LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY

LIGO Laboratory / LIGO Scientific Collaboration

LIGO-T1400200-v1 *LIGO* April 3, 2015

CDS Device Readout using EPICS Modbus

K. Thorne

Distribution of this document:

LIGO Scientific Collaboration

This is an internal working note of the LIGO Laboratory.

California Institute of Technology
LIGO Project – MS 18-34
1200 E. California Blvd.
Pasadena, CA 91125
Phone (626) 395-2129
Fax (626) 304-9834
E-mail: info@ligo.caltech.edu

Massachusetts Institute of Technology
LIGO Project – NW22-295
185 Albany St
Cambridge, MA 02139
Phone (617) 253-4824
Fax (617) 253-7014
E-mail: info@ligo.mit.edu

LIGO Hanford Observatory
P.O. Box 1970
Mail Stop S9-02
Richland WA 99352
Phone 509-372-8106
Fax 509-372-8137

LIGO Livingston Observatory
P.O. Box 940
Livingston, LA 70754
Phone 225-686-3100
Fax 225-686-7189

http://www.ligo.caltech.edu/

Contents

Figures

1 Introduction

The document provides examples of how to read out devices used by CDS with the EPICS Modbus package. The goal is to reduce the amount of work it takes CDS sys-admins to set up a new readout.

2 Scope

This document covers the readout of devices on a LIGO CDS network obeying the Modbus protocol to process variables (PVs) managed by an EPICS soft IOC. Specifically, it assumes the installation of an EPICS package modified for CDS (see EPICS Modifications for CDS) that provides standard locations to access the EPICS installation and its Modbus module. Development has also been restricted to Modbus over TCP/IP, but these methods should work on serial ports as well.

3 Overview

For aLIGO, the control systems are still using the EPICS software maintained by Argonne National Lab and others. Users have created modules to add to the base package to allow readout of various hardware devices. Of interest here is the Modbus module supported by Mark Rivers at the University of Chicago. This module "supports EPICS communications with PLCs and other devices via the Modbus protocol over TCP, serial RTU and serial ASCII links. This is built on the existing asynDriver module, also maintained by Mark Rivers, that provides generic support for asynchronous communications.

The LLO CDS group has found need to read out several devices (AC power monitoring, Acromag ADCs, etc.) that can communicate using the industry-standard Modbus protocol. This is especially useful for "slow controls" (rep rate of 1 Hz or less), that do not require precise timing. We have been able to leverage the support of the EPICS Modbus module to provide simple soft IOCs that can take only a couple text files to get up and running.

Three examples are shown, organized by degree of complexity

- AC power monitoring. Uses existing EPICS database templates in the Modbus module, so only two text files need to be customized for the application
- Beam-tube temperature. Uses custom EPICS database file to add calculations, etc.
- Cold-cathode gauge. Requires rebuilding Modbus module with additional C function (taken from existing LIGO vacuum equipment code)

4 Modbus module basics

The installation and use of the Modbus package is very well described in the "Modbus" module documentation, which we will not attempt to paraphrase here. The default installation path for CDS following the standard environment variables is

```
${EPICS_MODULES}/modbus
```

In the control-room systems, EPICS_MODULES is typically '/ligo/apps/ubuntu12/epics/modules'. Under this directory are the original source code, binaries, libraries, and database templates. The path to the standard modbusApp that is built when the package is installed becomes:

```
${EPICS_MODULES}/modbus/bin/${EPICS_HOST_ARCH}
```

For the control-room Linux systems, EPICS_HOST_ARCH is typically 'linux-x86_64'. This pre-built application can be used if no additional functions (c-code files) are needed. If they are, then a new modbusApp needs to be built incorporating the additional files. This is covered in the third example

5 Example A: AC power monitoring

Our AC power monitor vendor provided us with controllers that had a web interface with Java applets that could be used to provide reports. However, this was awkward, tedious and was not integrated with our existing data acquisition and archiving system (DAQ). We found that an add-on was available that would provide data over the Facilities LAN using Modbus over TCP/IP.

The data was simply voltages, currents in 32-bit floats. We weren't going to do any further processing of the values, so we used the pre-built templates for the EPICS database file.

We started by defining the location of the Modbus device to be read out. This is done by adding info to the IOC command file, as shown in Figure 1 below. This was all taken from the example command file, which we then customized.

We start with some epicsEnvSet commands to define the IOC name ('iocl0acmcb') and where to find the pre-build Modbus files using the environment variables.

