

# Measurement Computing 1608GX-2A0 Driver

## Version 1

- 2 simple analog outputs
- 1 parameter, ANALOG\_OUT\_VALUE
- Use asynPortDriver C++ base class
- 131 lines of code

```

/* drvUSB1608G_V1.cpp
 *
 * Driver for Measurement Computing USB-1608G
 * multi-function DAQ board using asynPortDriver base class
 *
 * This version implements only simple analog outputs
 *
 * Mark Rivers
 * April 14, 2012
 */

#include <iocsh.h>
#include <epicsExport.h>
#include <asynPortDriver.h>

#include "cbw.h"

static const char *driverName = "USB1608G";

// Analog output parameters
#define analogOutValueString      "ANALOG_OUT_VALUE"

#define NUM_ANALOG_OUT    2    // Number of analog outputs on 1608G
#define MAX_SIGNALS      NUM_ANALOG_OUT

```

```

/** Class definition for the USB1608G class
 */
class USB1608G : public asynPortDriver {
public:
    USB1608G(const char *portName, int boardNum);

    /* These are the methods that we override from asynPortDriver */
    virtual asynStatus writeInt32(asynUser *pasynUser, epicsInt32 value);
    virtual asynStatus getBounds(asynUser *pasynUser, epicsInt32 *low,
                                epicsInt32 *high);

protected:
    // Analog output parameters
    int analogOutValue_;
    #define FIRST_USB1608G_PARAM analogOutValue_
    #define LAST_USB1608G_PARAM analogOutValue_

private:
    int boardNum_;
};

#define NUM_PARAMS (&LAST_USB1608G_PARAM - &FIRST_USB1608G_PARAM + 1)

```

```

/** Constructor for the USB1608G class
 */
USB1608G::USB1608G(const char *portName, int boardNum)
    : asynPortDriver(portName, MAX_SIGNALS, NUM_PARAMS,
        asynInt32Mask | asynDrvUserMask, // Interfaces that we implement
        0, // Interfaces that do callbacks
        ASYN_MULTIDEVICE | ASYN_CANBLOCK, 1,
        /* ASYN_CANBLOCK=1, ASYN_MULTIDEVICE=1, autoConnect=1 */
        0, 0), /* Default priority and stack size */
        boardNum_(boardNum)
{
    // Analog output parameters
    createParam(analogOutValueString, asynParamInt32, &analogOutValue_);
}

```

```
asynStatus USB1608G::getBounds(asynUser *pasynUser, epicsInt32 *low,  
                               epicsInt32 *high)  
{  
    int function = pasynUser->reason;  
  
    // Analog outputs are 16-bit devices  
    if (function == analogOutValue_) {  
        *low = 0;  
        *high = 65535;  
        return(asynSuccess);  
    } else {  
        return(asynError);  
    }  
}
```

```

asynStatus USB1608G::writeInt32(asynUser *pasynUser, epicsInt32 value)
{
    int addr;
    int function = pasynUser->reason;
    int status=0;
    static const char *functionName = "writeInt32";

    this->getAddress(pasynUser, &addr);
    setIntegerParam(addr, function, value);

    // Analog output functions
    if (function == analogOutValue_) {
        status = cbAOut(boardNum_, addr, BIP10VOLTS, value);
    }

    callParamCallbacks(addr);
    if (status == 0) {
        asynPrint(pasynUser, ASYN_TRACEIO_DRIVER,
            "%s:%s, port %s, wrote %d to address %d\n",
            driverName, functionName, this->portName, value, addr);
    } else {
        asynPrint(pasynUser, ASYN_TRACE_ERROR,
            "%s:%s, port %s, ERROR writing %d to address %d, status=%d\n",
            driverName, functionName, this->portName, value, addr, status);
    }
    return (status==0) ? asynSuccess : asynError;
}

```

```

/** Configuration command, called directly or from iocsh */
extern "C" int USB1608GConfig(const char *portName, int boardNum)
{
    USB1608G *pUSB1608G = new USB1608G(portName, boardNum);
    pUSB1608G = NULL; /* This is just to avoid compiler warnings */
    return(asynSuccess);
}

static const iocshArg configArg0 = { "Port name",      iocshArgString};
static const iocshArg configArg1 = { "Board number",   iocshArgInt};
static const iocshArg * const configArgs[] = {&configArg0,
                                               &configArg1};

static const iocshFuncDef configFuncDef = {"USB1608GConfig", 2, configArgs};
static void configCallFunc(const iocshArgBuf *args)
{
    USB1608GConfig(args[0].sval, args[1].ival);
}

void drvUSB1608GRegister(void)
{
    iocshRegister(&configFuncDef, configCallFunc);
}

extern "C" {
epicsExportRegistrar(drvUSB1608GRegister);
}

```

## measCompAnalogOut.template

```
record(ao,"$(P)$$(R)") {
    field(PINI, "YES")
    field(DTYP, "asynInt32")
    field(FLNK, "$(P)$$(R)Return.PROC PP MS")
    field(OUT, "@asyn($(PORT),$(ADDR))ANALOG_OUT_VALUE")
    field(EGUL, "$(EGUL)")
    field(DRVL, "$(DRVL)")
    field(LOPR, "$(LOPR)")
    field(EGUF, "$(EGUF)")
    field(DRVH, "$(DRVH)")
    field(HOPR, "$(HOPR)")
    field(PREC, "$(PREC)")
    field(LINR, "LINEAR")
    field(VAL, "$(VAL)")
}
```



```
record(ao,"$(P)$(R)Return") {
    field(DTYP, "asynInt32")
    field(DISV, "0")
    field(SDIS, "$(P)$(R)Pulse.VAL  NPP NMS")
    field(OUT, "@asyn($(PORT),$(ADDR))ANALOG_OUT_VALUE")
    field(OMSL, "closed_loop")
    field(EGUL, "$(EGUL)")
    field(DRVL, "$(DRVL)")
    field(LOPR, "$(LOPR)")
    field(EGUF, "$(EGUF)")
    field(DRVH, "$(DRVH)")
    field(HOPR, "$(HOPR)")
    field(PREC, "$(PREC)")
    field(LINR, "LINEAR")
    field(VAL, "$(VAL)")
}
```

```
record(bo,"$(P)$(R)Pulse") {
    field(ZNAM, "Normal")
    field(ONAM, "Pulse")
}
```

```

record(ao,"$(P)$ (R)TweakVal") {
    field(PREC, "$(PREC)")
}

record(calcout,"$(P)$ (R)TweakUp") {
    field(CALC, "A+B")
    field(INPA, "$(P)$ (R).VAL  NPP MS")
    field(INPB, "$(P)$ (R)TweakVal.VAL  NPP MS")
    field(OUT,  "$(P)$ (R).VAL  PP MS")
    field(PREC, "$(PREC)")
}

record(calcout,"$(P)$ (R)TweakDown") {
    field(CALC, "A-B")
    field(INPA, "$(P)$ (R).VAL  NPP MS")
    field(INPB, "$(P)$ (R)TweakVal.VAL  NPP MS")
    field(OUT,  "$(P)$ (R).VAL  PP MS")
    field(PREC, "$(PREC)")
}

```

## st.cmd\_V1

```
< envPaths
```

```
## Register all support components
```

```
dbLoadDatabase "../../dbd/measCompApp.dbd"
```

```
measCompApp_registerRecordDeviceDriver pdbname
```

```
dbLoadTemplate("1608G.substitutions_V1")
```

```
## Configure port driver
```

```
# USB1608GConfig(
```

```
#   portName, # The name to give to this asyn port driver
```

```
#   boardNum) # The number of this board assigned by the
```

```
                #   Measurement Computing Instacal program
```

```
USB1608GConfig("1608G_1", 1)
```

```
#asynSetTraceMask 1608G_1 -1 255
```

```
iocInit
```

## 1608G.substitutions\_V1

```
# asyn record
```

```
file "$(ASYN)/db/asynRecord.db"
```

```
{
```

```
pattern
```

```
{ P, R, PORT, ADDR, IMAX, OMAX }
```

```
{ 1608G:, asyn1, 1608G_1, 0, 80, 80 }
```

```
}
```

```
# Analog outputs
```

```
file "$(MEASCOMP)/measCompApp/Db/measCompAnalogOut.template"
```

```
{
```

