COBHAM

The most important thing we build is trust

PXI 3000 Series

Programming Manual

Longacres House
Six Hills Way
Stevenage SG1 2AN
UK

Document no. 47090/266 (PDF version)

Issue 1

23 April 2014

About this manual

Intended audience

Engineers who want to learn how to control PXI modules programmatically for making RF measurements.

This manual assumes the engineer is competent in his chosen language and concentrates on the detail of controlling the PXI modules rather than providing a known good architecture for the engineer to modify.

Document conventions

The following conventions apply throughout this manual:

CAPS Capitals are used to identify names of controls and panel markings.

[CAPS] Capitals in square brackets indicate hard key titles.

[Italics] Italics in square brackets indicate soft key titles.

Associated publications

If you want to...	Refer to...
Find information about soft front panels, drivers, application software, data sheets, getting started and user manuals for this and other modules in the 3000 Series.	**PXI Modules CD-ROM** Part no. 46886/028 Supplied with the module
Install modules into a rack, interconnect them, power up and install drivers.	**3000 Series PXI Modules Common Installation Guide** Part no. 46882/663 On the CD-ROM and at www.cobham.com/wireless
Set up a populated chassis ready for use.	**3000 Series PXI Modules Installation Guide for Chassis** Part no. 46882/667 On the CD-ROM and at www.cobham.com/wireless

Contents

Precautions

Anti-virus protection

Your instrument is supplied without anti-virus protection software. We recommend that if you connect to the internet or accept files from portable media (for example, memory sticks), you install a proprietary anti-virus software product. Provided that you install the software in accordance with the manufacturer's recommendations, your Cobham warranty will not be affected.

Cobham accepts no liability whatsoever for any loss or damage caused by viruses, worms, Trojan horses or other malicious programs downloaded through connection of the instrument to the internet or external storage device or portable media

Scope

This manual currently limits the scope to exclude Cobham phase 5 modules 3320, 3050A and 3070A, which require a different programming interface.

Getting started

This document provides a programming guide to using PXI 3000 system modules. All example code fragments are written in the C# language. It is not a complete reference guide; it assumes that you have installed correctly the PXI modules in a chassis and connected it to the PC. It also assumes that NI VISA and the PXI module drivers (NI VISA runtime without Measurement and Automation Explorer is installed as part of the module driver installation) have both been installed. These are available from the following links.

http://www.ni.com/visa/

http://www.aeroflex.com/ats/products/prodfiles/download.cfm?PFID=9840

A guide to installing PXI modules and module software is available from the following link:

http://www.aeroflex.com/ats/products/prodfiles/opsmanuals/Common_Installation_Guide.pdf

All Cobham software is primarily written to provide C-style windows DLLs. To provide support for other languages, wrappers for COM and .NET exist see Figure 1. This guide focuses on coding against the .NET wrapper using C# in a Visual Studio 2008 IDE. Cobham's .NET wrapper can be used with .NET 2.0 and above.

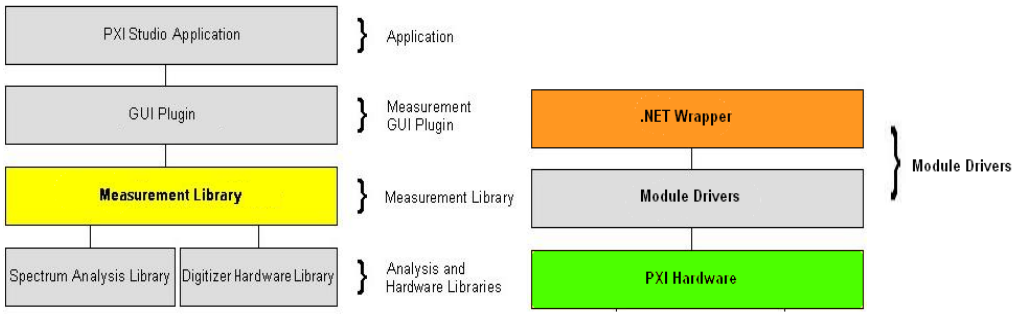


Figure 1 Cobham PXI HW/SW architecture

Adding a reference to the Cobham module drivers in Visual Studio

To start programming against the .NET module drivers a reference first needs to be added for the .NET assemblies within Visual Studio:

1 Load Visual Studio 2008.

2 Create a new project.

3 Make sure the Solution explorer is visible and right-click on References and click on 'Add Reference'. See Figure 2.

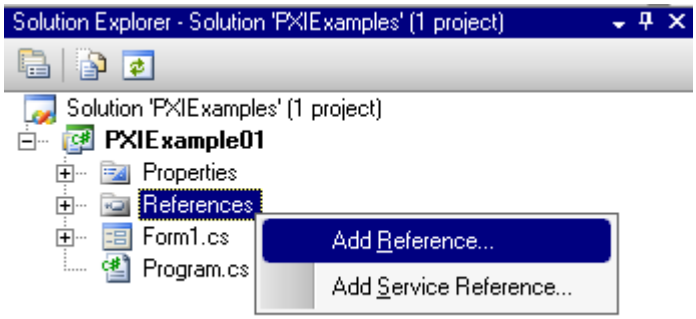


Figure 2 Adding a reference to the Module DLLs

4 From the dialog box that pops up click on the 'browse' tab and navigate to and add the following DLLs. See Figure 3.

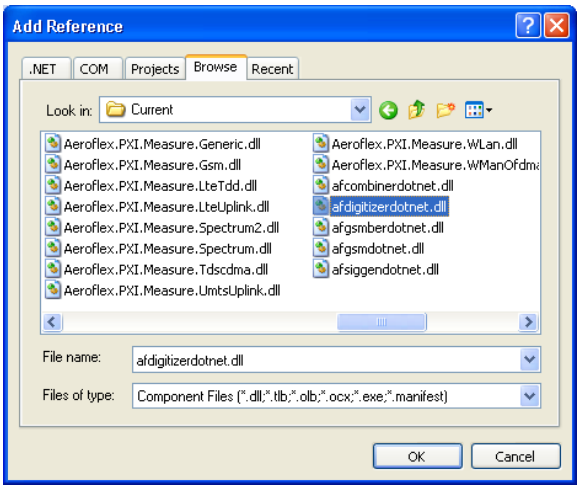


Figure 3 Selecting the .NET DLLs

File name	File path	Comment
afdigitizerdotnet.dll	C:\Program Files\Aeroflex\PXI\Current\	The.NET 303x (digitizer) dll.
afsiggendotnet.dll	C:\Program Files\Aeroflex\PXI\Current\	The.NET 302x (Signal Generator) dll.
afcombinerdotnet.dll	C:\Program Files\Aeroflex\PXI\Current\	The.NET 306x (Combiner) dll.
Aeroflex.PXI.Components.Interface.dll	C:\Program Files\Aeroflex\PXI\Current\	The common interface DLL is required by the other DLLs and therefore needs to be referenced.