We then customize the 'drvAsynIPPortConfigure' call from the 'asyn' package the 'modbus' package is built on. We use portName of "l0acmcb" (arbitrary), then the hostInfo of TCP address of 10.110.70.51 and IP port of 502 (default for Modbus). The remaining values are recommended for TCP connections.

Next to be customized is the 'modbusInterposeConfig' call. Here we include the portName from 'drvAsynIPPortConfigure', the linkType of 0, a timeout of 2000 msec, and default writeDelay as we are not writing anything out to the device.

```
# l0acmcb.cmd
epicsEnvSet("IOC","iocl0acmcb")

epicsEnvSet("ARCH","linux-x86_64")
epicsEnvSet("TOP","${EPICS_MODULES}/modbus")
epicsEnvSet("MDBTOP","${EPICS_MODULES}/modbus")
dbLoadDatabase("$(MDBTOP)/dbd/modbus.dbd")
modbus_registerRecordDeviceDriver(pdbbase)

# Use the following commands for TCP/IP
#drvAsynIPPortConfigure(const char *portName,
#                       const char *hostInfo,
#                       unsigned int priority,
#                       int noAutoConnect,
#                       int noProcessEos);
drvAsynIPPortConfigure("l0acmcb","10.110.70.51:502",0,0,1)

#modbusInterposeConfig(const char *portName,
#                      modbusLinkType linkType,
#                      int timeoutMsec,
#                      int writeDelayMsec)
modbusInterposeConfig("l0acmcb",0,2000,0)

#drvModbusAsynConfigure(portName,
#                       tcpPortName,
#                       slaveAddress,
#                       modbusFunction,
#                       modbusStartAddress,
#                       modbusLength,
#                       dataType,
#                       pollMsec,
#                       plcType);
# The Chiller Yard Building AC monitor has 8 voltages starting at Modbus offset
30000
# CB Mon is userID 154 (turns out to be the unit identifier)
# Function code is 4
# Start at Decimal address 0 (offset 30000 is the default start)
# Access 40 words - 20 32-bit floats.
# Modbus data type 8 (Float32 bigEndian)
# poll every 100 msec
drvModbusAsynConfigure("CBMon",   "l0acmcb",  154, 4,  0, 40,    8,  100,
"CBMon")

dbLoadTemplate("l0acmcb.substitutions")

iocInit
```

Figure 1: l0acmcb.cmd

The last customization in the command file is ‘drvModbusAsynConfigure’ which is used to specify each Modbus data chunk. We define a new portName to be used in putting the data into EPICS variables. The tcpPortName is to match that in the drvAsynIPPortConfigure call. The slaveAddress is set to the Modbus unit identifier (154) provided by the vendor. We specify a Modbus function of ‘read’ (4). It took a bit of testing to determine that the default Modbus start for the ‘read’ is 30000, so we have a startAddress of ‘0’. We set modbusLength to 40 (16-bit words), which is 20 32-bit words. The dataType is ‘8’ for Float32 bigEndian. We set the polling period to 100 msec. The pclType is made the same as the portName (but this is only used in some monitoring functions not covered here.

We then load the database template file with ‘dbLoadTemplate’, and initialize the IOC with ‘iocInit’

