A CALC record that performs an arithmetical operation on the value of its input field A, which must be obtained via an input link from another record. The CALC record should process, and its output value updated, only when a new value is input.

2. Another, similar pair of records, differing in that the CALC record should have its SCAN field set so that it is processed once every 2 seconds.

3. Another, similar pair of records, differing only in that the CALC record should be "Passive" and that it will be processed 5 times a second.

Don't forget the *$(user)* prefix on all your record names as in Exercise 1.
Load the database into an IOC and demonstrate the use of the records with a graphical display created with EDM (create a file called: *exerciseApp/opi/edl/exercise2.edl*). Don't forget to create a *startgui2* script for this example. Copy the one from Exercise 1 and edit as necessary.

Exercise 3 – Database Development

Design a database, *exerciseApp/Db/exercise3.db*, using Visual DCT, which performs the following functions of a temperature monitoring system:

1. Read in a 'temperature' value, periodically, from a sensor which is interfaced to the EtherCAT ES3104 analogue input terminal. The associated analogue input record should have its **DTYP** field set to: asynInt32. The **INP** field should be specified as: *@asyn(ERIO.1)AIStandardChannel1.Value*. Scale the data (use EGUL/EGUF) to display the 'temperature' in the range 0-100 degrees Centigrade. Note: the left-most dial on the test box is connected to this signal and you should be able to change the value of the temperature with this. Make sure the left-most "*SELECT METER DISPLAY*" switch is set to "AN. I/P 0".
2. When the temperature exceeds a set limit (HIHI limit), we need to make sure a MAJOR alarm will be raised.
3. The temperature limit value should be stored in a separate record.
4. A temperature alarm should cause an interlock to be set by writing "1" to an address on the EtherCAT EL2024 digital output terminal. The **DTYP** of the binary output record should be set to: *asynInt32*. The record's **OUT** field should be specified as: *@asyn(ERIO.4)Channel1.Output*. When the temperature falls below the alarm limit, the interlock should not be cleared automatically, but a separate method must be used to write a "0" to the digital output terminal. Define the two state strings in the EPICS record to be "SET" and "CLEARED". Note: the top-left LED, on the EL2024 terminal, is connected to this signal. You should be able to make it light-up and go off.
5. Add simulation to your own database, by using the fields *SIOL & SIML* in the analogue input record discussed in (1) above. In simulation mode, we do NOT want to write the interlock value to the hardware? How can we achieve this? You will need two further records:
    a. One to hold the simulation state. This should have a choice of two values: *HARDWARE or SIMULATED*.
    b. One to hold a simulated value for the temperature.

Don't forget to set the *$(user)* macro in the IOC startup script.

Using EDM, design a graphical user interface *exerciseApp/opi/edl/exercise3.edl* containing the following objects:

1. ..... A text entry object to allow the simulated temperature to be input.
2. ..... A menu button allowing the user to choose between hardware and simulated.
3. ..... A text entry object to allow the temperature alarm limit to be set.
4. ..... A text update object to monitor the temperature alarm severity.
5. ..... A text update object to monitor the current temperature.
6. ..... A text update object to monitor the status of the interlock.
7. ..... Add an EDM macro to this GUI for sensor number, so that you can distinguish one sensor from another if you have multiple sensors (see question 10)!

Create a GUI startup script: *startgui3*. Copy *startgui1* and edit as necessary, you'll need to specify the macro for the sensor number!

Using this interface, convince yourself that the database works with the EtherCAT hardware and with the simulation in your database. Check that the interlock is not written to the hardware when in simulation mode.

Exercise 4 – Database Development

An instrument includes a linear filter slide which is to be controlled using EPICS. The slide contains 4 filters but only one filter can be used at a time. A binary switch is located at each filter position to sense whether that particular filter is in or out of the beam. When the filter is in the beam, the switch is set LOW (0). The four switch outputs are interfaced to the EtherCAT digital input terminal, ES1014. The appropriate **DTYP** setting for a record reading from this terminal is: *asynInt32*.

Assume the following filters are located in the slide: RED, BLUE, ND1 and ND2, corresponding to switches 1 to 4 respectively. The input from switch 1 is addressed from the **INP** link of an EPICS record by: *@asyn(ERIO.3)Channel1.Input*. The input from switch 2 is addressed as: *@asyn(ERIO.3)Channel2.Input* etc.

1. Use VDCT to create database: *exerciseApp/Db/exercise4.db*. This database should contain 4 EPICS records to read the switches and transfer the values to a CALC record. The CALC record should perform a calculation on its inputs to produce the decimal equivalent of the bit pattern of the switches. This should then be read by an MBBI record (called *$(user):filter*) with its DTYP set to `Raw Soft Channel`. The MBBI record should provide the name of the filter which is currently in the beam. Include an alarm to indicate when the switch input data has an unexpected value (e.g. due to switch hardware or interface card malfunction).
   Note: Normally, we would use a single EPICS record (MBBI) to read all 4 channels at once. However, this is not supported for this particular EtherCAT terminal.