Instantiating the digitizer, signal generator and combiner objects

Add the namespace for the digitizer, signal generator and combiner.

```
using afDigitizerDll_32Wrapper;
using afSigGenDll_32Wrapper;
using afCombinerDll_32Wrapper;
```

Now add three private member variables to a class to represent each of the three PXI modules.

```
private afSigGen SigGenObj           = null;
private afDigitizer digitizerObj     = null;
private afCombiner combinerObj       = null;
```

Now call the individual constructors to instantiate the objects that were defined.

```
combinerObj    = new afCombiner();
SigGenObj      = new afSigGen();
digitizerObj   = new afDigitizer();
```

Error handling

A method can be installed for the signal generator, digitizer and combiner that is called when either an error or warning occurs. In addition, each method call returns an integer error code that can be looked up to determine the error.

```
// Turn on and install a method to be called when an error/warning occurs in the digitizer.
digitizerObj.EventOnError.Enable_Set(AfTrue);
digitizerObj.EventOnError.Handler_Set(MyErrorMethodDigitizer,0);
digitizerObj.EventOnWarning.Enable_Set(AfTrue);
digitizerObj.EventOnWarning.Handler_Set(MyWarningMethodDigitizer, 0);

// Turn on and install a method to be called when an error/warning occurs in the Signal
Generator.
SigGenObj.EventOnError.Enable_Set(AfTrue);
SigGenObj.EventOnError.Handler_Set(MyErrorMethodSigGen,0);
SigGenObj.EventOnWarning.Enable_Set(AfTrue);
SigGenObj.EventOnWarning.Handler_Set(MyWarningMethodSigGen,0);

// Turn on and install a method to be called when an error/warning occurs in the combiner.
combinerObj.EventOnError_Set(AfTrue);
combinerObj.EventOnError.Handler_Set(MyErrorMethodCombiner, 0);
combinerObj.EventOnWarning_Set(AfTrue);
combinerObj.EventOnWarning.Handler_Set(MyWarningMethodCombiner, 0);
```

The method that is called when an error/warning happens is called with a series of parameters that allow for additional debug information. These are:

HandlerID — A unique module ID that is supplied during the method installation that allows for different modules of the same type to identified in a system.

Module Type — ENUM describing the module type throwing the error or warning. (this could include the LO as well as the digitizer module as they are booted together.

Error Code — An error code value.

Error Message — A message describing the error.

```
private void MyErrorMethodDigitizer(int HandlerId,
                                    afDigitizerDll_mtModuleType_t ModuleType,
                                    int errorCode, string errorMessage)
{
        MessageBox.Show ("Handler ID: " + HandlerId.ToString() +
                         "\nModule Type: " + ModuleType.ToString() +
                          "\nError Code: " + errorCode.ToString() +
                          "\nError Message: " + errorMessage);

}
```

```
private void MyErrorMethodSigGen(int HandlerId,
                                 afSigGenDll_mtModuleType_t ModuleType,
                                 int errorCode, string errorMessage)
{
       MessageBox.Show ("Handler ID: " + HandlerId.ToString() +
                        "\nModule Type: " + ModuleType.ToString() +
                        "\nError Code: " + errorCode.ToString() +
                        "\nError Message: " + errorMessage);

}
```

```
private void MyErrorMethodCombiner(int HandlerId,
                                   afCombinerDll_mtModuleType_t ModuleType,
                                   int errorCode, string errorMessage)
{
       MessageBox.Show ("Handler ID: " + HandlerId.ToString() +
                        "\nModule Type: " + ModuleType.ToString() +
                        "\nError Code: " + errorCode.ToString() +
                        "\nError Message: " + errorMessage);

}
```

***Note**: the warning methods are not shown, but they have the same method prototypes as their corresponding error methods*

***Note**: the methods above would look the perfect place to put a throw Exception(...) call in but these methods are actually call backs and are therefore called within the sequence of the module drivers code. Exceptions are swallowed within the call back, preventing module driver code from exiting incorrectly. Therefore they can be used for additional information only.*

Checking each returned error code and parsing it within a helper method can allow the programmer to adopt a traditional .NET throw exception, capture exception model.

```
int errorCode;
errorCode = SigGenObj.BootInstrument(SigGenLOResource, SigGenRFResource, AfFalse);
ErrorCheckSource(errorCode);
```

The combiner, signal generator and digitizer each have their own method to look up errors, therefore three separate helper methods can be used while programming to the Cobham PXI modules.

```
public void ErrorCheckSource(int errorCode)
{
        string messagebuffer = new string(' ', 1024);

        SigGenObj.ErrorMessage_Get(ref messagebuffer, 1024);
        if (errorCode < 0) // Error
        {
                throw new Exception("Source : " + messagebuffer);
        }
        else if (errorCode > 0) // Warning
        {
                // Log Warning
        }
}

public void ErrorCheckDigitizer(int errorCode)
{
        string messagebuffer = new string(' ', 1024);

         digitizerObj.ErrorMessage_Get(ref messagebuffer, 1024);
         if (errorCode < 0) // Error
         {
                throw new Exception("Digitizer : " + messagebuffer);
          }
           else if (errorCode > 0) // Warning
           {
                // Log Warning
           }
}

public void ErrorCheckCombiner(int errorCode)
{
        string messagebuffer = new string(' ', 1024);

        combinerObj.ErrorMessage_Get(ref messagebuffer, 1024);
        if (errorCode < 0) // Error
        {
                throw new Exception("Combiner : " + messagebuffer);
        }
        else if (errorCode > 0) // Warning
        {
                // Log Warning
        }

}
```

Booting Cobham PXI modules

Once an object reference for each module has been obtained, the modules need to be booted before they can be used. Each module in a PXI system has a separate unique address. Cobham makes use of the VISA layer for communicating to PXI modules and therefore each module has a unique VISA address. To discover the different module addresses NI's Measurement and Automation (MAX) application can be used. See Figure 4.

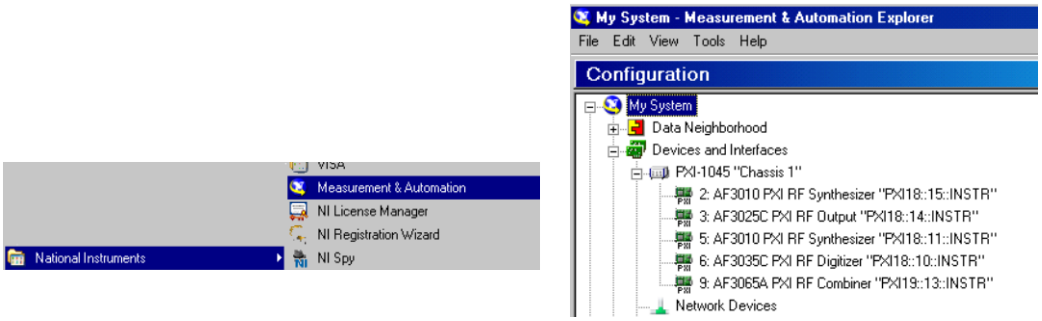


Figure 4 Use MAX to find the module VISA addresses

The 302x Signal Generator series and the 303x Digitizer series require a local oscillator (LO) to function. These are separate modules within the PXI system, part of the 301x series of local oscillators, and are also known as RF synthesizers. The 306x Combiner series does not require an LO to function. When the signal generator or digitizer is booted, the VISA address of the actual module and that of the LO supplying the tuning frequency is required. From looking at MAX it can be seen what the different VISA addresses are for each module within the author's system. Therefore they are added into the example code as follows.

```
// Set this to the correct address for your system.
private string SigGenLOAddress      = "PXI18::15::INSTR";
private string SigGenAddress        = "PXI18::14::INSTR";
private string DigitizerLOAddress   = "PXI18::11::INSTR";
private string DigitizerAddress     = "PXI18::10::INSTR";
private string CombinerAddress      = "PXI19::13::INSTR";
```