We define all the EPICS variables using the template substitutions file, shown in Figure 2.

```
file "${EPICS_MODULES}/modbus/db/aiFloat64.template" { pattern
{P,          R,           PORT,  OFFSET, DATA_TYPE,  HOPR,  LOPR,   PREC,   SCAN}
{L0:ACM-CB_,  ATOB_VOLTS,   CBMon,   0,    FLOAT32_BE, 10000, 0, 6, "I/O Intr"}
{L0:ACM-CB_,  BTOC_VOLTS,   CBMon,   2,    FLOAT32_BE, 10000, 0, 6, "I/O Intr"}
{L0:ACM-CB_,  CTOA_VOLTS,   CBMon,   4,    FLOAT32_BE, 10000, 0, 6, "I/O Intr"}
{L0:ACM-CB_,  LTOL_VOLTS,   CBMon,   6,    FLOAT32_BE, 10000, 0, 6, "I/O Intr"}
{L0:ACM-CB_,  ATON_VOLTS,   CBMon,   8,    FLOAT32_BE, 10000, 0, 6, "I/O Intr"}
{L0:ACM-CB_,  BTON_VOLTS,   CBMon,  10,    FLOAT32_BE, 10000, 0, 6, "I/O Intr"}
{L0:ACM-CB_,  CTON_VOLTS,   CBMon,  12,    FLOAT32_BE, 10000, 0, 6, "I/O Intr"}
{L0:ACM-CB_,  LTON_VOLTS,   CBMon,  14,    FLOAT32_BE, 10000, 0, 6, "I/O Intr"}
{L0:ACM-CB_,  A_AMPS,      CBMon,  16,     FLOAT32_BE, 10000, 0, 6, "I/O Intr"}
{L0:ACM-CB_,  B_AMPS,      CBMon,  18,     FLOAT32_BE, 10000, 0, 6, "I/O Intr"}
{L0:ACM-CB_,  C_AMPS,      CBMon,  20,     FLOAT32_BE, 10000, 0, 6, "I/O Intr"}
{L0:ACM-CB_,  AVG_AMPS,    CBMon,  22,     FLOAT32_BE, 10000, 0, 6, "I/O Intr"}
{L0:ACM-CB_,  A_PFCTR,    CBMon,  24,     FLOAT32_BE, 1, 0, 6, "I/O Intr"}
{L0:ACM-CB_,  B_PFCTR,    CBMon,  26,     FLOAT32_BE, 1, 0, 6, "I/O Intr"}
{L0:ACM-CB_,  C_PFCTR,    CBMon,  28,     FLOAT32_BE, 1, 0, 6, "I/O Intr"}
{L0:ACM-CB_,  AVG_PFCTR,  CBMon,  30,     FLOAT32_BE, 1, 0, 6, "I/O Intr"}
{L0:ACM-CB_,  A_KW,       CBMon,  32,     FLOAT32_BE, 10000, 0, 6, "I/O Intr"}
{L0:ACM-CB_,  B_KW,       CBMon,  34,     FLOAT32_BE, 10000, 0, 6, "I/O Intr"}
{L0:ACM-CB_,  C_KW,       CBMon,  36,     FLOAT32_BE, 10000, 0, 6, "I/O Intr"}
{L0:ACM-CB_,  SUM_KW,     CBMon,  38,     FLOAT32_BE, 10000, 0, 6, "I/O Intr"}
}
```

Figure 2: l0acmcb.subscriptions

The Modbus module in our EPICS installation already has a ‘modbusApp’ executable built. We merely need to run it, feeding it the EPICS command file ‘l0acmcb.cmd’. We set up a directory (‘/ligo/cds/llo/target/fmcsscript1/acm’) to hold the command and substitutions files. To run the app, we use a script to move to the command file directory and start ‘modbusApp’. For this, we use script ‘start_l0acmcb.sh’ in Figure 3 below.

```
#!/bin/sh
# start AC power monitoring at Chiller Yard building readout as window 3

. /ligo/cdscfg/stdenv.sh

tmux new-window -t acm:2 -n 'CB' 'cd
/ligo/cds/llo/target/fmcsscript1/acm/iocl0acmcb &&
${EPICS_MODULES}/modbus/bin/${EPICS_HOST_ARCH}/modbusApp l0acmcb.cmd'
```

Figure 3: script start_l0acmcb.sh

Note the use of 'tmux'. The EPICS IOCs require a terminal session to run, so we use 'tmux' for this. There are other solutions.

6 Example B: Beam-tube monitoring

The next example involves using Acromag ADCs to read out sensors in the beam-tube enclosure. In this case, we want to use our own EPICS database files so we can add in simple calculations from the raw data. This still uses the pre-built 'modbusApp', but with an expanded EPICS IOC command file and some EPICS database files.

We again started by defining the location of the Modbus device to be read out. This as done by adding info to the IOC command file, as shown in Figure 4 below. We again start with epicsEnvSet commands to define the IOC name ('iocl0vey24c') and where to find the pre-built Modbus files using the environment variables.

We then customized the 'drvAsynIPPortConfigure' call to use portName of "l0vey24c" (arbitrary), then the hostInfo of TCP address of 10.110.60:204 and IP port of 502 (default for Modbus). Next, we customized the 'modbusInterposeConfig' call to include the portName from 'drvAsynIPPortConfigure', the linkType of 0, a timeout of 5000 msec, and default writeDelay as we are not writing anything out to the device.

We customize 'drvModbusAsynConfigure' calls to specify each Modbus data chunk. In each case, we specify a Modbus function of 'read' (4). Remember that the Modbus start for the 'read' is 30000. For the 9 words of Status registers at Modbus offset 30000, we use a portName of 'Status_Reg', startAddress of '0', modbusLength of 9 (16-bit words) and dataType to '0' for 16-bit integer. We set the polling period to 1000 msec.