2. Create an EDM GUI for this database: *exerciseApp/opi/edl/exercise4.edl*. Add an object which will monitor the value and alarm severity.

3. Create a GUI startup script: *startgui4* by copying *startgui1* and editing. Create an IOC startup script: *stexercise4.src* by copying *stexercise1.src* and editing.

4. To demonstrate the use of the MBBI record with the EtherCAT ES1014 terminal, use the 4 switches mounted on-top of the terminal. These are connected to the channels in order 1-4 as we move from the top to the bottom of the terminal. You should be able to move between different filters by setting the switches appropriately and see this reflected on your EDM screen.

5. To demonstrate the use of the MBBI record in simulation, we will use the simulation fields of the record: *SIOL* and *SIML*, as in exercise 3. Create a second EPICS record: *$(user):simState* with state strings *"OFF"* and *"ON",* which will be used to control the simulation. Create a third EPICS record: *$(user):simSwitch* to hold the simulated switch value. Using *SIOL* and *SIML*, connect the simulation records to *$(user):filter*. Link *$(user):simSwitch* to *$(user):filter* so that when new simulated data is written to *$(user):simSwitch*, it is propagated to *$(user):filter*.

6. Add two further widgets to your EDM screen, one to allow the user to place simulated data in *$(user):simSwitch* and one to allow the user to switch the simulation on and off.

7. Add a fourth EPICS record: *$(user):simSelectFilter* to the database which:

a. Takes a filter name as input.
b. Will output the switch data value corresponding to the input filter name.
c. Sends its output to *$(user):simSwitch*.

8. Add an object to your EDM screen, connected to *$(user):simSelectFilter*, which will allow the user to select a filter name. You should now be able to select a filter, when simulation is ON, using *$(user):simSelectFilter* and see how this is correctly propagated to *$(user):filter* by looking at the object which monitors the output of *$(user):filter*.

Exercise 5 - StreamDevice

In this exercise, you will make use of the EPICS StreamDevice facility, the generic EPICS support for devices using TCP/IP or serial interfaces. You are asked to provide an IOC to communicate with a magnet power supply (Danfysik MPS model 9100).

For this exercise, we have a software simulation of the MPS 9100.

Please take your own copy of the simulator executable:

cp /dls_sw/work/R3.14.12.7/support/ajf67/EPICSExercises/streamsSimulator/MPS9100Sim .

./MPS9100Sim <port number>

The port number will be assigned to you during the training course.

The MPS 9100 protocol supports the commands shown in the table below (this is a subset of the commands supported by the device).

Every command sent to the device should be terminated with a Carriage Return character.

Every response message received from the device is terminated with a Line Feed character, followed by a Carriage Return character.

Please produce:

1. A protocol file listing each of the commands below.
2. A database file: *exerciseApp/Db/exercise5.db* containing sufficient records to demonstrate each of the commands below. Reading one of the DAC channels and acquiring the version information should be done every second.
3. An EDM GUI: *exerciseApp/opi/edl/exercise5.edl* which displays the version string and allows the user to write to one of the DAC channels. The current value of that DAC channel should be displayed along with the channel number. We would also like a Power Control button.
4. Don't forget to create an IOC startup script: *stexercise5.src* and an EDM startup script: *startgui5* for this exercise.

Command	Command Format	Response Format
Write to one of the 4 Digital to Analog convertors	"DA <chan> <val>" <chan> is a channel number in the range 1-4 <val> is a 6 digit signed integer value.	6 digit signed integer value (The return value can be ignored)
Read from one of the 4 Digital to Analog convertors	"DA <chan>" <chan> is a channel number in the range 1..4	<chan> <val>> where <val> is a 6 digit signed integer value
Power Control	"F" = Switch power off "N" = Switch power on	String "OK"
Firmware version Information	"VER"	Three lines, each containing 23 characters plus terminator (LF CR). Hint: redirect second two lines into 2 different records.