For historic reasons Cobham defines its own version of True and False and therefore for convenience when programming in c#:

```
public const int AfTrue = -1;
public const int AfFalse = 0;
```

EepromCacheEnable

To improve the speed at which the 3020, 3020A, 3025, 3030, 3030A and 3035 PXI modules boot, an eeprom cache file is stored on the PC and when the '<module>.EepromCacheEnable_Set' method is set to true, makes use of the cache file. However, the first-time boot is significantly extended.

Booting Cobham signal generators

When calling the signal generator 'bootInstrument' method, the first and second parameters are the VISA resource strings for the 301x and 302x modules. The third parameter is to indicate if the LO being used is an Cobham 301x module or a separate LO. In practice a 301x is always used and the parameter is set to AfFalse to indicate this. This manual does not cover how to use a non-Cobham LO.

```
errorCode = SigGenObj.EepromCacheEnable_Set(AfFalse);
ErrorCheckSource(errorCode);
errorCode = SigGenObj.BootInstrument(SigGenLOResource, SigGenRFResource, AfFalse);
ErrorCheckSource(errorCode);
```

Booting Cobham digitizers

When calling the digitizer 'bootInstrument' method the first and second parameters are the VISA resource strings for the 301x and 303x modules. The third parameter is to indicate if the LO being used is an Cobham 301x module or a separate LO. In practice a 301x is always used and the parameter is set to AfFalse to indicate this. This manual does not cover how to use a non-Cobham LO.

```
errorCode = digitizerObj.EepromCacheEnable_Set(AfFalse);
ErrorCheckDigitizer(errorCode);
errorCode = digitizerObj.BootInstrument(DigitizerLOResource, DigitizerRFResource, AfFalse);
ErrorCheckDigitizer(errorCode);
```

Booting Cobham combiners

The combiner does not require an LO and therefore only requires the VISA resource string for the combiner to be booted.

```
errorCode = combinerObj.EepromCacheEnable_Set(AfFalse);
ErrorCheckCombiner(errorCode);
errorCode = combinerObj.BootInstrument(CombinerAddress);
ErrorCheckCombiner(errorCode);
```

Checking options

Each module in a PXI system has a series of options that allow HW/SW related features to be enabled such as LO option 1 — 'Fast Switching' or digitizer option 100 — 'GSM/EDGE'. Each option can be queried as follows:

```
int SigGenhasOCXO             = AfFalse;
int SigGenhasIQCreator        = AfFalse;

errorCode = SigGenObj.LO.Options.CheckFitted(99, ref SigGenhasOCXO);
ErrorCheckSource(errorCode);
errorCode = SigGenObj.RF.Options.CheckFitted(100, ref SigGenhasIQCreator);
ErrorCheckSource(errorCode);
```

The 10 MHz reference and PXI module configurations

All PXI modules require to be phase and time aligned and therefore the front panel provides two parallel 10 MHZ sockets. The 10 MHz can either be generated from one of the LOs or all modules can be set up to receive an external 10 MHz reference. Cobham supplies a 3011 LO that uses an oven controlled crystal oscillator (OCXO) high accuracy 10 MHz reference.

The coding example below uses the physical setup as seen in Figure 5, a 3011 with OCXO generating a 10 MHz reference for the 3025C, a 3010 and the 3035C.

***Note**: the combiner does not require a 10 MHz reference.*

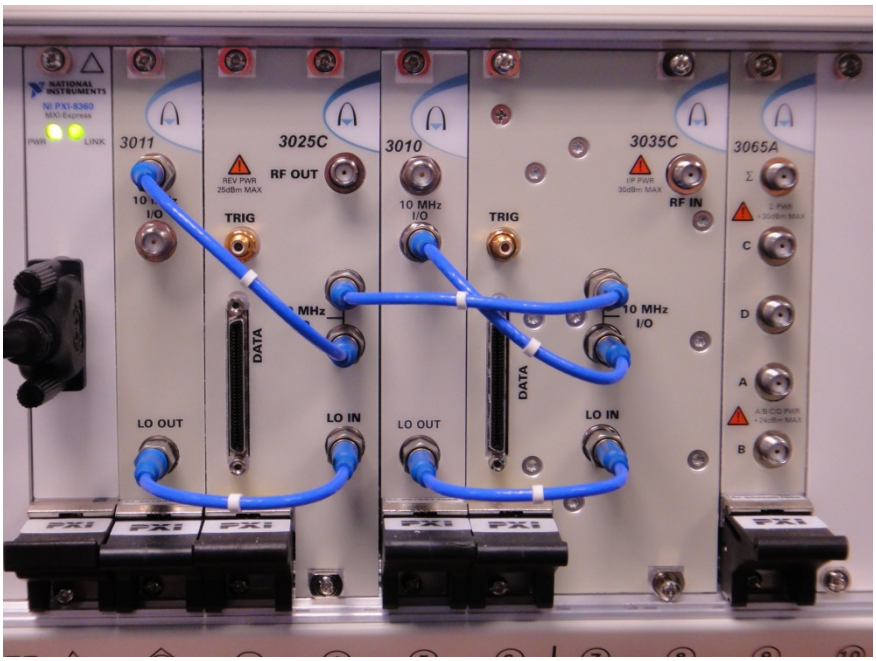


Figure 5 3011 generating 10 MHz

The code below checks to see if the LO associated with the signal generator has option 99 fitted (OCXO). If it does, it uses it to generate the 10 MHZ; if not, it throws an error. The digitizer is then set to 'external daisy' so that it makes use of the externally received 10 MHz reference rather than generating its own.

```
// Now setup the Siggen to generate the 10 MHz reference. Using the OCXO option if it's fitted.
errorCode = SigGenObj.LO.Options.CheckFitted(99, ref SigGenhasOCXO);
ErrorCheckSource(errorCode);

if (SigGenhasOCXO != 0)
{
        errorCode = SigGenObj.LO.Reference_Set(afSigGenDll_lormReferenceMode_t.afSigGenDll_lormOCXO);
        ErrorCheckSource(errorCode);

        errorCode =
        digitizerObj.LO.Reference_Set(afDigitizerDll_lormReferenceMode_t.afDigitizerDll_lormExternalDaisy);
        ErrorCheckDigitizer(errorCode);
}
else
{
        throw new ApplicationException("System Not Locked!!");
}
```

Once the LOs are configured to generate or receive the reference, the signal generator and digitizer need to be configured to make use of the external reference.

```
// If we're using a AF3020, AF3020A or AF3025 Signal Generator then we can also
// set the Voltage-controlled oscillator to be external for improved performance.
errorCode = SigGenObj.VCO.ExternalReference_Set(AfTrue);
ErrorCheckSource(errorCode);

// Now set the digitizer to use the external 10MHz reference.
errorCode =
digitizerObj.RF.ExternalReference_Set(afDigitizerDll_erExternalReference_t.afDigitizerDll_erLockTo10MHz);
ErrorCheckDigitizer(errorCode);
```

A method that can be used to boot PXI modules, handling different PXI VISA addresses and different 10 MHz references, can be found in the example code that is associated with this manual.