For the 6 words for ADC data as percentages at Modbus offset 30009, we use portName of 'Value_Reg', startAddress of '9', modbusLength of 6 and dataType to '0' for 16-bit integer. For the 6 words for ADC data as raw counts at Modbus offset 30016, we use portName of 'ADC_Reg', startAddress of '16', modbusLength of 6 and dataType to '0' for 16-bit integer.

The Acromag 961EN has 6 16-bit ADCs that operate on 4-20 mA current loop. The 961EN provides access to the ADC data as both percentage and raw counts. In this application, all the sensors return percentage data (0-100 %) that would cover the full 16mA range, so we will be using the percentage 'Value_Reg' registers.

We then load our custon database file with 'dbLoadTemplate', and initialize the IOC with 'iocInit'.

```
# l0vey24c.cmd
epicsEnvSet("IOC","iocl0vey24c")

epicsEnvSet("ARCH","linux-x86_64")
epicsEnvSet("TOP","${EPICS_MODULES}/modbus")
epicsEnvSet("MDBTOP","${EPICS_MODULES}/modbus")
dbLoadDatabase("$(MDBTOP)/dbd/modbus.dbd")
modbus_registerRecordDeviceDriver(pdbbase)

drvAsynIPPortConfigure("l0vey24c","10.110.60.204:502",0,0,1)

modbusInterposeConfig("l0vey24c",0,5000,0)

# The 961EN has bit access to the Status registers at Modbus offset 30000
# Access 9 words - 16 bits each (0000-0008).  Function code=4
drvModbusAsynConfigure("Status_Reg", "l0vey24c", 0, 4, 0, 9, 0, 1000,
"Acromag")

# The 961EN has word access (Percentage) to the ADCs at Modbus offset 30009
# Access 6 words (0009-000E) as inputs. Function code=4, Modbus type 4 (16-bit
int)
drvModbusAsynConfigure("Value_Reg", "l0vey24c", 0, 4, 9, 6, 4, 1000, "Acromag")

# The 961EN has word access (Raw counts) to the ADCs at Modbus offset 30016
# Access 6 words (000F-0014) as inputs. Function code=4, Modbus type 4 (16-bit
int).
drvModbusAsynConfigure("ADC_Reg", "l0vey24c", 0, 4, 15, 6, 4, 1000, "Acromag")

dbLoadDatabase("./LVE-BY_Y24C.db")

iocInit
```