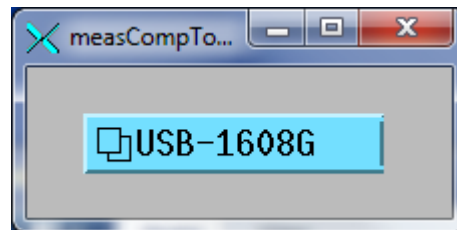
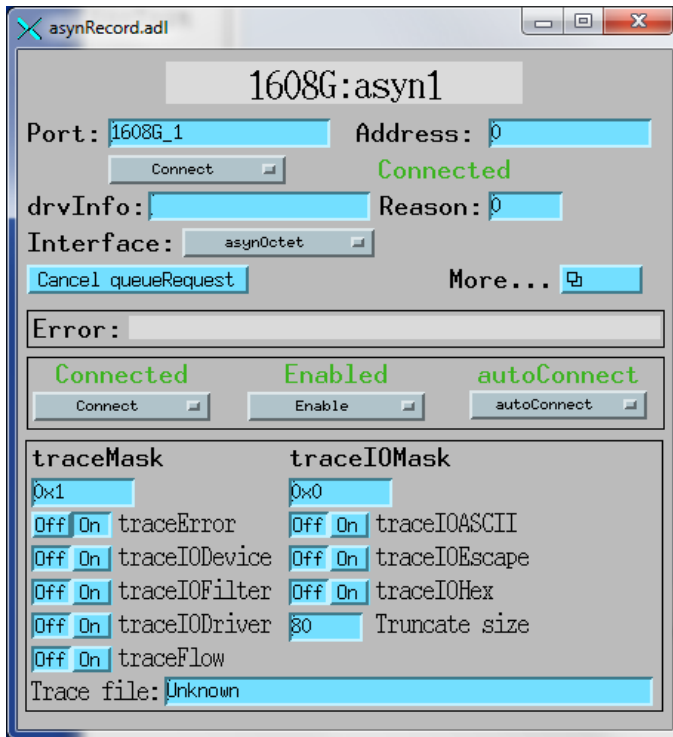
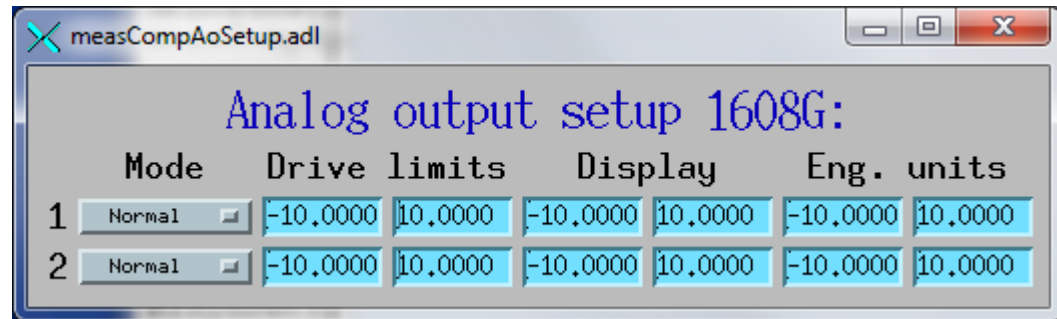
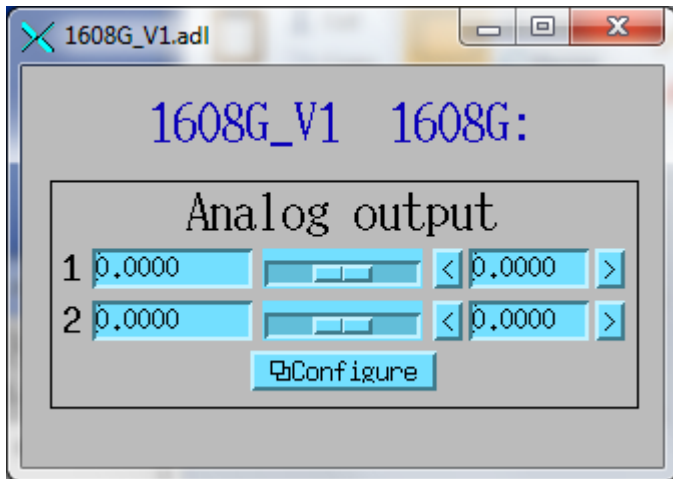
```
pattern
```

```
{ P, R, VAL, PORT, ADDR, EGUL, DRVL, LOPR, EGUF, DRVH, HOPR, PREC }
```

```
{ 1608G:, Ao1, 0, 1608G_1, 0, -10., -10., -10., 10., 10., 10., 4 }
```

```
{ 1608G:, Ao2, 0, 1608G_1, 1, -10., -10., -10., 10., 10., 10., 4 }
```

```
}
```



# Measurement Computing 1608GX-2A0 Driver

## Version 2

- Add 8 simple analog inputs
  - 2 new parameters, `ANALOG_IN_VALUE`, `ANALOG_IN_RANGE`
- Add `report()` function
- 197 lines of code (~60 more than V1)

```

/* drvUSB1608G_V2.cpp
*
* Driver for Measurement Computing USB-1608G multi-function DAQ board using
asynPortDriver base class
*
* This version implements simple analog inputs and simple analog outputs
*
* Mark Rivers
* April 14, 2012
*/

#include <iocsh.h>
#include <epicsExport.h>
#include <asynPortDriver.h>

#include "cbw.h"

static const char *driverName = "USB1608G";

// Analog output parameters
#define analogOutValueString      "ANALOG_OUT_VALUE"

// Analog input parameters
#define analogInValueString       "ANALOG_IN_VALUE"
#define analogInRangeString      "ANALOG_IN_RANGE"

#define NUM_ANALOG_IN      16 // Number analog inputs on 1608G
#define NUM_ANALOG_OUT     2  // Number of analog outputs on 1608G
#define MAX_SIGNALS        NUM_ANALOG_IN

```

```

/** Class definition for the USB1608G class
 */
class USB1608G : public asynPortDriver {
public:
    USB1608G(const char *portName, int boardNum);

    /* These are the methods that we override from asynPortDriver */
    virtual asynStatus writeInt32(asynUser *pasynUser, epicsInt32 value);
    virtual asynStatus readInt32(asynUser *pasynUser, epicsInt32 *value);
    virtual asynStatus getBounds(asynUser *pasynUser, epicsInt32 *low,
                                epicsInt32 *high);
    virtual void report(FILE *fp, int details);

protected:
    // Analog output parameters
    int analogOutValue_;
    #define FIRST_USB1608G_PARAM    analogOutValue_

    // Analog input parameters
    int analogInValue_;
    int analogInRange_;
    #define LAST_USB1608G_PARAM    analogInRange_

private:
    int boardNum_;
};

#define NUM_PARAMS (&LAST_USB1608G_PARAM - &FIRST_USB1608G_PARAM + 1)

```



```

/** Constructor for the USB1608G class
 */
USB1608G::USB1608G(const char *portName, int boardNum)
 : asynPortDriver(portName, MAX_SIGNALS, NUM_PARAMS,
   asynInt32Mask | asynDrvUserMask, // Interfaces that we implement
   0, // Interfaces that do callbacks
   ASYN_MULTIDEVICE | ASYN_CANBLOCK, 1,
   /* ASYN_CANBLOCK=1, ASYN_MULTIDEVICE=1, autoConnect=1 */
   0, 0), /* Default priority and stack size */
   boardNum_(boardNum)
{
 // Analog output parameters
 createParam(analogOutValueString, asynParamInt32, &analogOutValue_);

 // Analog input parameters
 createParam(analogInValueString, asynParamInt32, &analogInValue_);
 createParam(analogInRangeString, asynParamInt32, &analogInRange_);
}

```

```

asynStatus USB1608G::getBounds(asynUser *pasynUser, epicsInt32 *low,
                               epicsInt32 *high)
{
    int function = pasynUser->reason;

    // Both the analog outputs and analog inputs are 16-bit devices
    if ((function == analogOutValue_) ||
        (function == analogInValue_)) {
        *low = 0;
        *high = 65535;
        return(asynSuccess);
    } else {
        return(asynError);
    }
}