Checking the PXI system is locked to the 10 MHz reference

Once the system is booted and configured it is prudent to check that it is in lock. This can be achieved by querying the digitizer and signal generator's ReferenceLocked_Get(...) method. Looping a number of times also checks for intermittent lock issues that might be missed by only checking once. The signal generator should also be checked to make sure the VCO is set to external. If it cannot detect a 10 MHz reference it defaults to 'Internal', which is not of sufficient quality to provide an accurate signal.

```
int isDigReflocked = -99;
int isSigReflocked = -99;
int errorCode;

// Loop a number of times in case we have an intermitant lock.
for (int i = 0; i < 10; i++)
{
     // Check LO
     errorCode = digitizerObj.LO.ReferenceLocked_Get(ref isDigLoReflocked);
     ErrorCheckDigitizer(errorCode);

     errorCode = SigGenObj.LO.ReferenceLocked_Get(ref isSigLoReflocked);
     ErrorCheckSource(errorCode);

     // Siggen and Digizer are in Lock.
     errorCode = digitizerObj.RF.ReferenceLocked_Get(ref isDigReflocked);
     ErrorCheckDigitizer(errorCode);

     errorCode = SigGenObj.RF.ReferenceLocked_Get(ref isSigReflocked);
     ErrorCheckSource(errorCode);

     // Check VCO is set to external.
     errorCode = SigGenObj.VCO.ExternalReference_Get(ref isSigVCOExternal);
     ErrorCheckSource(errorCode);

        // Check to see if they're in lock or not.
     // Check to see if they're in lock or not.
     if ((isSigReflocked == AfFalse) || (isDigReflocked == AfFalse) ||
        (isSigLoReflocked == AfFalse) || (isDigLoReflocked == AfFalse) ||
     isSigVCOExternal == AfFalse)
     {
          throw new ApplicationException("PXI System Out of Lock. Check your 10 MHz
reference!");
     }
     // Give up the reminder of this threads timeslice so that we don't busy wait.
     System.Threading.Thread.Sleep(0);

}
```

10 MHz termination

The LOs can be configured in such a way so that they can terminate a 10 MHz reference signal. When this mode is set, the 10 MHz I/O sockets are set to 50 Ω and there is no output to the I/O socket. As an example, the code to terminate the 10 MHz reference at the LO associated with the digitizer in the system would be as follows. The physical layout of the system would have the 10 MHz cabled as seen in Figure 6.

```
// Now set the digitizer to use the external 10MHz reference.
errorCode = digitizerObj.RF.ExternalReference_Set(afDigitizerDll_erExternalReference_t.
afDigitizerDll_lormExternalTerminated);
ErrorCheckDigitizer(errorCode);
```

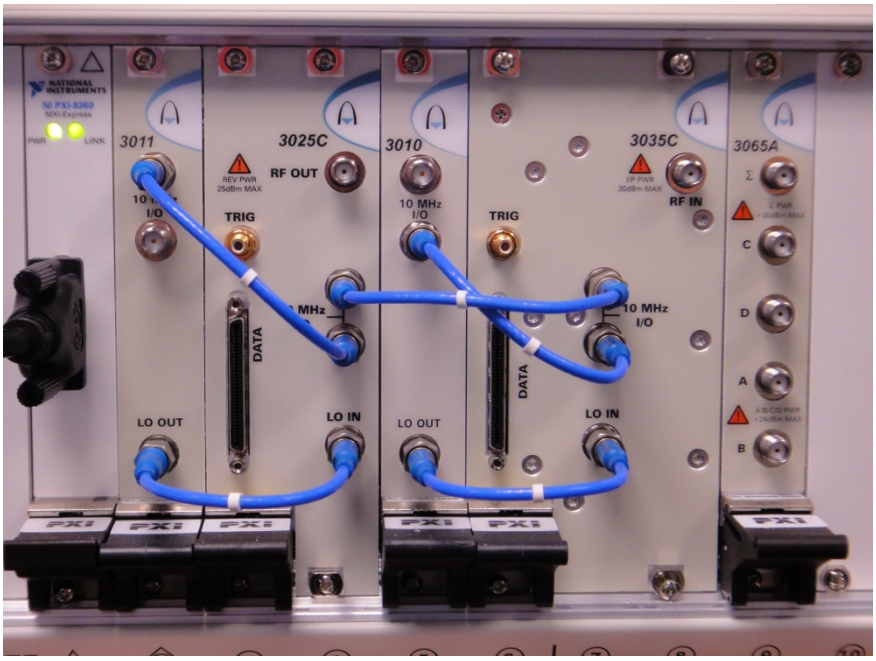


Figure 6 Physical cable configuration of the 10 MHz reference terminating on the 3010 associated with 3035C

Fast switching Option 01

For commercial reasons Cobham PXI has an option that enables switching frequencies to be completed in less than 250 µs typically. This is achieved by enabling Option 01 on the 3010 or 3011. With Option 01 enabled, loop bandwidth defaults to Normal (fast). With Option 01 disabled, loop bandwidth defaults to Narrow for better phase noise performance. When the loop bandwidth is set to Narrow the 301x can take up to 3 ms to fully settle, so unless phase noise is a priority it is better to always set it to Normal.

```
status = digitizerAf.LO.Options.CheckFitted(1, ref fastswitchingHWoptionfitted);
ErrorCheckDigitizer(status);

if (fastswitchingHWoptionfitted == Aeroflex303XConstant.AfTrue)
{
        // Fast switching is enabled, therefore LO Speed Synch is not needed and the LO loop
        // bandwidth can be set to Normal for best frequency switching times.

        status = this.digitizerAf.LoRfSpeedSyncEnable_Set(Aeroflex303XConstant.AfFalse);
        ErrorCheckDigitizer(status);
        status =
        this.digitizerAf.LO.LoopBandwidth_Set(afDigitizerDll_lolbLoopBandwidth_t.afDigitizerDll_lolbNormal)
        ;
        ErrorCheckDigitizer(status);
}
else
{
        // Fast switching is disabled, therefore LO Speed Synch needs to be enabled to make sure
        // the LO moves in time with the digitizer. Loop bandwidth is set to narrow for better
        // phase noise performance.
        status = this.digitizerAf.LoRfSpeedSyncEnable_Set(Aeroflex303XConstant.AfTrue);
        ErrorCheckDigitizer(status);
        status =
        this.digitizerAf.LO.LoopBandwidth_Set(afDigitizerDll_lolbLoopBandwidth_t.afDigitizerDll_lolbNarrow)
        ;
        ErrorCheckDigitizer(status);
}
```

Combiners

A combiner is an RF switch and allows for the DUT to be connected to the signal generator and digitizer.