Figure 4: l0vey24c.cmd

We then composed the EPICS database to access the data in the Value_Reg and do calculations. See file 'LVE-BY_Y24C.db' displayed in Figure 5 below. The first 'ai' record defines LVE-BY_BT_TEMP_PCT as using the first 'Value_Reg' quantity. This is done by customizing the 'DTYP' to 'asynInt32', and the 'INP' field to '@asynMask(Value_Reg 0 -16)MODBUS_DATA'. Here '-16' means 16-bit integer. The only tricky parts are the Engineering Unit limits EGUL, EGUF. The Acromag devices allow for offsets due to calibration by allowing a full scale of (-163.84, +163.835) to match 16-bit ranges. The real range that you can access is 0-100. Once I have done that, I added in normal EPICS database records to do calculations. I did similar entries for the dew-point percentage 'LVE-BY:Y24_AIR_DWP_PCT' and air temperature percentage 'LVE-BY:Y24_AIR_TEMP_PCT'.

```
record(ai,"LVE-BY:Y24_BT_TEMP_PCT")
{
      field(SCAN,"1 second")
      field(FLNK,"LVE-BY:Y24_BT_TEMP_C.PROC")
        field(DESC,"Beamtube Temp Percentage")
        field(DTYP,"asynInt32")
        field(INP,"@asynMask(Value_Reg 0 -16)MODBUS_DATA")
        field(EGU,"PCT")
        field(EGUL,"-163.84")
        field(EGUF,"163.835")
        field(LINR,"LINEAR")
        field(PREC,"2")
        field(HIHI,"100")
        field(HIGH,"99")
        field(LOW,"1")
        field(LOLO,"0")
        field(HHSV,"MAJOR")
        field(HSV,"MINOR")
        field(LSV,"MINOR")
        field(LLSV,"MAJOR")
}
record(calc,"LVE-BY:Y24_BT_TEMP_C")
{
      field(INPA,"LVE-BY:Y24_BT_TEMP_PCT.VAL NPP MS")
        field(DESC,"Convert Beamtube Temp pct to Deg C")
        field(CALC,"A-20")
        field(EGU,"DegC")
        field(PREC,"2")
        field(HOPR,"80")
        field(LOPR,"-20")
}
record(ai,"LVE-BY:Y24_AIR_DWPT_PCT")
{
      field(SCAN,"1 second")
      field(FLNK,"LVE-BY:Y24_AIR_DWPT_C.PROC")
        field(DESC,"Air Dewpoint Percentage")
        field(DTYP,"asynInt32")
        field(INP,"@asynMask(Value_Reg 1 -16)MODBUS_DATA")
        field(EGU,"PCT")
        field(EGUL,"-163.84")
        field(EGUF,"163.835")
        field(LINR,"LINEAR")
        field(PREC,"2")
        field(HIHI,"100")
        field(HIGH,"99")
        field(LOW,"1")
        field(LOLO,"0")
        field(HHSV,"MAJOR")
        field(HSV,"MINOR")
        field(LSV,"MINOR")
        field(LLSV,"MAJOR")
}
record(calc,"LVE-BY:Y24_AIR_DWPT_C")
{
      field(INPA,"LVE-BY:Y24_AIR_DWPT_PCT.VAL NPP MS")
        field(DESC,"Convert Dewpoint pct to Deg C")
        field(CALC,"A-20")
        field(EGU,"DegC")
```

```
        field(PREC,"2")
        field(HOPR,"80")
        field(LOPR,"-20")
}
record(ai,"LVE-BY:Y24_AIR_TEMP_PCT")
{
      field(SCAN,"1 second")
      field(FLNK,"LVE-BY:Y24_AIR_TEMP_C.PROC")
        field(DESC,"Air Temp Percentage")
        field(DTYP,"asynInt32")
        field(INP,"@asynMask(Value_Reg 2 -16)MODBUS_DATA")
        field(EGU,"PCT")
        field(EGUL,"-163.84")
        field(EGUF,"163.835")
        field(LINR,"LINEAR")
        field(PREC,"2")
        field(HIHI,"100")
        field(HIGH,"99")
        field(LOW,"1")
        field(LOLO,"0")
        field(HHSV,"MAJOR")
        field(HSV,"MINOR")
        field(LSV,"MINOR")
        field(LLSV,"MAJOR")
}
record(calc,"LVE-BY:Y24_AIR_TEMP_C")
{
      field(INPA,"LVE-BY:Y24_AIR_TEMP_PCT.VAL NPP MS")
        field(DESC,"Convert Temp pct to Deg C")
        field(CALC,"A-20")
        field(EGU,"DegC")
        field(PREC,"2")
        field(HOPR,"80")
        field(LOPR,"-20")
}
```

Figure 5: LVE-BY_Y24C.db

Once I had that, I created a start-up script ‘start_l0vey24c.sh’, as shown in Figure 6.

```
#!/bin/sh
# start beam-tube environment monitoring of l0vey24c

. /ligo/cdscfg/stdenv.sh

# running on l0pemmsr1 for now
tmux new-window -t vacuum:4 -n 'y24c' 'cd
/ligo/cds/llo/target/vescript1/bte/iocl0vey24c &&
${EPICS_MODULES}/modbus/bin/${EPICS_HOST_ARCH}/modbusApp l0vey24c.cmd'
```

Figure 6: start_l0vey24c.sh

7 Example C: Vacuum gauges

For our final example, we will again use an Acromag ADC with Modbus readout. This was done to replace the failed mid-station (MX) readout. Here, we want to invoke some C functions for doing the calculations. In this case, we have to break down and do a custom build of a 'modbusApp' including our C source code. We again use custom database files.

To build the Modbus EPICS application, download the tar-ball from the EPICS Modbus web-site and expand it in the directory of your choice. We then want to customize its configuration so it used the EPICS Base, sequencer and ASYN packages already installed. We do this by modifying the 'configure/RELEASE' file to define ASYN and EPICS_BASE using our environment variables (this follows the changes detailed in EPICS Modifications for CDS). We also want to follow the old CDS process of building the EPICS code in one directory, but installing it in another. To do that, we need to modify the file to define INSTALL_LOCATION. See Figure 7 below.

```
#RELEASE Location of external products
# Run "gnumake clean uninstall install" in the application
# top directory each time this file is changed.
#
# NOTE: The build does not check dependancies on files
# external to this application. Thus you should run
# "gnumake clean uninstall install" in the top directory
# each time EPICS_BASE, SNCSEQ, or any other external
# module defined in the RELEASE file is rebuilt.

TEMPLATE_TOP=$(EPICS_BASE)/templates/makeBaseApp/top

ASYN=${EPICS_MODULES}/asyn

# If you don't want to install into $(TOP) then
# define INSTALL_LOCATION_APP here
INSTALL_LOCATION_APP=/ligo/cds/llo/target/l0vemx

# EPICS_BASE usually appears last so other apps can override stuff:
EPICS_BASE=${EPICS_BASEDIR}
-include $(TOP)/../configure/EPICS_BASE.$(EPICS_HOST_ARCH)

#Capfast users may need the following definitions
#CAPFAST_TEMPLATES=
#SCH2EDIF_PATH=
```

Figure 7: configure/RELEASE

We also rename the 'iocTest' directory in 'iocBoot' to the more specific 'iocl0vemx'.