```

**NOTE: No change, we don't handle analogOutOfRange\_ here,  
just put in parameter library**

```
asynStatus USB1608G::writeInt32(asynUser *pasynUser, epicsInt32 value)
{
    int addr;
    int function = pasynUser->reason;
    int status=0;
    static const char *functionName = "writeInt32";

    this->getAddress(pasynUser, &addr);
    setIntegerParam(addr, function, value);

    // Analog output functions
    if (function == analogOutValue_) {
        status = cbAOut(boardNum_, addr, BIP10VOLTS, value);
    }

    callParamCallbacks(addr);
    if (status == 0) {
        asynPrint(pasynUser, ASYN_TRACEIO_DRIVER,
            "%s:%s, port %s, wrote %d to address %d\n",
            driverName, functionName, this->portName, value, addr);
    } else {
        asynPrint(pasynUser, ASYN_TRACE_ERROR,
            "%s:%s, port %s, ERROR writing %d to address %d, status=%d\n",
            driverName, functionName, this->portName, value, addr, status);
    }
    return (status==0) ? asynSuccess : asynError;
}
```

```

asynStatus USB1608G::readInt32(asynUser *pasynUser, epicsInt32 *value)
{
    int addr;
    int function = pasynUser->reason;
    int status=0;
    unsigned short shortVal;
    int range;
    //static const char *functionName = "readInt32";

    this->getAddress(pasynUser, &addr);

    // Analog input function
    if (function == analogInValue_) {
        getIntegerParam(addr, analogInRange_, &range);
        status = cbAIn(boardNum_, addr, range, &shortVal);
        *value = shortVal;
        setIntegerParam(addr, analogInValue_, *value);
    }

    // Other functions we call the base class method
    else {
        status = asynPortDriver::readInt32(pasynUser, value);
    }

    callParamCallbacks(addr);
    return (status==0) ? asynSuccess : asynError;
}

```

```

/* Report parameters */
void USB1608G::report(FILE *fp, int details)
{
    int i;
    int range;

    fprintf(fp, "  Port: %s, board number=%d\n",
            this->portName, boardNum_);
    if (details >= 1) {
        for (i=0; i<NUM_ANALOG_IN; i++) {
            getIntegerParam(i, analogInRange_, &range);
            fprintf(fp, "channel %d range=%d", i, range);
        }
        fprintf(fp, "\n");
    }
    asynPortDriver::report(fp, details);
}

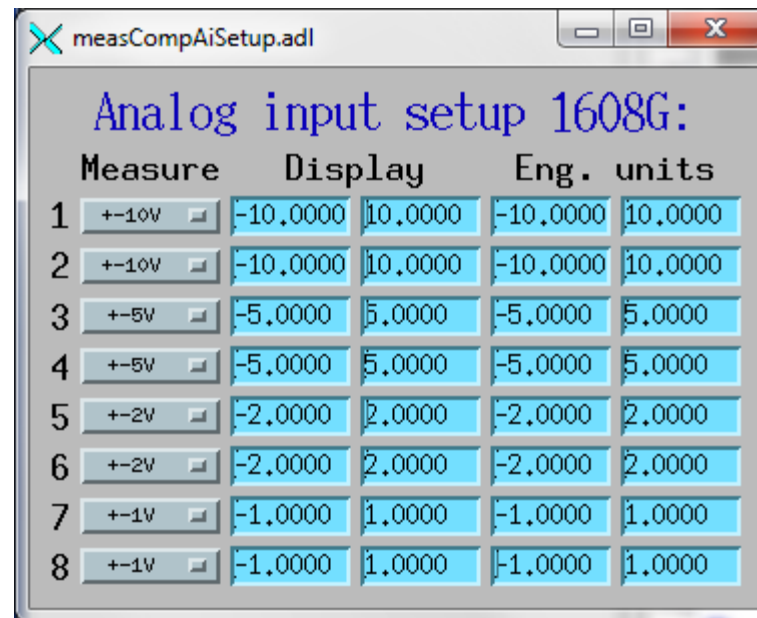
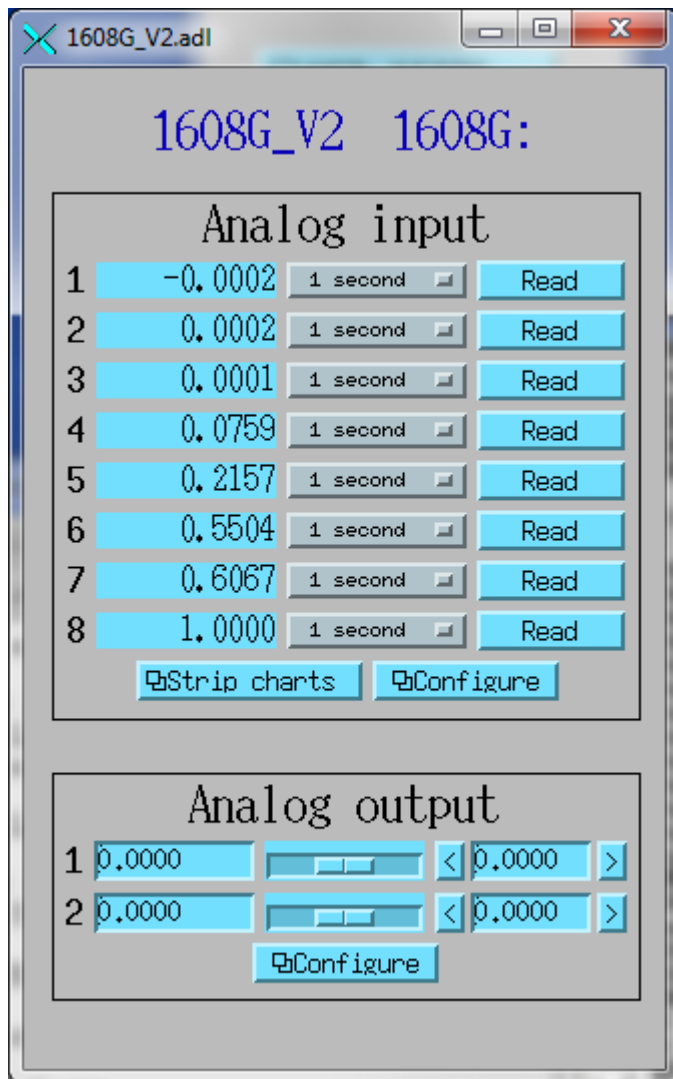
```

## measCompAnalogIn.template

```
record(ai,"$(P)$ (R)")
{
    field(SCAN, "$(SCAN)")
    field(DTYP, "asynInt32")
    field(INP, "@asyn($(PORT),$(ADDR))ANALOG_IN_VALUE")
    field(LINR, "LINEAR")
    field(EGUF, "$(EGUF)")
    field(EGUL, "$(EGUL)")
    field(HOPR, "$(HOPR)")
    field(LOPR, "$(LOPR)")
    field(PREC, "$(PREC)")
}
# Note: the ZRVL, etc. fields correspond to the values of BIP1VOLTS, etc. in cbw.h
record(mbbo,"$(P)$ (R)Range")
{
    field(PINI, "YES")
    field(DTYP, "asynInt32")
    field(OUT, "@asyn($(PORT),$(ADDR))ANALOG_IN_RANGE")
    field(ZRST, "+-1V")
    field(ZRVL, "4")
    field(ONST, "+-2V")
    field(ONVL, "14")
    field(TWST, "+-5V")
    field(TWVL, "0")
    field(THST, "+-10V")
    field(THVL, "1")
    field(VAL, "$(RANGE)")
}
```

## 1608G.substitutions\_V2

```
# Analog inputs
file "$(MEASCOMP)/measCompApp/Db/measCompAnalogIn.template"
{
pattern
{
P,      R,      PORT,      ADDR,      EGUL,      LOPR,      EGUF,      HOPR,      RANGE,      SCAN,      PREC}
{
1608G:, Ai1, 1608G_1, 0, -10., -10., 10., 10., 3, "1 second", 4}
{
1608G:, Ai2, 1608G_1, 1, -10., -10., 10., 10., 3, "1 second", 4}
{
1608G:, Ai3, 1608G_1, 2, -5., -5., 5., 5., 2, "1 second", 4}
{
1608G:, Ai4, 1608G_1, 3, -5., -5., 5., 5., 2, "1 second", 4}
{
1608G:, Ai5, 1608G_1, 4, -2., -2., 2., 2., 1, "1 second", 4}
{
1608G:, Ai6, 1608G_1, 5, -2., -2., 2., 2., 1, "1 second", 4}
{
1608G:, Ai7, 1608G_1, 6, -1., -1., 1., 1., 0, "1 second", 4}
{
1608G:, Ai8, 1608G_1, 7, -1., -1., 1., 1., 0, "1 second", 4}
}
```