Note: if cables are being routed through the combiner it needs to be booted, otherwise large losses in the RF path and in the signal generated or captured may not be seen.

Closing down the instrument

When the user software exits, the CloseInstrument() method should be called on the digitizer, combiner and signal generator objects if they were instantiated.

```
if (digitizerObj != null)
{
        errorCode = digitizerObj.CloseInstrument();
        ErrorCheckDigitizer(errorCode);
}

if (SigGenObj != null)
{
        errorCode = SigGenObj.CloseInstrument();
        ErrorCheckSource(errorCode);
}

if (combinerObj != null)
{
        errorCode = combinerObj.CloseInstrument();
        ErrorCheckCombiner(errorCode);
}
```

Signal generator basics

The signal generator has five modes of operation each providing a different way of controlling the signal generator. The modes are as follows:

- **Manual** — a simplified interface for the control of the signal generator. This mode provides a limited number of functions and properties, providing all the basic signal generator functionality.
- **ArbSeq** — a form of list mode: The purpose of this mode is to allow the definition of a sequence of ARB files. The emphasis of this mode is placed on ARB files and the sequence in which they are played.
- **Hopping** — a form of list mode: The emphasis of this mode is the control of a frequency and or level hop/sweep and as such provides the ability to define the dwell duration for each list point independently of each other. It does not provide channelized ARB file selection or control.
- **GroupSeq** — a method of sequencing ARB files. Specifically designed to allow basic power control of DUTs.
- **Full** — this is the default and is provided for legacy support. It provides full access to all channelized and list mode controls. This mode requires comprehensive knowledge and understanding of the signal generator to achieve the functionality provided by any of the other modes of operation.

Setting the mode can be achieved as follows:

```
// Set the Signal Generator Mode to Full.
errorCode = SigGenObj.Mode_Set(afSigGenDll_mMode_t.afSigGenDll_mManual);
ErrorCheckSource(errorCode);
```

Leveling mode

There are four leveling modes used in the 302x signal generator:

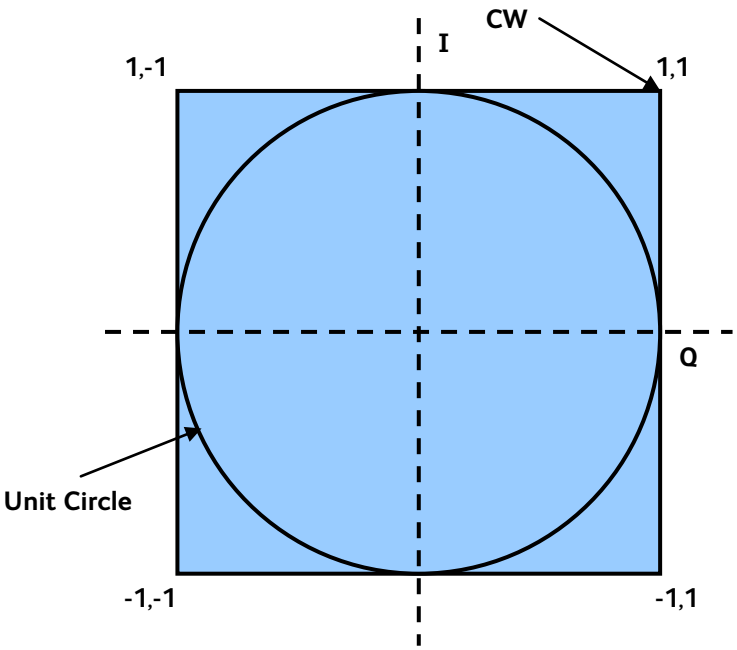
- Auto
- Peak
- RMS
- Frozen

AIQ file level information

An AIQ file can be opened using a text editor. The header in the file contains the following information — key items of interest have been highlighted.

```
[File]

Date=10/18/2011

Time=09:03:53

FileVers=

Packager=44540/198

PackSWVers=01.21

Samples=301

Title=PowerRamp50kHz

SampleRate=50000

Description=

RMS=4729

RelRMS=-7.78165

CrestFactor=  8.4113

LevelMode=IQDefault

AlcBW=Narrow

SignalBandwidth=0
```

CW vs. ARB file



IQ information can be anywhere in the 'IQ' square, with I or Q having values in the range -1 to +1.

CW

When a 302x series signal generator generates a CW signal, the IQ modulator is set with the values I=1 and Q=1.

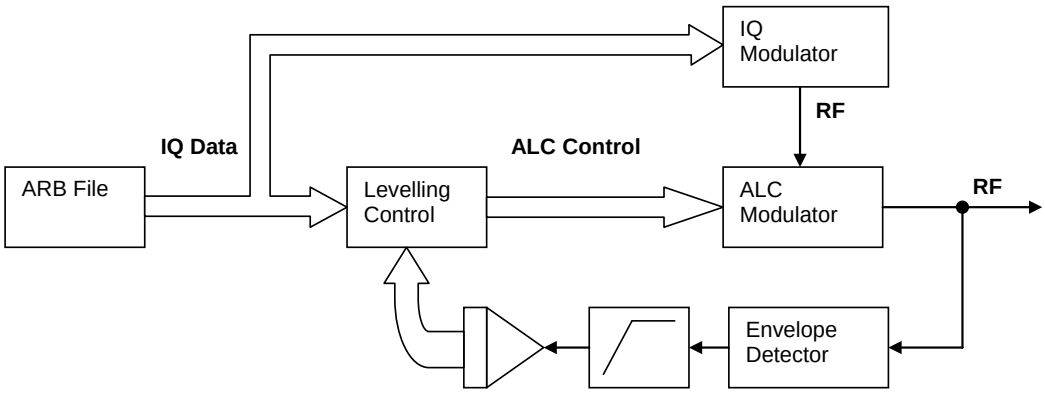
Arb file

When an ARB file is created, it contains multiple IQ pairs. Usually (but not always), these IQ pairs are such that the combination of I and Q are within the 'Unit Circle' (where (I2+Q2) = 1).

The Unit Circle is -3 dB relative to the 1,1 CW point.

302x leveling control

The 302x Automatic leveling control is a simple feedback loop.



It performs 'live' leveling of the modulated signal.

The IQ modulated RF is fed to an envelope detector. The voltage output of the envelope detector is passed through a low pass filter and measured with an ADC.

The ADC output is compared to the IQ data, and the ALC modulator (variable attenuator) is adjusted so that the level of the signal is the same as the level of the IQ file.