The EPICS code is not perfectly set up for installing in a different directory from the build. To enable that, we also modify 'iocBoot/iocl0vemx/Makefile' to add an install area to copy over all the files we will need post-build. See Figure 8 below

```
TOP = ../..
include $(TOP)/configure/CONFIG
ARCH = $(EPICS_HOST_ARCH)
TARGETS = envPaths cdCommands
include $(TOP)/configure/RULES.ioc

IOCDIR = $(INSTALL_LOCATION)/iocBoot/iocl0vemx

install:
      mkdir -p $(IOCDIR)
      cp *.db $(IOCDIR)
      cp *.cmd $(IOCDIR)
      cp *.substitutions $(IOCDIR)
      cp envPaths $(IOCDIR)
      cp cdCommands $(IOCDIR)
```

Figure 8: iocBoot/iocl0vemx/Makefile

We already use C code functions for the original VME-based EPICS vacuum controls to do proper scaling of the readout. To adapt this file, we first copied 'VE_AMPS_TO_TORR.c' to 'modbusApp/src/. We then only had to update the headers to match those from the newer EPICS Base (3.14.12.2) we are now using, and add calls to epicsRegisterFunction at the bottom. See Figure 9 below.

```
/*--------------------------------------------------------------------------*/
/*                                                                          */
/* Module Name: VE_AMPS_TO_TORR.c                                           */
/*                                                                          */
/* Module Description:  LIGO Vacuum Monitor/Control System                  */
/*                 Subroutine to convert current to Torr                    */
/*                       for gate valve annulus ion pump.                   */
/*--------------------------------------------------------------------------*/

#include <stdio.h>
#include <tgmath.h>

#include <dbDefs.h>
#include <registryFunction.h>
#include <subRecord.h>
#include <aSubRecord.h>
#include <epicsExport.h>

static long ampsTorrInit(subRecord *psub )
{
/*    printf( "ampsTorrInit was called\n" ); */
      psub->val = 0.0;
      return( 0 );
}

static long ampsTorrProcess(subRecord *psub)
{
      float y;
      float amps;
      float temp;
      float torr;
      float mAmps;
      const float SLOPE = 1.065;
      const float XREF = log10( 7.0E-10 );
      const float YREF = log10( 1.0E-7 );

      mAmps = psub->a;
      amps = mAmps / 1000;
      y = log10( amps );
      temp = 0.0;
      torr = 0.0;

      if ( amps > 1.0E-7 && amps < 1.0E-1 ){
          temp = XREF - ( ( YREF - y ) / SLOPE );
          torr = pow( 10.0, temp );
      }

      psub->val = torr;
      return( 0 );
}
epicsRegisterFunction(ampsTorrInit);
epicsRegisterFunction(ampsTorrProcess);
```

Figure 9: modbusApp/src/VE_AMPS_TO_TORR.c

To add our source code functions to the Modbus build, we had to created a DBD file ‘dbd/vecode.dbd’ that holds the function definitions, as shown in Figure 10 below.

```
function(ampsTorrInit)
function(ampsTorrProcess)
```