# Measurement Computing 1608GX-2A0 Driver

## Version 3

- Add digital outputs
  - 2 new parameters, `DIGITAL_DIRECTION`, `DIGITAL_OUTPUT`
- 253 lines of code (~60 more than Version 2)

```

/* drvUSB1608G_V3.cpp
 *
 * This version implements digital outputs, simple analog inputs and simple analog
outputs
 *
 * Mark Rivers
 * April 14, 2012
 */
#include <iocsh.h>
#include <epicsExport.h>
#include <asynPortDriver.h>
#include "cbw.h"

static const char *driverName = "USB1608G";

// Analog output parameters
#define analogOutValueString      "ANALOG_OUT_VALUE"

// Analog input parameters
#define analogInValueString       "ANALOG_IN_VALUE"
#define analogInRangeString      "ANALOG_IN_RANGE"

// Digital I/O parameters
#define digitalDirectionString    "DIGITAL_DIRECTION"
#define digitalOutputString      "DIGITAL_OUTPUT"

#define NUM_ANALOG_IN      16 // Number analog inputs on 1608G
#define NUM_ANALOG_OUT    2  // Number of analog outputs on 1608G
#define NUM_IO_BITS       8  // Number of digital I/O bits on 1608G
#define MAX_SIGNALS      NUM_ANALOG_IN

```

```

/** Class definition for the USB1608G class
 */
class USB1608G : public asynPortDriver {
public:
    USB1608G(const char *portName, int boardNum);

    /* These are the methods that we override from asynPortDriver */
    virtual asynStatus writeInt32(asynUser *pasynUser, epicsInt32 value);
    virtual asynStatus readInt32(asynUser *pasynUser, epicsInt32 *value);
    virtual asynStatus getBounds(asynUser *pasynUser, epicsInt32 *low,
                                epicsInt32 *high);
    virtual asynStatus writeUInt32Digital(asynUser *pasynUser, epicsUInt32 value,
                                         epicsUInt32 mask);
    virtual void report(FILE *fp, int details);

protected:
    // Analog output parameters
    int analogOutValue_;
    #define FIRST_USB1608G_PARAM analogOutValue_
    // Analog input parameters
    int analogInValue_;
    int analogInRange_;
    // Digital I/O parameters
    int digitalDirection_;
    int digitalOutput_;
    #define LAST_USB1608G_PARAM digitalOutput_

private:
    int boardNum_;
};

```

```

/** Constructor for the USB1608G class
 */
USB1608G::USB1608G(const char *portName, int boardNum)
    : asynPortDriver(portName, MAX_SIGNALS, NUM_PARAMS,
        // Interfaces that we implement
        asynInt32Mask | asynUInt32DigitalMask | asynDrvUserMask,
        // Interfaces that do callbacks
        asynUInt32DigitalMask,
        ASYN_MULTIDEVICE | ASYN_CANBLOCK, 1,
        /* ASYN_CANBLOCK=1, ASYN_MULTIDEVICE=1, autoConnect=1 */
        0, 0), /* Default priority and stack size */
    boardNum_(boardNum)
{
    // Analog output parameters
    createParam(analogOutValueString, asynParamInt32, &analogOutValue_);

    // Analog input parameters
    createParam(analogInValueString, asynParamInt32, &analogInValue_);
    createParam(analogInRangeString, asynParamInt32, &analogInRange_);

    // Digital I/O parameters
    createParam(digitalDirectionString, asynParamUInt32Digital, &digitalDirection_);
    createParam(digitalOutputString, asynParamUInt32Digital, &digitalOutput_);
}

```

```

asynStatus USB1608G::writeUInt32Digital(asynUser *pasynUser, epicsUInt32 value,
epicsUInt32 mask)
{
    int function = pasynUser->reason;
    int status=0;
    int i;
    epicsUInt32 outValue=0, outMask, direction=0;
    static const char *functionName = "writeUInt32Digital";

    setUIntDigitalParam(function, value, mask);
    if (function == digitalDirection_) {
        outValue = (value == 0) ? DIGITALIN : DIGITALOUT;
        for (i=0; i<NUM_IO_BITS; i++) {
            if ((mask & (1<<i)) != 0) {
                status = cbDConfigBit(boardNum_, AUXPORT, i, outValue);
            }
        }
    }
}

else if (function == digitalOutput_) {
    getUIntDigitalParam(digitalDirection_, &direction, 0xFFFFFFFF);
    for (i=0, outMask=1; i<NUM_IO_BITS; i++, outMask = (outMask<<1)) {
        // Only write the value if the mask has this bit set and the direction
        // for that bit is output (1)
        outValue = ((value & outMask) == 0) ? 0 : 1;
        if ((mask & outMask & direction) != 0) {
            status = cbDBitOut(boardNum_, AUXPORT, i, outValue);
        }
    }
}
}

```

```

}

callParamCallbacks();
if (status == 0) {
    asynPrint(pasynUser, ASYN_TRACEIO_DRIVER,
        "%s:%s, port %s, wrote outValue=0x%x, value=0x%x, mask=0x%x, direction=0x%x\n",
        driverName, functionName, this->portName, outValue, value, mask, direction);
} else {
    asynPrint(pasynUser, ASYN_TRACE_ERROR,
        "%s:%s, port %s, ERROR writing outValue=0x%x, value=0x%x, mask=0x%x,
        direction=0x%x, status=%d\n",
        driverName, functionName, this->portName, outValue, value, mask, direction,
        status);
}
return (status==0) ? asynSuccess : asynError;
}

```

### measCompBinaryDir.template

```
record(bo, "$(P)$ (R)")
{
    field(PINI, "YES")
    field(DTYP, "asynUInt32Digital")
    field(OUT, "@asynMask($(PORT),$(ADDR),$(MASK))DIGITAL_DIRECTION")
    field(ZNAM, "In")
    field(ONAM, "Out")
    field(VAL, "$(VAL)")
    field(PINI, "YES")
}
```

### measCompBinaryOut.template

```
record(bo, "$(P)$ (R)")
{
    field(PINI, "YES")
    field(DTYP, "asynUInt32Digital")
    field(OUT, "@asynMask($(PORT),$(ADDR),$(MASK))DIGITAL_OUTPUT")
    field(ZNAM, "Low")
    field(ONAM, "High")
}
```

**# This is a readback of the output, with SCAN=I/O Intr**

```
record(bi, "$(P)$ (R)_RBV")
{
    field(DTYP, "asynUInt32Digital")
    field(INP, "@asynMask($(PORT),$(ADDR),$(MASK))DIGITAL_OUTPUT")
    field(ZNAM, "Low")
    field(ONAM, "High")
    field(SCAN, "I/O Intr")
}
```

```
}
```

### measCompLongOut.template

```
record(longout, "$(P)$(R)")
{
    field(PINI, "YES")
    field(DTYP, "asynUInt32Digital")
    field(OUT, "@asynMask($(PORT),$(ADDR),$(MASK))DIGITAL_OUTPUT")
}

# This is a readback of the output, with SCAN=I/O Intr
record(longin, "$(P)$(R)_RBV")
{
    field(DTYP, "asynUInt32Digital")
    field(INP, "@asynMask($(PORT),$(ADDR),$(MASK))DIGITAL_OUTPUT")
    field(SCAN, "I/O Intr")
}
```