Auto leveling mode

This enters the desired RMS power level. The output should be a signal with an RMS level the same as that you have entered. Using the relative RMS value in the example header files above:

```
// Set the level to be -10 dBm.
errorCode = SignalGeneratorObj.Manual.Level_Set-30);
ErrorCheckSource (errorCode);
```

The signal generator sets the attenuators as if it were trying to achieve a CW power 7.78165 dB *higher* than -10 dBm (i.e. -2.21835 dBm). The RelRMS is the 'Relative Level of the RMS' of the IQ data to the 1,1 point on the IQ square (CW).

The leveling loop is then used to measure the instantaneous power of the ARB file as it is playing. From the IQ data, it knows the expected power level at the output — it adjusts the ALC modulator to correct to reach the desired RMS power.

Peak leveling mode

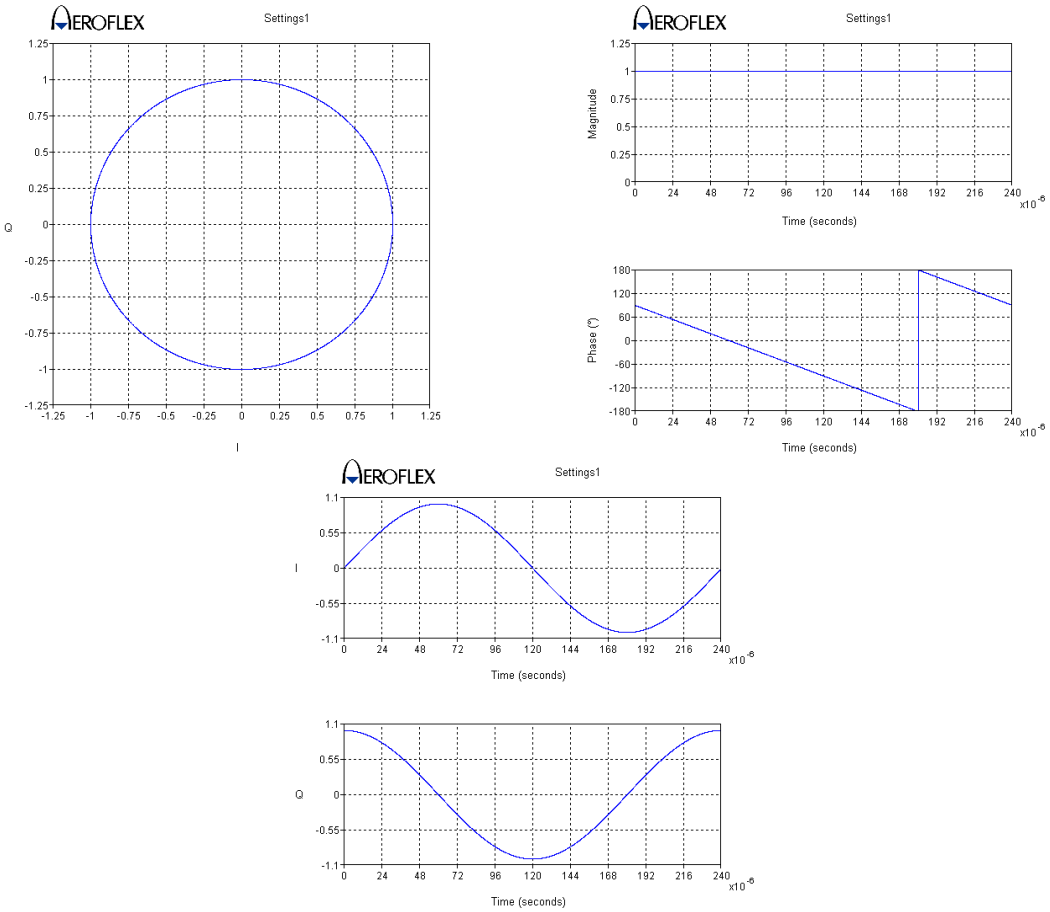
This power level is equivalent to the 1,1 point on the IQ square (IQ magnitude of 1.414).

The ALC then adjusts the power so that the IQ value of the ARB file is correct relative to the 1,1 value.

For example, a file is played that has a maximum IQ value on the unit circle ((I2+Q2) = 1), this is -3dB from the PEAK power.

The PEAK power that the Signal generator allows is 1,1 (magnitude of 1.414).

Example using a Rotating IQ vector on the unit circle:



This ARB file has IQ values so that it produces a rotating vector: it has a constant magnitude of 1 (which is -3 dB from the 1,1 point on the IQ square).

RMS leveling mode

RMS leveling mode is essentially the same as Peak leveling mode except that you can specify the RMS level of the signal relative to the 1,1 peak:

```
// Set the Level Mode to be RMS
errorCode =
SignalGeneratorObj.Manual.LevelMode_Set(afSigGenDll_lmLevelMode_t.afSigGenDll_lmRms);
ErrorCheckSource(errorCode);

// Set the RMS value
errorCode = SignalGeneratorObj.Manual.RMSdBc_Set(-3);
ErrorCheckSource(errorCode);
```

Frozen mode

In frozen mode, the current settings for the ALC are 'frozen, based on the current value' and the ALC modulator is kept to this fixed value. The step attenuators are also 'frozen'. The output power varies, dependent upon the IQ data. Changing the set power level does NOT change the actual output power level until 'Frozen Mode' is exited.

Detector Zero user calibration

Detector Zero calibration is run in the factory and sets the leveling detector to zero to ensure that the module meets the level accuracy specified in the data sheet. Detector Zero user calibration needs to be run once per module boot up. Detector Zero is not required to cope with temperature changes since that is taken care of by temperature correction that happens automatically and transparently every five minutes. The temperature correction process does not affect the 3020's time to respond because the 3020 carries on using the old data until it has been updated.

```
errorCode = SigGenObj.Calibrate.Detector.Zero();
ErrorCheckSource(errorCode);
```

IQ user calibration

Factory calibration of the IQ modulator is used to calculate and store IQ modulator errors that are used in the 302x to correct the I and Q data to optimize modulation accuracy. IQ user calibration is also available to give further improvement in performance for the specific conditions of operation and ensure that guaranteed specifications of carrier leak and image suppression are met. The module calibrates at the current frequency, or at a range of frequencies, and stores the results so that if you change frequency and return again, the calibration still applies. During the user calibration process the 3020 output is disabled. For optimum performance the user calibration should be done following a period of warm-up, and periodically if the operating temperature is expected to change by more than 5˚C. The following types of user IQ calibration are available:

- Cal All Bands
- Cal Current Frequency
- Cal Selected Bands

```
// Calibrate All Bands.
errorCode = SigGenObj.Calibrate.IQ.AllBands();
ErrorCheckSource(errorCode);

// Calibrate the current frequency Only.
errorCode = SigGenObj.Calibrate.IQ.CurrentFrequency();
ErrorCheckSource(errorCode);

double startfreq = 99;
double stopfreq = 99;

// Interigate the signal generator to find out the frequency range in Hz of the Band 0
errorCode = SigGenObj.Calibrate.IQ.GetBandInformation(0,ref startfreq,ref stopfreq);
ErrorCheckSource(errorCode);

// Calibrate Band 0 76MHz - 93.5MHz
errorCode = SigGenObj.Calibrate.IQ.SelectedBand(0);

ErrorCheckSource(errorCode);
```

Loss compensation

No loss compensation exists for the 302x Series module and so any accounting for loss needs to be manually applied.