Figure 10: dbd/vecode.dbd

We then added three lines to the existing Modbus app Makefile ‘modbusApp/src/Makefile’ as shown in Figure 11 below. In the DBD section, we added the line ‘DBD += vecode.dbd’ to build the function definitions. In the LIB_SRC section, we added the line ‘LIB_SRCS += VE_AMPS_TO_TORR.c’ so the code file would be compiled. We also had to add to the PROD_SRCS_DEFAULT to ensure a Record Device Driver would be built from vecode.dbd.

```
# Makefile

TOP = ../..
include $(TOP)/configure/CONFIG

#----------------------------------------
#  ADD MACRO DEFINITIONS AFTER THIS LINE
#============================
#===================================================
# Build an IOC support library

# <name>.dbd will be created from <name>Include.dbd
DBD += modbus.dbd
DBD += modbusSupport.dbd
DBD += vecode.dbd

INC += drvModbusAsyn.h

LIBRARY_IOC = modbus

LIB_SRCS += drvModbusAsyn.c
LIB_SRCS += modbusInterpose.c
LIB_SRCS += VE_AMPS_TO_TORR.c

LIB_LIBS += asyn
LIB_LIBS += $(EPICS_BASE_IOC_LIBS)
LIB_SYS_LIBS_WIN32 += WS2_32

#============================

#Note that the command line that builds the
#library $(LIBNAME) may be HUGE (>3kB)

#============================
# build an ioc application

PROD_IOC = modbusApp

# <name>_registerRecordDeviceDriver.cpp will be created from <name>.dbd
PROD_SRCS_DEFAULT += modbus_registerRecordDeviceDriver.cpp modbusMain.cpp
PROD_SRCS_vxWorks += modbus_registerRecordDeviceDriver.cpp

# <name>_registerRecordDeviceDriver.cpp will be created from <name>.dbd
PROD_SRCS_DEFAULT += vecode_registerRecordDeviceDriver.cpp

PROD_LIBS += modbus
PROD_LIBS += asyn

PROD_LIBS += $(EPICS_BASE_IOC_LIBS)

#
include $(TOP)/configure/RULES
#----------------------------------------
#  ADD RULES AFTER THIS LINE
```

Figure 11: modbusApp/src/Makefile

We created another command file ‘iocBoot/iocl0vemx/l0vemx.cmd’ to load things. As shown in Figure 12 below, we added lines of ‘dbLoadDatabase’ and ‘registerRecordDeviceDriver’ to load vecode. The Acromag 968EN is an 8-channel DC voltage ADC and has a slightly different Modbus register definition than the 961EN. We will access the raw data registers this time, as we are digitizing logarithmic pressure read-backs. We then load the two EPICS database files that are defined below.

```
# l0vemx.cmd
epicsEnvSet("IOC","iocl0vemx")

epicsEnvSet("ARCH","linux-x86_64")
epicsEnvSet("TOP","/ligo/cds/llo/target/l0vemx")
dbLoadDatabase("../../dbd/modbus.dbd")
modbus_registerRecordDeviceDriver(pdbbase)

dbLoadDatabase("../../dbd/vecode.dbd")
vecode_registerRecordDeviceDriver(pdbbase)

drvAsynIPPortConfigure("l0vemx","10.110.60.65:502",0,0,1)

modbusInterposeConfig("l0vemx",0,2000,0)

# The 986EN has bit access to the Status registers at Modbus offset 30001
# Access 9 words - 16 bits each (0000-0009).  Function code=4
drvModbusAsynConfigure("Status_Reg", "l0vemx", 0, 4, 0, 9, 0, 100, "Acromag")

# The 986EN has word access to the ADCs at Modbus offset 30018
# Access 8 words (0011-0025) as inputs.  Function code=4, data type 16 bit int
drvModbusAsynConfigure("Adc_Reg",  "l0vemx", 0, 4, 17, 8,  4, 100,  "Acromag")

dbLoadDatabase("./LVE-MX_GV8.db")
dbLoadDatabase("./LVE-MX_X1.db")

iocInit
```

Figure 12: iocBoot/iocl0vemx/l0vemx.cmd

Now we create two EPICS database files ‘LVE-MX_X1.db’ and ‘LVE-MX_GV8.db’, also in ‘iocBoot/iocl0vemx’. These were derived from the older VME files. The first file (Figure 13) defines the cold-cathode gauge readout. Note that we scale the readouts (EGUL, EGUF) to -16.384, +16.384 even though the ADC range is set as -10V to +10V. Again, this is to accommodate the calibration feature of the device that allows for substantial offset correction. The second file (Figure 14) defines the annulus ion pump readout. The ‘LVE-MX:GV8_AIPTORR’ record shows how the C functions ‘ampsTorrInit’ and ‘ampsTorrProcess’ are used. The INAM field specifies the initialization function and SNAM the processing function for new values.

```
record(ai,"LVE-MX:X1_PT653B")
{
      field(SCAN,".5 second")
      field(FLNK,"LVE-MX:X1_653BTORR.PROC")
        field(DESC,"Cold Cathode Gauge Reading")
        field(DTYP,"asynInt32")
        field(INP,"@asynMask(Adc_Reg 1 -16)MODBUS_DATA")
        field(EGU,"Volts")
        field(EGUL,"-16.384")
        field(EGUF,"16.384")
        field(LINR,"LINEAR")
        field(PREC,"2")
        field(HIHI,"8")
        field(HIGH,"6")
        field(LOW,"1.5")
        field(LOLO,".5")
        field(HHSV,"MAJOR")
        field(HSV,"MINOR")
        field(LSV,"MINOR")
        field(LLSV,"MAJOR")
}
record(calc,"LVE-MX:X1_653BTORR")
{
      field(FLNK,"LVE-MX:X1_653BLOG.PROC")
      field(INPA,"LVE-MX:X1_PT653B.VAL  NPP MS")
        field(DESC,"Convert CC volts to torr")
        field(CALC,"((0<A)&&(A<=10))?10^(A-11):0")
        field(EGU,"Torr")
        field(PREC,"2")
        field(HOPR,"1.0e-3")
        field(LOPR,"3.0e-11")
}
record(calc,"LVE-MX:X1_653BLOG")
{
field(INPA,"LVE-MX:X1_653BTORR.VAL  NPP NMS")
        field(DESC,"Convert CC torr to log")
        field(CALC,"LOG(A)")
        field(EGU,"Log")
        field(PREC,"2")
        field(HOPR,"0")
        field(LOPR,"-10")
}
```