### 1608G.substitutions\_V3

```
file "$(MEASCOMP)/measCompApp/Db/measCompLongOut.template"
{
pattern
{
    P,      R,      MASK,      PORT,      ADDR}
{
    1608G:,  Lo,     0xFF,    1608G_1,    0}
}
```

```
file "$(MEASCOMP)/measCompApp/Db/measCompBinaryOut.template"
{
pattern
{
    P,      R,      MASK,      PORT,      ADDR}
{
    1608G:,  Bo1,    0x01,    1608G_1,    0}
{
    1608G:,  Bo2,    0x02,    1608G_1,    0}
{
    1608G:,  Bo3,    0x04,    1608G_1,    0}
{
    1608G:,  Bo4,    0x08,    1608G_1,    0}
{
    1608G:,  Bo5,    0x10,    1608G_1,    0}
{
    1608G:,  Bo6,    0x20,    1608G_1,    0}
{
    1608G:,  Bo7,    0x40,    1608G_1,    0}
{
    1608G:,  Bo8,    0x80,    1608G_1,    0}
}
```

```

# Direction bits on binary I/O
# VAL 0=input, 1=output
file "$(MEASCOMP)/measCompApp/Db/measCompBinaryDir.template"
{
pattern
{
    P,      R,      MASK,  VAL,  PORT,  ADDR}
{ 1608G:, Bd1, 0x01, 0, 1608G_1, 0}
{ 1608G:, Bd2, 0x02, 0, 1608G_1, 0}
{ 1608G:, Bd3, 0x04, 0, 1608G_1, 0}
{ 1608G:, Bd4, 0x08, 0, 1608G_1, 0}
{ 1608G:, Bd5, 0x10, 1, 1608G_1, 0}
{ 1608G:, Bd6, 0x20, 1, 1608G_1, 0}
{ 1608G:, Bd7, 0x40, 1, 1608G_1, 0}
{ 1608G:, Bd8, 0x80, 1, 1608G_1, 0}
}
}

```

1608G\_V3.adl

## 1608G\_V3 1608G:

### Analog output

1 2.0000  < 0.0000 >

2 0.0000  < 0.0000 >

### Analog input

1	1.9994	1 second	Read
2	-0.0002	1 second	Read
3	0.0001	1 second	Read
4	0.1032	1 second	Read
5	0.2089	1 second	Read
6	0.5475	1 second	Read
7	0.6105	1 second	Read
8	1.0000	1 second	Read

### Digital I/O

	0	1	2	3	4	5	6	7	
Outputs	Low	Low	Low	Low	Low	Low	Low	Low	0x0
	High	High	High	High	High	High	High	High	
Direction	In	In	In	In	In	In	In	In	
	Out	Out	Out	Out	Out	Out	Out	Out	

# Measurement Computing 1608GX-2A0 Driver

## Version 4

- Add digital inputs
  - 1 new parameter, `DIGITAL_INPUT`
- Add a poller thread to read digital inputs; their records are `SCAN=I/O Intr`
- 314 lines of code (~60 more than Version 3)

```

/* drvUSB1608G_V2.cpp
* This version implements digital inputs and outputs, simple analog inputs and simple
analog outputs, with a poller thread
*
* Mark Rivers
* April 14, 2012
*/

#include <iocsh.h>
#include <epicsExport.h>
#include <epicsThread.h>
#include <asynPortDriver.h>
#include "cbw.h"

static const char *driverName = "USB1608G";

// Analog output parameters
#define analogOutValueString      "ANALOG_OUT_VALUE"
// Analog input parameters
#define analogInValueString      "ANALOG_IN_VALUE"
#define analogInRangeString     "ANALOG_IN_RANGE"
// Digital I/O parameters
#define digitalDirectionString   "DIGITAL_DIRECTION"
#define digitalInputString     "DIGITAL_INPUT"
#define digitalOutputString     "DIGITAL_OUTPUT"

#define NUM_ANALOG_IN    16 // Number analog inputs on 1608G
#define NUM_ANALOG_OUT  2  // Number of analog outputs on 1608G
#define NUM_IO_BITS     8  // Number of digital I/O bits on 1608G
#define MAX_SIGNALS     NUM_ANALOG_IN

```

```

#define DEFAULT_POLL_TIME 0.01
/** Class definition for the USB1608G class
 */
class USB1608G : public asynPortDriver {
public:
    USB1608G(const char *portName, int boardNum);

    /** These are the methods that we override from asynPortDriver */
    virtual asynStatus writeInt32(asynUser *pasynUser, epicsInt32 value);
    virtual asynStatus readInt32(asynUser *pasynUser, epicsInt32 *value);
    virtual asynStatus getBounds(asynUser *pasynUser, epicsInt32 *low,
                                epicsInt32 *high);
    virtual asynStatus writeUInt32Digital(asynUser *pasynUser, epicsUInt32 value,
                                         epicsUInt32 mask);
    virtual void report(FILE *fp, int details);
    /** These should be private but are called from C
    virtual void pollerThread(void);

protected:
    /** Analog output parameters
    int analogOutValue_;
    #define FIRST_USB1608G_PARAM    analogOutValue_

    /** Analog input parameters
    int analogInValue_;
    int analogInRange_;

    /** Digital I/O parameters
    int digitalDirection_;
    int digitalInput_;

```

```
int digitalOutput_;
#define LAST_USB1608G_PARAM digitalOutput_

private:
    int boardNum_;
    double pollTime_;
    int forceCallback_;
};

#define NUM_PARAMS (&LAST_USB1608G_PARAM - &FIRST_USB1608G_PARAM + 1)

static void pollerThreadC(void * pPvt)
{
    USB1608G *pUSB1608G = (USB1608G *)pPvt;
    pUSB1608G->pollerThread();
}
```

```

/** Constructor for the USB1608G class */
USB1608G::USB1608G(const char *portName, int boardNum)
    : asynPortDriver(portName, MAX_SIGNALS, NUM_PARAMS,
        // Interfaces that we implement
        asynInt32Mask | asynUInt32DigitalMask | asynDrvUserMask,
        // Interfaces that do callbacks
        asynUInt32DigitalMask,
        ASYN_MULTIDEVICE | ASYN_CANBLOCK, 1,
        /* ASYN_CANBLOCK=1, ASYN_MULTIDEVICE=1, autoConnect=1 */
        0, 0), /* Default priority and stack size */
        boardNum_(boardNum),
        pollTime_(DEFAULT_POLL_TIME),
        forceCallback_(1)
{
    // Analog output parameters
    createParam(analogOutValueString, asynParamInt32, &analogOutValue_);
    // Analog input parameters
    createParam(analogInValueString, asynParamInt32, &analogInValue_);
    createParam(analogInRangeString, asynParamInt32, &analogInRange_);
    // Digital I/O parameters
    createParam(digitalDirectionString, asynParamUInt32Digital, &digitalDirection_);
    createParam(digitalInputString, asynParamUInt32Digital, &digitalInput_);
    createParam(digitalOutputString, asynParamUInt32Digital, &digitalOutput_);
    /* Start the thread to poll digital inputs and do callbacks to device support */
    epicsThreadCreate("USB1608GPoller",
        epicsThreadPriorityLow,
        epicsThreadGetStackSize(epicsThreadStackMedium),
        (EPICSTHREADFUNC)pollerThreadC,
        this);
}

```



```

void USB1608G::pollerThread()
{
    /* This function runs in a separate thread. It waits for the poll time */
    static const char *functionName = "pollerThread";
    epicsUInt32 newValue, changedBits, prevInput=0;
    unsigned short biVal;;
    int i, status;

    while(1) {
        lock();
        // Read the digital inputs
        status = cbDIn(boardNum_, AUXPORT, &biVal);
        if (status)
            asynPrint(pasynUserSelf, ASYN_TRACE_ERROR,
                "%s:%s: ERROR calling cbDIn, status=%d\n",
                driverName, functionName, status);
        newValue = biVal;
        changedBits = newValue ^ prevInput;
        if (forceCallback_ || (changedBits != 0)) {
            prevInput = newValue;
            forceCallback_ = 0;
            setUIntDigitalParam(digitalInput_, newValue, 0xFFFFFFFF);
        }
        for (i=0; i<MAX_SIGNALS; i++) {
            callParamCallbacks(i);
        }
        unlock();
        epicsThreadSleep(pollTime_);
    }
}

```

### measCompBinaryIn.template

```
record(bi, "$(P)$ (R)")
{
    field(DTYP, "asynUInt32Digital")
    field(INP, "@asynMask($(PORT),$(ADDR),$(MASK))DIGITAL_INPUT")
    field(ZNAM, "Low")
    field(ONAM, "High")
    field(SCAN, "I/O Intr")
}
```

### measCompLongIn.template

```
record(longin, "$(P)$ (R)")
{
    field(DTYP, "asynUInt32Digital")
    field(INP, "@asynMask($(PORT),$(ADDR),$(MASK))DIGITAL_INPUT")
    field(SCAN, "I/O Intr")
}
```