Exercise 6 – Custom/Subroutine Records

Let us assume that the linear slide of exercise 4 is driven by a motor under the control of a PID servo and we want to provide a simulation of the servo in case the real hardware is not available (just as it may not be on this training course)! The discrete form of the PID expression is as follows:

M(n) = P + I + D
P = KP * E(n)
I = KP * KI * SUMi (E(i) * dT(n))
D = KP * KD * ((E(n) – E(n-1)) / dT(n))

where:

M(n) = Change in value of manipulated variable at nth instant
P = Proportional Term
I = Integral Term
D = Derivative Term
KP = Proportional Gain
KI = Integral Gain
KD = Derivative Gain
E(n) = Error at nth sampling instant
SUMi = Sum from i=0 to i=n
dT(n) = Time difference between n-1 instance and nth instance

A 'C' function which implements the above controller has been supplied to you in: *exerciseApp/src/exercise6.c.* Attach the name of the function in this file to the SNAM field of an aSub record called *$(user):servo*. Copy *exerciseApp/Db/exercise4.db* to *exerciseApp/Db/exercise6.db.* Add the *$(user):servo* record to *exercise6.db*. The aSub record should be set to process at 10 Hz. Set the PREC field to 4, otherwise you will not see any decimal places. It should take 6 input values (using the input record fields shown) as follows (all input fields are type DOUBLE):

(i) The demanded position ("A")
(ii) The proportional gain ("B")
(iii) The integral gain ("C")
(iv) The derivative gain ("D")
(v) The position tolerance ("E")
(vi) The velocity tolerance ("F")

It should provide 3 output values (using the output record fields shown) as follows:

(i) The current position ("VALA"). Type=DOUBLE
(ii) The switch data appropriate to the current position ("VALB"). Type=LONG
(iii) The status of the slide. This can be STOPPED or MOVING ("VALC"). Type=STRING.

It is assumed that the four filters: RED, BLUE, ND1 and ND2 are located at positions 75, 150, 225 and 300 mm respectively. The servo code calculates the status of the slide to be STOPPED, when the current position is within the given position tolerance **and** the current velocity is within the given velocity tolerance.

Feed the switch data output to *$(user):simSwitch*. Add an EPICS record: *$(user):controlFilter* which will allow you to choose a filter by name and will output the position of that filter. (Tip: This record must have its DTYP field set to `"Raw Soft Channel"`). Feed the output from this record to the location within the aSub which holds the demanded position.

Create an EDM interface screen: *exerciseApp/opi/edm/exercise6.edl* which contains the following objects:

(i) A text entry object for the demanded position.
(ii) A text entry object for the proportional gain.
(iii) A text entry object for the integral gain.
(iv) A text entry object for the derivative gain.
(v) A text entry object for the position tolerance.
(vi) A text entry object for the velocity tolerance.
(vii) A text update object which monitors the current switch data.
(viii) A text update object which monitors the current status of the slide.
(ix) A text update object which monitors the current position of the slide.
(x) A stripchart object which monitors the current position (make this quite large).
(xi) A text update object which monitors the output from *$(user):filter*.
(xii) An object, connected to *$(user):controlFilter,* which allows you to select a filter by name.

Don't forget to create an IOC startup script: *stexercise6.src* and an EDM startup script: *startgui6* for this exercise.

Use your interface screen to set values for the position and velocity tolerance.
Try these values for proportional, integral and derivative gains: KP = 0.005, KI = 20, KD =2.
You will be able to achieve a better control loop with just a proportional controller, but then the trajectory will not be as interesting! Select filters and watch the action of the servo.

Exercise 7 – State Notation Language

Copy *exerciseApp/Db/exercise6.db* to *exerciseApp/Db/exercise7.db*.
Disconnect *$(user):controlFilter* and *$(user):servo*. Write a sequence program in: *exerciseApp/src/exercise7.stt* that will monitor *$(user):controlFilter* so that when a new filter is selected by name, the filter slide is driven to the position for that filter.

Edit *exerciseApp/src/Makefile* and uncomment the lines containing *exercise7.stt* and *exercise7.dbd*.
Create *iocBoot/iocexercise/stexercise7.src* from *iocBoot/iocexercise/stexercise1.src.*
Modify the *dbLoadRecords* line to load the new database.
Uncomment the line which starts the sequence program at the bottom of the file.

Start the IOC and demonstrate that it is your sequence program that is driving the movement of the filter by doing the following at the IOC console:

1. Issue the *seqShow* command to obtain the *task_ID* of the sequence program.

2. Issue *seqStop <task ID>*. This stops the sequence program. After this, further selection of a filter should cause no movement.

3. Now start the sequence program again with:
   Seq &controlFilter, “user=<Fed ID>”
   After this, the filter slide should move again.

Copy *exerciseApp/opi/edl/exercise6.edl* to *exerciseApp/opi/edl/exercise7.edl.* Use this EDM screen for this question.