Figure 13: iocBoot/iocl0vemx/LVE-MX_X1.db

```
record(ai,"LVE-MX:GV8_II850")
{
      field(SCAN,".5 second")
      field(FLNK,"LVE-MX:GV8_850LOG.PROC ")
        field(DESC,"Annulus Ion Pump Current")
        field(DTYP,"asynInt32")
        field(INP,"@asynMask(Adc_Reg 0 -16)MODBUS_DATA")
        field(EGU,"mA")
        field(EGUL,"-16.384")
        field(EGUF,"16.384")
        field(LINR,"LINEAR")
        field(HOPR,"10")
        field(PREC,"4")
        field(LOPR,"0")
        field(HIGH,"10")
        field(HSV,"MAJOR")
        field(LOW,".001")
        field(LSV,"MAJOR")
}
record(calc,"LVE-MX:GV8_850LOG")
{
      field(FLNK,"LVE-MX:GV8_AIPINTLK.PROC ")
      field(INPA,"LVE-MX:GV8_II850.VAL  PP MS")
        field(DESC,"Convert mA to log")
        field(CALC,"0<A?LOG(A):-3")
        field(EGU,"Log")
        field(HOPR,"1")
        field(LOPR,"-5")
}
record(calc,"LVE-MX:GV8_AIPINTLK")
{
      field(FLNK,"LVE-MX:GV8_AIPTORR.PROC ")
      field(INPA,"LVE-MX:GV8_II850.VAL  NPP MS")
        field(DESC,"Annulus Ion Pump Current Interlock")
        field(CALC,"(0<A)&&(A<3.0)?1:0")
}
record(sub,"LVE-MX:GV8_AIPTORR")
{
      field(INPA,"LVE-MX:GV8_II850.VAL  NPP MS")
        field(PREC,"2")
        field(DESC,"Convert Annulus Ion Pump Current to Torr")
        field(INAM,"ampsTorrInit")
        field(SNAM,"ampsTorrProcess")
        field(EGU,"Torr")
        field(HOPR,"1.0e-04")
        field(LOPR,"1.0e-10")
}
```

Figure 14: iocBoot/iocl0vemx/LVE-MX_GV8.db

Now that all is in place, we can build the application by going to the top-level directory (here '/ligo/cds/projects/vacuum/l0vemx') and typing 'make' at the command line. If all goes well, the executables will be built and all the required files will be installed at our INSTALL_LOCATION (here '/ligo/cds/llo/target/l0vemx'). You should check that all the files you need have been copied to the installation directory. We again need a script to start this, so we created 'startup.sh' (see Figure 15) to go into to iocBoot/iocl0vemx directory and start the local 'modbusApp' using our command file. We invoke the CDS stdsetup script to ensure we find our EPICS installation.

```
#!/bin/bash

source /ligo/cdscfg/stdsetup.sh

cd /ligo/cds/llo/target/l0vemx

cd iocBoot/iocl0vemx

../../bin/linux-x86_64/modbusApp l0vemx.cmd
```

Figure 15: /ligo/cds/llo/target/l0vemx/startup.sh

8 Conclusion

Clearly this is just a start. For instance, in the LLO beam-tube enclosure (BTE), we now have multiple environment monitoring and vacuum gauge readout. Scripts were written to automate the installation of these. Each of the vacuum gauge readouts could use the same custom modbusApp, built with all possible vacuum conversion routines.

We have also found that the Acromag MODBUS devices support up to 10 different MODBUS readouts. This allowed our facility controls vendor to add a separate readout of the same humidity sensors into that system while maintain our independent readout.