## 1608G.substitutions\_V4

```
file "$(MEASCOMP)/measCompApp/Db/measCompLongIn.template"
{
pattern
{
    P,      R,      MASK,      PORT,      ADDR}
{
    1608G:,  Li,     0xFF,    1608G_1,    0}
}
```

```
file "$(MEASCOMP)/measCompApp/Db/measCompBinaryIn.template"
{
pattern
{
    P,      R,      MASK,      PORT,      ADDR}
{
    1608G:,  Bi1,    0x01,    1608G_1,    0}
{
    1608G:,  Bi2,    0x02,    1608G_1,    0}
{
    1608G:,  Bi3,    0x04,    1608G_1,    0}
{
    1608G:,  Bi4,    0x08,    1608G_1,    0}
{
    1608G:,  Bi5,    0x10,    1608G_1,    0}
{
    1608G:,  Bi6,    0x20,    1608G_1,    0}
{
    1608G:,  Bi7,    0x40,    1608G_1,    0}
{
    1608G:,  Bi8,    0x80,    1608G_1,    0}
}
```

1608G\_V4.adl

## 1608G\_V4 1608G:

### Analog output

1 0.0000  < 0.0000 >

2 0.0000  < 0.0000 >

### Analog input

1	-0.0002	1 second	Read
2	-0.0002	1 second	Read
3	-0.0001	1 second	Read
4	0.0832	1 second	Read
5	0.2199	1 second	Read
6	0.5943	1 second	Read
7	0.6385	1 second	Read
8	1.0000	1 second	Read

### Digital I/O

	0	1	2	3	4	5	6	7	
Inputs	<span style="color: red;">●</span>	<span style="color: green;">●</span>	<span style="color: green;">●</span>	<span style="color: green;">●</span>	<span style="color: red;">●</span>	<span style="color: green;">●</span>	<span style="color: green;">●</span>	<span style="color: green;">●</span>	0x11
Outputs	Low High	Low High	Low High	Low High	Low High	Low High	Low High	Low High	0x0
Direction	In Out	In Out	In Out	In Out	In Out	In Out	In Out	In Out	

# Measurement Computing 1608GX-2A0 Driver

## Version 5

- Add pulse generator output and counter inputs
  - 8 new parameters, PULSE\_RUN, PULSE\_PERIOD, PULSE\_WIDTH, PULSE\_DELAY, PULSE\_COUNT, PULSE\_IDLE\_STATE, COUNTER\_VALUE, COUNTER\_RESET
- Counter inputs are polled in poller thread, SCAN=I/O Intr
- 484 lines of code (~170 more than Version 4)

```

/* drvUSB1608G_V5.cpp
 *
 * Driver for Measurement Computing USB-1608G multi-function DAQ board using
 asynPortDriver base class
 *
 * This version implements digital inputs and outputs, simple analog inputs and
 simple analog outputs, will a poller thread
 *
 * Mark Rivers
 * April 14, 2012
 */

#include <iocsh.h>
#include <epicsExport.h>
#include <epicsThread.h>
#include <asynPortDriver.h>

#include "cbw.h"

static const char *driverName = "USB1608G";

// Analog output parameters
#define analogOutValueString      "ANALOG_OUT_VALUE"

// Analog input parameters
#define analogInValueString       "ANALOG_IN_VALUE"
#define analogInRangeString      "ANALOG_IN_RANGE"

// Digital I/O parameters
#define digitalDirectionString    "DIGITAL_DIRECTION"

```

```

#define digitalInputString      "DIGITAL_INPUT"
#define digitalOutputString    "DIGITAL_OUTPUT"

// Pulse output parameters
#define pulseGenRunString      "PULSE_RUN"
#define pulseGenPeriodString  "PULSE_PERIOD"
#define pulseGenWidthString   "PULSE_WIDTH"
#define pulseGenDelayString   "PULSE_DELAY"
#define pulseGenCountString   "PULSE_COUNT"
#define pulseGenIdleStateString "PULSE_IDLE_STATE"

// Counter parameters
#define counterCountsString    "COUNTER_VALUE"
#define counterResetString     "COUNTER_RESET"

#define MIN_FREQUENCY          0.0149
#define MAX_FREQUENCY          32e6
#define MIN_DELAY              0.
#define MAX_DELAY              67.11
#define NUM_ANALOG_IN          16 // Number analog inputs on 1608G
#define NUM_ANALOG_OUT         2 // Number of analog outputs on 1608G
#define NUM_COUNTERS           2 // Number of counters on 1608G
#define NUM_TIMERS              1 // Number of timers on 1608G
#define NUM_IO_BITS            8 // Number of digital I/O bits on 1608G
#define MAX_SIGNALS            NUM_ANALOG_IN
#define DEFAULT_POLL_TIME      0.01

```

```

/** Class definition for the USB1608G class
 */
class USB1608G : public asynPortDriver {
public:
    USB1608G(const char *portName, int boardNum);

    /* These are the methods that we override from asynPortDriver */
    virtual asynStatus writeInt32(asynUser *pasynUser, epicsInt32 value);
    virtual asynStatus readInt32(asynUser *pasynUser, epicsInt32 *value);
    virtual asynStatus writeFloat64(asynUser *pasynUser, epicsFloat64 value);
    virtual asynStatus getBounds(asynUser *pasynUser, epicsInt32 *low,
                                  epicsInt32 *high);
    virtual asynStatus writeUInt32Digital(asynUser *pasynUser, epicsUInt32 value,
                                          epicsUInt32 mask);
    virtual void report(FILE *fp, int details);
    /* These should be private but are called from C */
    virtual void pollerThread(void);

protected:
    // Pulse generator parameters
    int pulseGenRun_;
    #define FIRST_USB1608G_PARAM pulseGenRun_
    int pulseGenPeriod_;
    int pulseGenWidth_;
    int pulseGenDelay_;
    int pulseGenCount_;
    int pulseGenIdleState_;

    // Counter parameters
    int counterCounts_;

```



```

int counterReset_;

// Analog output parameters
int analogOutValue_;

// Analog input parameters
int analogInValue_;
int analogInRange_;

// Digital I/O parameters
int digitalDirection_;
int digitalInput_;
int digitalOutput_;
#define LAST_USB1608G_PARAM digitalOutput_

private:
    int boardNum_;
    double pollTime_;
    int forceCallback_;
    int startPulseGenerator();
    int stopPulseGenerator();
    int pulseGenRunning_;
};

#define NUM_PARAMS (&LAST_USB1608G_PARAM - &FIRST_USB1608G_PARAM + 1)

```

```

/** Constructor for the USB1608G class
 */
USB1608G::USB1608G(const char *portName, int boardNum)
    : asynPortDriver(portName, MAX_SIGNALS, NUM_PARAMS,
        // Interfaces that we implement
        asynInt32Mask | asynFloat64Mask | asynUInt32DigitalMask | asynDrvUserMask,
        // Interfaces that do callbacks
        asynInt32Mask | asynFloat64Mask | asynUInt32DigitalMask,
        /* ASYN_CANBLOCK=1, ASYN_MULTIDEVICE=1, autoConnect=1 */
        ASYN_MULTIDEVICE | ASYN_CANBLOCK, 1, 0, 0), /* Default priority and stack size */
        boardNum_(boardNum),
        pollTime_(DEFAULT_POLL_TIME),
        forceCallback_(1)
{
    // Pulse generator parameters
    createParam(pulseGenRunString,                asynParamInt32, &pulseGenRun_);
    createParam(pulseGenPeriodString,            asynParamFloat64, &pulseGenPeriod_);
    createParam(pulseGenWidthString,            asynParamFloat64, &pulseGenWidth_);
    createParam(pulseGenDelayString,            asynParamFloat64, &pulseGenDelay_);
    createParam(pulseGenCountString,            asynParamInt32, &pulseGenCount_);
    createParam(pulseGenIdleStateString,        asynParamInt32, &pulseGenIdleState_);

    // Counter parameters
    createParam(counterCountsString,            asynParamInt32, &counterCounts_);
    createParam(counterResetString,            asynParamInt32, &counterReset_);

    // Analog output parameters
    createParam(analogOutValueString,          asynParamInt32, &analogOutValue_);

    // Analog input parameters