Manual mode

This mode of operation provides a simplified interface for the control of the signal generator. A limited number of functions and properties is available, providing all the basic signal generator functionality:

- RF frequency
- RF level
- RF levelling mode
- Modulation — CW, AM, FM and ARB
- ARB control.

Generating a CW tone at a given power and frequency

To generate a basic CW tone from the signal generator the mode needs to be set to manual, the modulation set to CW, the power and frequency set and the RF output turned on:

```
// Set the MODE to manual
errorCode = SignalGeneratorObj.Mode_Set(afSigGenDll_mMode_t.afSigGenDll_mManual);
ErrorCheckDigitizer(errorCode);

// Set the modulation source to be a CW tone.
errorCode = SignalGeneratorObj.Manual.ModulationSource_Set(
             afSigGenDll_msModulationSource_t.afSigGenDll_msCW);
ErrorCheckDigitizer(errorCode);

// Set the frequency to 890 MHz, the level to -30 and the RF to be on.
errorCode = SignalGeneratorObj.Manual.Frequency_Set(890e6);
ErrorCheckDigitizer(errorCode);
errorCode = SignalGeneratorObj.Manual.Level_Set(-30);
ErrorCheckDigitizer(errorCode);
errorCode = SignalGeneratorObj.Manual.RfState_Set(AfTrue);
ErrorCheckDigitizer(errorCode);
```

Generating a waveform in *IQCreator* ®

Cobham uses a software application called *IQCreator* ® to generate waveforms that can be played out of the arbitrary waveform generator (ARB) within the 302x series of PXI signal generators. It is a free download from Cobham but to play a generated waveform, the correct format option needs to be installed. For the purposes of this manual, it is now assumed a GSM waveform with a marker bit set at the start of the GSM frame called *GSM-NORMAL-1SLOT-TSC5.aiq* has been created. This can either be created using *IQCreator* following the guide:

http://www.aeroflex.com/ats/products/prodfiles/software/IQCreator_User_Guide_14.pdf

or downloaded as part of the PXIExample01.zip available from:

https://stv-subversion/Applications/PXI/Resources/PXI Module Software/Application Notes/PXI 3000 Programming Manual/PXI Example 01/PXIExample 01.suo

Playing out a waveform at a known power and frequency

To play out the *GSM-NORMAL-1SLOT-TSC5.aiq* waveform, the steps are similar to that of generating a CW tone except that a waveform needs to be loaded, played out and the modulation source set to 'Arb':

```
// Set the MODE to manual
errorCode = SignalGeneratorObj.Mode_Set(afSigGenDll_mMode_t.afSigGenDll_mManual);
ErrorCheckSource (errorCode);

// Set the modulation source to ARB for waveform playback.
errorCode = SignalGeneratorObj.Manual.ModulationSource_Set(
            afSigGenDll_msModulationSource_t.afSigGenDll_msARB);
ErrorCheckSource (errorCode);

// Now load the AIQ file into the signal Generator.
errorCode = SignalGeneratorObj.ARB.Catalogue.AddFile(fileName);
ErrorCheckSource (errorCode);

// Now set the AIQ file to be played out.
errorCode = SignalGeneratorObj.Manual.ArbFile_Set("GSM-NORMAL-1SLOT-TSC5.aiq")
ErrorCheckSource (errorCode);

// Set the frequency to 890 MHz, the level to -30 and the RF to be on.
errorCode = SignalGeneratorObj.Manual.Frequency_Set(890e6);
ErrorCheckSource (errorCode);
errorCode = SignalGeneratorObj.Manual.Level_Set(-30);
ErrorCheckSource (errorCode);
errorCode = SignalGeneratorObj.Manual.RfState_Set(AfTrue);
ErrorCheckSource (errorCode);
```

***Note**: take care when loading ARB files to make sure they are not already loaded.*

***Note**: certain actions cannot be performed while a waveform is playing. Changing the ARB file that is to be played out while it is currently playing out an ARB file results in an error.*

The ARB file playback can be stopped using the '*ArbStopPlaying()*' method and the RF output turned off using the '**`RfState_Set (...)`**' method:

```
// Stop the AIQ file from being played out.
errorCode = SignalGeneratorObj.Manual.ArbStopPlaying();
ErrorCheckSource(errorCode);

// Turn RF off.
errorCode = SignalGeneratorObj.Manual.RfState_Set(AfFalse);
ErrorCheckSource(errorCode);
```

Digitizer basics

The digitizer captures IQ at a known frequency and input range. The digitizer also supports a complex list mode whereby IQ can be captured at different frequencies and input power levels in sequence. Once the IQ has been made available it can be fed into the analysis libraries, and results obtained. This section describes the basic operation of the digitizer, and includes calculating the GSM-specific results from the capture of a GSM signal generated from the signal generator using knowledge from the previous section.

Sample rate

The sample rate is the number of samples the digitizer captures per second. The higher the sample rate then the larger the amount of data that is produced during a capture. It is important to set the sample rate high enough so that each symbol within the signal being captured is captured but not too high such that too much data is produced. The digitizer supports a modulation mode that allows the sample rate to be configured correctly for a given format. It also supports a 'generic mode' whereby the sample rate can be specified manually. This manual makes use of the generic mode as it gives the greatest freedom.

***Note**: if the decimation ratio is changed internally the sample rate is changed to match this, but this is not reflected in the decimations ratios get() method. This is also true when using the sample rates get method if you have set the decimation value.*

```
double SampleRate = 1083333.33; // Sample Rate For GSM.
errorCode =
digitizerObj.Modulation.Mode_Set(afDigitizerDll_32Wrapper.afDigitizerDll_mmModulationMode_t.afDig
itizerDll_mmGeneric);
ErrorCheckDigitizer(errorCode);

errorCode = digitizerObj.Modulation.GenericSamplingFrequency_Set(SampleRate);
ErrorCheckDigitizer(errorCode);
```

Data IQ resolution

Depending on the sample rate and decimation, the IQ data that has been captured can be transferred to the controller PC with 16-bit or 32-bit resolution. This affects the time to transfer the data and therefore should be set appropriately. This can be easily achieved in software using the Resolution_Set(...) command, which can be configured to Auto whereby the IQ resolution is chosen based on the decimation ratio and sample rate.

```
errorCode =
digitizerObj.Capture.IQ.Resolution_Set(afDigitizerDll_32Wrapper.afDigitizerDll_iqrIQResolution_t.
afDigitizerDll_iqrAuto);
ErrorCheckDigitizer(errorCode);
```

***Note**: for UMTS, if a sample rate of 30.72 MHz is defined, the module drivers then set the decimation ratio to 2 and the rules used in the driver set a 32-bit resolution. In this case, a 16-bit resolution would be preferable.*

The general rule is 32-bit is available for sampling rates < MaxSamplingRate MHz/8, otherwise 16-bit is used.