Exercise 8 – Motor Support

Setup a motion IOC to communicate with the Python PMAC simulator.
To start the Python PMAC simulator:

```
# Setup your virtual python environment and activate it
virtualenv --no-site-packages -p /usr/bin/python venv27
source venv27/bin/activate

# Install npyscreen which is required for the simulator
pip install npyscreen

# Run the simulator
python /dls_sw/work/R3.14.12.7/support/ajf67/EPICSExercises/newPmacSim/SimPMAC.py

# On the Python GUI, choose a port number. This will be given to you on the training course.
```

Create *exerciseApp/Db/exercise8.db* to contain one motor record. The motor will be attached to axis 1 on the simulator. It has a maximum velocity of 20 mm/s and a resolution of 0.001mm/cnt. We would normally drive it at 15 mm/s. The acceleration time is 0.2 seconds. The dial limits are +200mm and -200mm. We want to see a major alarm if either of these are exceeded.

The DTYP field of the motor record should be set to: *asynMotor* and the OUT field should be set to: *@asyn(<str>,<m>)* where <str> is the same as:
pmacCreateController("<str>", ...) in your startup script and <m> is the axis number on the controller.

Create an EDM GUI: *exerciseApp/opi/edl/exercise8.edl* containing a related display widget which is connected to "motor.edl". Create a GUI startup script: *startgui8*. In this case, we must set the EDM macro to be: "motor=FedID:axis1".

Create a suitable startup script: *iocBoot/iocexercise/stexercise8.src* for the IOC. The motor record presentation will contain a good example!

It is later discovered that this axis is suffering from backlash and best results are obtained if the demand position is always approached from the positive side of the desired position. Typically, we want backlash control to start 10mm away from the final desired position. During backlash control, we need to reduce the velocity to 1mm/s and decrease the acceleration time to 0.4 seconds. Using the motor record GUI, configure the record to achieve this.

Now start StripTool and connect to the "current position" field of the motor record. Drive the motor to a few different positions and watch the trajectory on StripTool. Is it behaving as specified by your backlash conditions?

Exercise 9 - areaDetector

(1) Start the IOC and GUI for the areaDetector instance <n> you have been assigned, in two separate terminal windows, like this:
cd /dls_sw/work/R3.14.12.7/support/ajf67/EPICSExercises/ADTraining/iocs/exUser<n>
./bin/linux-x86_64/stexUser<n>.sh
./bin/linux-x86_64/stexUser<n>-gui

(2) Check the ordering of the plugins (look at the Port Name and Input Array Port on each plugin). Check: Are they enabled/disabled?

(3) Start the detector and look at the three images with *mjpg*.

(4) Choose a black and white image and put a cross on the image. To do this, use the *over* plugin. Rectangle, Ellipse, Cross, will need the width of the lines setting to something other than 0 to become visible. Look in "Extended Options" for these settings. Also be careful of the colour of the cross, is it the same as the background colour?

(5) Change the *over* plugin to point at the *roi* plugin (*mjpg* should be pointing to *over*). Select a region of interest on the *roi* plugin. What happens to the cross?

(6) Now reverse the *over* and *roi* plugins i.e. make the *roi* plugin point to the *over* plugin (*mjpg* should be pointing to *roi*). What happens to the cross?

(7) Now put the *stat* plugin into the chain, pointing to the output from the camera/detector. Switch on centroid and profiles and look at the X and Y cursor profiles. By positioning the cross at the same position as the cursor, you should be able to see which strip of the image you are getting a profile on? "Profile at cursor Y" means: *the Y profile through the image in the column defined by the cursor X position* and vice-versa. Does the graph of the profile make sense? Note: the *stat* plugin only works on black and white images. Swap to the colour image and you will see no data on the profiles.

(8) Save some HDF images in all 3 capture modes:

1. **Single mode** – Set directory and file name. Use a directory off of your home directory to save the files in. Start/Stop do nothing in this mode. We control the saving by toggling "auto save" from "No" to "Yes" to "No". Remember to switch on "Auto increment" first. Make sure the detector is not running too fast, otherwise you will fill up the disk! An "Acq Period" of **one second maximum** is good!

2. Examine the images using "hdfview".
Type: module load hdfview
Type: hdfview <path to file> to load a file in the viewer.
Go to "entry/data/data"
Right click on last "data", choose "open as", tick "Image" and "Ok".

3. Switch to **“Capture” mode**. Set number of frames to capture, to say 10. Click Start. Watch the images being counted up. Then “Write” will write a single file containing all 10 images. Load the file in “hdfview” as before. Now, when you open, select “open as” and tick “image”, but also, set Height = dim1 and Width = dim2 before clicking Ok. Use the “yellow arrows” at the top, to display each image in the file.

4. Finally try **“Stream” mode**. In this mode, make sure “# Capture” is set to 0. Now press “start”, the images continue to be written to one file on disk until you press “Stop”. Collect a few and then examine again with “hdfview”. Make sure to press “Stop”!