```

```

createParam(analogInValueString,          asynParamInt32, &analogInValue_);
createParam(analogInRangeString,         asynParamInt32, &analogInRange_);

// Digital I/O parameters
createParam(digitalDirectionString,     asynParamUInt32Digital, &digitalDirection_);
createParam(digitalInputString,         asynParamUInt32Digital, &digitalInput_);
createParam(digitalOutputString,        asynParamUInt32Digital, &digitalOutput_);

/* Start the thread to poll digital inputs and do callbacks to
 * device support */
epicsThreadCreate("USB1608GPoller",
                  epicsThreadPriorityLow,
                  epicsThreadGetStackSize(epicsThreadStackMedium),
                  (EPICSTHREADFUNC)pollerThreadC,
                  this);
}

```

```

int USB1608G::startPulseGenerator()
{
    int status=0;
    double frequency, period, width, delay;
    int timerNum=0;
    double dutyCycle;
    int count, idleState;
    static const char *functionName = "startPulseGenerator";

    getDoubleParam (timerNum, pulseGenPeriod_,    &period);
    getDoubleParam (timerNum, pulseGenWidth_,     &width);
    getDoubleParam (timerNum, pulseGenDelay_,     &delay);
    getIntegerParam(timerNum, pulseGenCount_,     &count);
    getIntegerParam(timerNum, pulseGenIdleState_, &idleState);

    frequency = 1./period;
    if (frequency < MIN_FREQUENCY) frequency = MIN_FREQUENCY;
    if (frequency > MAX_FREQUENCY) frequency = MAX_FREQUENCY;
    dutyCycle = width * frequency;
    period = 1. / frequency;
    if (dutyCycle <= 0.) dutyCycle = .0001;
    if (dutyCycle >= 1.) dutyCycle = .9999;
    if (delay < MIN_DELAY) delay = MIN_DELAY;
    if (delay > MAX_DELAY) delay = MAX_DELAY;

    status = cbPulseOutStart(boardNum_, timerNum, &frequency, &dutyCycle, count,
                             &delay, idleState, 0);
    if (status != 0) {
        asynPrint(pasynUserSelf, ASYN_TRACE_ERROR,
            "%s:%s: started pulse generator %d period=%f, width=%f, count=%d,

```

```

        delay=%f, idleState=%d, status=%d\n",
        driverName, functionName, timerNum, period, width, count, delay,
        idleState, status);
    return status;
}
// We may not have gotten the frequency, dutyCycle, and delay we asked for,
// set the actual values in the parameter library
pulseGenRunning_ = 1;
period = 1. / frequency;
width = period * dutyCycle;
asynPrint(pasynUserSelf, ASYN_TRACE_FLOW,
    "%s:%s: started pulse generator %d actual frequency=%f, actual period=%f,
    actual width=%f, actual delay=%f\n",
    driverName, functionName, timerNum, frequency, period, width, delay);
setDoubleParam(timerNum, pulseGenPeriod_, period);
setDoubleParam(timerNum, pulseGenWidth_, width);
setDoubleParam(timerNum, pulseGenDelay_, delay);
return 0;
}

int USB1608G::stopPulseGenerator()
{
    pulseGenRunning_ = 0;
    return cbPulseOutStop(boardNum_, 0);
}

```

```

asynStatus USB1608G::writeInt32(asynUser *pasynUser, epicsInt32 value)
{
    int addr;
    int function = pasynUser->reason;
    int status=0;
    static const char *functionName = "writeInt32";

    this->getAddress(pasynUser, &addr);
    setIntegerParam(addr, function, value);

    // Pulse generator functions
    if (function == pulseGenRun_) {
        // Allow starting a run even if it thinks its running,
        // since there is no way to know when it got done if Count!=0
        if (value) {
            status = startPulseGenerator();
        }
        else if (!value && pulseGenRunning_) {
            status = stopPulseGenerator();
        }
    }
    if ((function == pulseGenCount_) ||
        (function == pulseGenIdleState_)) {
        if (pulseGenRunning_) {
            status = stopPulseGenerator();
            status |= startPulseGenerator();
        }
    }
}

// Counter functions

```

```

if (function == counterReset_) {
    // LOADREG0=0, LOADREG1=1, so we use addr
    status = cbCLoad32(boardNum_, addr, 0);
}

// Analog output functions
if (function == analogOutValue_) {
    status = cbAOut(boardNum_, addr, BIP10VOLTS, value);
}

callParamCallbacks(addr);
if (status == 0) {
    asynPrint(pasynUser, ASYN_TRACEIO_DRIVER,
              "%s:%s, port %s, wrote %d to address %d\n",
              driverName, functionName, this->portName, value, addr);
} else {
    asynPrint(pasynUser, ASYN_TRACE_ERROR,
              "%s:%s, port %s, ERROR writing %d to address %d, status=%d\n",
              driverName, functionName, this->portName, value, addr, status);
}
return (status==0) ? asynSuccess : asynError;
}

```

```

asynStatus USB1608G::writeFloat64(asynUser *pasynUser, epicsFloat64 value)
{
    int addr;
    int function = pasynUser->reason;
    int status=0;
    static const char *functionName = "writeFloat64";

    this->getAddress(pasynUser, &addr);
    setDoubleParam(addr, function, value);
    // Pulse generator functions
    if ((function == pulseGenPeriod_)    ||
        (function == pulseGenWidth_)    ||
        (function == pulseGenDelay_)) {
        if (pulseGenRunning_) {
            status = stopPulseGenerator();
            status |= startPulseGenerator();
        }
    }
    callParamCallbacks(addr);
    if (status == 0) {
        asynPrint(pasynUser, ASYN_TRACEIO_DRIVER,
            "%s:%s, port %s, wrote %d to address %d\n",
            driverName, functionName, this->portName, value, addr);
    } else {
        asynPrint(pasynUser, ASYN_TRACE_ERROR,
            "%s:%s, port %s, ERROR writing %f to address %d, status=%d\n",
            driverName, functionName, this->portName, value, addr, status);
    }
    return (status==0) ? asynSuccess : asynError;
}

```



```

void USB1608G::pollerThread()
{
    /* This function runs in a separate thread.  It waits for the poll time */
    static const char *functionName = "pollerThread";
    epicsUInt32 newValue, changedBits, prevInput=0;
    unsigned short biVal;;
    unsigned long countVal;
    int i;
    int status;

    while(1) {
        lock();

        // Read the counter inputs
        for (i=0; i<NUM_COUNTERS; i++) {
            status = cbCIn32(boardNum_, i, &countVal);
            if (status)
                asynPrint(pasynUserSelf, ASYN_TRACE_ERROR,
                    "%s:%s: ERROR calling cbCIn32, status=%d\n",
                    driverName, functionName, status);
            setIntegerParam(i, counterCounts_, countVal);
        }

        // Read the digital inputs
        status = cbDIn(boardNum_, AUXPORT, &biVal);
        if (status)
            asynPrint(pasynUserSelf, ASYN_TRACE_ERROR,
                "%s:%s: ERROR calling cbDIn, status=%d\n",
                driverName, functionName, status);
        newValue = biVal;
    }
}

```

```
changedBits = newValue ^ prevInput;
if (forceCallback_ || (changedBits != 0)) {
    prevInput = newValue;
    forceCallback_ = 0;
    setUIntDigitalParam(digitalInput_, newValue, 0xFFFFFFFF);
}

for (i=0; i<MAX_SIGNALS; i++) {
    callParamCallbacks(i);
}
unlock();
epicsThreadSleep(pollTime_);
}
}
```

## measCompPulseGen.template

```
#####  
# Pulse start/stop #  
#####  
record(bo, "$(P)$ (R)Run")  
{  
    field(DTYP, "asynInt32")  
    field(OUT, "@asyn($(PORT),$(ADDR))PULSE_RUN")  
    field(ZNAM, "Stop")  
    field(ZSV, "NO_ALARM")  
    field(ONAM, "Run")  
    field(OSV, "MINOR")  
}  
  
# NOTE: The records for the period and the frequency are a bit  
# complex because we want to be able to change either ao record  
# and have the other one update  
  
#####  
# Pulse period #  
#####  
record(ao, "$(P)$ (R)Period")  
{  
    field(PINI, "YES")  
    field(DTYP, "asynFloat64")  
    field(OUT, "@asyn($(PORT),$(ADDR))PULSE_PERIOD")  
    field(VAL, "0.001")  
    field(PREC, "$(PREC)")  
    field(FLNK, "$(P)$ (R)CalcFrequency")  
}
```

```
#####
# Calculate frequency based on new period #
#####
record(calcout, "$(P)$ (R)CalcFrequency")
{
    field(INPA, "$(P)$ (R)Period")
    field(CALC, "1/A")
    field(SDIS, "$(P)$ (R)Frequency.PROC")
    field(DISV, "1")
    field(OUT, "$(P)$ (R)Frequency PP MS")
}

```

```
#####
# Pulse frequency #
#####
record(ao, "$(P)$ (R)Frequency")
{
    field(PREC, "$(PREC)")
    field(FLNK, "$(P)$ (R)CalcPeriod PP MS")
}

```

```
#####
# Calculate period based on new frequency #
#####
record(calcout, "$(P)$ (R)CalcPeriod")
{
    field(INPA, "$(P)$ (R)Frequency")
    field(CALC, "1/A")
    field(SDIS, "$(P)$ (R)Period.PROC")
}

```

```

    field(DISV, "1")
    field(OUT, "$(P)$ (R)Period PP MS")
}

```

```

#####
# Pulse width #
#####
record(ao, "$(P)$ (R)Width")
{
    field(PINI, "YES")
    field(DTYP, "asynFloat64")
    field(OUT, "@asyn($(PORT),$(ADDR))PULSE_WIDTH")
    field(VAL, "0.0001")
    field(PREC, "$(PREC)")
}

```

```

#####
# Pulse delay #
#####
record(ao, "$(P)$ (R)Delay")
{
    field(PINI, "YES")
    field(DTYP, "asynFloat64")
    field(OUT, "@asyn($(PORT),$(ADDR))PULSE_DELAY")
    field(VAL, "0.")
    field(PREC, "$(PREC)")
}

```

```
#####
# Pulse count #
#####
record(longout, "$(P)$(R)Count")
{
    field(PINI, "YES")
    field(DTYP, "asynInt32")
    field(OUT, "@asyn($(PORT),$(ADDR))PULSE_COUNT")
}

```

```
#####
# Pulse idle state #
#####
record(bo, "$(P)$(R)IdleState")
{
    field(PINI, "YES")
    field(DTYP, "asynInt32")
    field(OUT, "@asyn($(PORT),$(ADDR))PULSE_IDLE_STATE")
    field(ZNAM, "Low")
    field(ONAM, "High")
    field(VAL, "0")
}

```

## 1608GCounter.template

```
record(longin, "$(P)$(R)Counts")
{
    field(DTYP, "asynInt32")
    field(INP, "@asyn($(PORT),$(ADDR))COUNTER_VALUE")
    field(SCAN, "I/O Intr")
}

record(bo, "$(P)$(R)Reset")
{
    field(PINI, "YES")
    field(DTYP, "asynInt32")
    field(OUT, "@asyn($(PORT),$(ADDR))COUNTER_RESET")
    field(VAL, "1")
}
```

## 1608G.substitutions\_V5

```
file "$(MEASCOMP)/measCompApp/Db/measCompPulseGen.template"
{
pattern
{      P,          R,          PORT,    ADDR,    PREC}
{  1608G:,  PulseGen1,  1608G_1,    0,      4}
}
```

```
file "$(MEASCOMP)/measCompApp/Db/1608GCounter.template"
{
pattern
{      P,          R,          PORT,    ADDR}
{  1608G:,  Counter1,  1608G_1,    0}
{  1608G:,  Counter2,  1608G_1,    1}
}
```



1608G\_V5.adl

## 1608G\_V5 1608G:

### Analog output

1 0.0000  < 0.0000 >

2 0.0000  < 0.0000 >

### Analog input

1	-0.0002	1 second	Read
2	-0.0014	1 second	Read
3	0.0001	1 second	Read
4	0.0866	1 second	Read
5	0.2354	1 second	Read
6	0.5816	1 second	Read
7	0.6258	1 second	Read
8	1.0000	1 second	Read

### Pulse generator

Frequency

Period

Width

Initial delay

# pulses

Idle state

### Counters

1	13035	Reset
2	0	Reset

### Digital I/O

	0	1	2	3	4	5	6	7	
Inputs	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input type="text" value="0x0"/>
Outputs	<input type="text" value="Low"/>	<input type="text" value="Low"/>	<input type="text" value="Low"/>	<input type="text" value="Low"/>	<input type="text" value="Low"/>	<input type="text" value="Low"/>	<input type="text" value="Low"/>	<input type="text" value="Low"/>	<input type="text" value="0x0"/>
Direction	<input type="text" value="In"/>	<input type="text" value="In"/>	<input type="text" value="In"/>	<input type="text" value="In"/>	<input type="text" value="In"/>	<input type="text" value="In"/>	<input type="text" value="In"/>	<input type="text" value="In"/>	
	<input type="text" value="Out"/>	<input type="text" value="Out"/>	<input type="text" value="Out"/>	<input type="text" value="Out"/>	<input type="text" value="Out"/>	<input type="text" value="Out"/>	<input type="text" value="Out"/>	<input type="text" value="Out"/>	

# Measurement Computing 1608GX-2A0 Driver

## Full Released Version

- Add waveform generator with both predefined and user-defined waveforms
- Add 8-channel waveform digitizer
- Add trigger support
- 42 new parameters
- 1254 lines of code (~770 more than Version 4)
- Supports virtually all features of 1608GX-2A0

1608G\_module.adl

## USB-1608G-2AO 1608G:

### Pulse generator

Frequency

Period

Width

Initial delay

# pulses

Idle state

Start Stop Done

### Counters

1  Reset

2  Reset

### Digital I/O

	0	1	2	3	4	5	6	7	
Inputs	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input type="text" value="0x0"/>
Outputs	<input type="text" value="Low"/>	<input type="text" value="Low"/>	<input type="text" value="Low"/>	<input type="text" value="Low"/>	<input type="text" value="Low"/>	<input type="text" value="Low"/>	<input type="text" value="Low"/>	<input type="text" value="Low"/>	<input type="text" value="0x0"/>
Direction	<input type="text" value="In"/>	<input type="text" value="In"/>	<input type="text" value="In"/>	<input type="text" value="In"/>	<input type="text" value="In"/>	<input type="text" value="In"/>	<input type="text" value="In"/>	<input type="text" value="In"/>	

### Analog input

1	<input type="text" value="7.0373"/>	<input type="text" value=".1 second"/>	<span>Read</span>
2	<input type="text" value="0.0002"/>	<input type="text" value=".1 second"/>	<span>Read</span>
3	<input type="text" value="0.0000"/>	<input type="text" value="Passive"/>	<span>Read</span>
4	<input type="text" value="0.0000"/>	<input type="text" value="Passive"/>	<span>Read</span>
5	<input type="text" value="0.0000"/>	<input type="text" value="Passive"/>	<span>Read</span>
6	<input type="text" value="0.0000"/>	<input type="text" value="Passive"/>	<span>Read</span>
7	<input type="text" value="0.0000"/>	<input type="text" value="Passive"/>	<span>Read</span>
8	<input type="text" value="0.0000"/>	<input type="text" value="Passive"/>	<span>Read</span>

Strip charts Configure

### Waveform digitizer

Current point

# points

Time/point

Total time

Time/point

# points

First chan

# chans

Burst mode

Trigger

Retrigger

Trigger count

Clock

Continuous

Auto restart

Read rate

Read Read

Start Stop Done

Plots

### Waveform generator

Trigger

Retrigger

Trigger count

Clock

Repeat

Current point

# points

Frequency

Time/point

Total time

### Output 1

Enable

Waveform

Amplitude

Offset

Pulse width

### Output 2

Enable

Waveform

Amplitude

Offset

Pulse width

### User-defined waveforms

Frequency

Time/point

# points

Start Stop Running

### Pre-defined waveforms

Frequency

Time/point

# points

Plots

### Analog output

1	<input type="text" value="0.0000"/>	<input type="text" value="0.1000"/>	<span>&lt;</span> <span>&gt;</span>
2	<input type="text" value="0.0000"/>	<input type="text" value="0.1000"/>	<span>&lt;</span> <span>&gt;</span>

Configure

### Trigger

Mode