Setting the frequency and expected input level

The frequency can be set using the following commands:

```
double Frequency = 890e6; // 890 MHz
errorCode = digitizerObj.RF.CentreFrequency_Set(Frequency);
ErrorCheckDigitizer(errorCode);
```

And the level can be set using the following command:

```
double RFInputLevel = -20; // -20 dBm
errorCode = digitizerObj.RF.RFInputLevel_Set(RFInputLevel);
ErrorCheckDigitizer(errorCode);
```

***Note**: when setting the input level take care to include any headroom for signals with a high peak-to-average ratio as the digitizer does not have an explicit API call for this.*

Data type

The data captured can be set to either IQ or IF using the following method:

```
errorCode = digitizerObj.Capture.SampleDataType_Set(
                            afDigitizerDll_sdtSampleDataType_t.afDigitizerDll_sdtIQData);
ErrorCheckDigitizer(errorCode);
```

A different set of API commands is used to capture and extract IQ or IF samples. This manual focuses on collecting IQ samples.

Triggering

Cobham PXI provides a number of different triggering techniques. Triggering can be set to 'free run', 'RF triggered', or 'externally triggered'. 'Free run' or 'software triggered' captures immediately once the capture method is issued. 'RF triggered' or 'internal triggered' captures only when certain triggering conditions are met, such as when a rising edge is seen above a threshold level. 'External triggering' causes a capture to occur once a TTL signal is seen on a defined external trigger line. It is the task of the programmer to handle timeout conditions.

Software trigger or free run trigger

The software trigger can either be Immediate or Armed. When in Immediate mode, a capture is made when the IQ data is retrieved from the digitizer using the CaptMem(...) method. When in Armed mode, the capture is triggered when the TriggerArm method is sent. A check can then be made using the CaptComplete method for when the capture has fully completed (this allows the programmer to analyze previously captured data while waiting, thus gaining an improvement in speed).

```
errorCode =
digitizerObj.Trigger.Source_Set(afDigitizerDll_tsTrigSource_t.afDigitizerDll_tsSW_TRIG);
ErrorCheckDigitizer(errorCode);

errorCode =
digitizerObj.Trigger.SwTriggerMode_Set(afDigitizerDll_swtSwTrigMode_t.afDigitizerDll_swtArmed)
ErrorCheckDigitizer(errorCode);

digitizerObj.Capture.IQ.TriggerArm(TotalDwellSamples)
```

RF triggered or internal triggered

In the code below an RF trigger is configured to capture 5400 samples (based on a 5 ms capture at a sample rate of 1 080 000 Hz):

```
using System.Diagnostics;

Stopwatch sw = new Stopwatch();

errorCode =
digitizerObj.Trigger.Source_Set(afDigitizerDll_tsTrigSource_t.afDigitizerDll_tsINT_TRIG);
ErrorCheckDigitizer(errorCode);

errorCode = digitizerObj.Trigger.IntTriggerAbsThreshold_Set(-30); // -30 dBm
ErrorCheckDigitizer(errorCode);

errorCode = digitizerObj.Trigger.PreEdgeTriggerSamples_Set(100); // -92 µs
ErrorCheckDigitizer(errorCode);

errorCode = digitizerObj.Capture.IQ.TriggerArm(5400);
ErrorCheckDigitizer(errorCode);

sw.Reset();
sw.Start();

do
{
        System.Threading.Thread.Sleep(0); // Don't busy wait.
        errorCode = digitizerObj.Capture.IQ.CaptComplete_Get(ref CaptureComplete);
        ErrorCheckDigitizer(errorCode);
        elapsedTime = sw.ElapsedMilliseconds;
} while (CaptureComplete != AfTrue && (elapsedTime < measTimeout));

// Check to see if we exited the loop due to a timeout.
if (elapsedTime >= measTimeout)
{
        throw new ApplicationException("Timed Out!");
}
```

Trigger delay

The trigger delay is the number of samples to include before the trigger. Trigger delay is often expressed in time so the formula below can be used to calculate the number of samples:

Number of samples = sample rate (Hz) x trigger delay (s)

```
errorCode = digitizerObj.Trigger.PreEdgeTriggerSamples_Set(100); // -92 µs
ErrorCheckDigitizer(errorCode);
```

PXI routing

Cobham provides extensive routing capabilities. This is achieved by using the routing SetConnect() method to connect two lines. One line is the input or where the signal is to be routed from. The other line is the output, which is where the line is to be routed to. Once the connection has been made the output needs to be enabled using the SetOutputEnable() method. Connections can be made on the signal generator or on the digitizer. In the example below the marker 1 bits(input) within the ARB file are routed onto the PXI_TRIGGER_0 line(output) and then the output enabled.

```
// Configure the ARB marker 1 to be routed to PXI trigger 0.
errorCode =
SigGenObj.RF.Routing.SetConnect(afSigGenDll_rmRoutingMatrix_t.afSigGenDll_rmPXI_TRIG_0,
afSigGenDll_rmRoutingMatrix_t.afSigGenDll_rmARB_MARKER_1);
ErrorCheckSource(errorCode);

// Enable the routing.
errorCode =
SigGenObj.RF.Routing.SetOutputEnable(afSigGenDll_rmRoutingMatrix_t.afSigGenDll_rmPXI_TRIG_0,
AfTrue);
ErrorCheckSource(errorCode);
```

This causes the PXI_TRIGGER_0 line to go high every time the marker 1 bits are set high within an ARB file.

External trigger

An external trigger causes the capture to be started whenever the configured trigger line goes high. The example below routes the marker 1 bits within an ARB file to PXI_TRIGGER_0 and then set external trigger to be PXI_TRIGGER_0. This causes a capture to take place whenever the ARB file being played out encounters the marker 1 bits. In the example code the sample waveform *GSM-NORMAL-1SLOT-TSC5.aiq* has marker 1 set to high on the first few samples. This causes the capture to align to the start of the playback and capture the entire frame synchronized to timeslot 0.

```
using System.Diagnostics;

Stopwatch sw = new Stopwatch();

// Route and enable the ARB marker 1 to the PXI TRIG 0
errorCode =
SigGenObj.RF.Routing.SetConnect(afSigGenDll_rmRoutingMatrix_t.afSigGenDll_rmPXI_TRIG_0,
afSigGenDll_rmRoutingMatrix_t.afSigGenDll_rmARB_MARKER_1);
ErrorCheckSource(errorCode);

errorCode =
SigGenObj.RF.Routing.SetOutputEnable(afSigGenDll_rmRoutingMatrix_t.afSigGenDll_rmPXI_TRIG_0,
AfTrue);
ErrorCheckSource(errorCode);

// Set the digitizer to trigger of PXI TRIG 0.
errorCode =
digitizerObj.Trigger.Source_Set(afDigitizerDll_tsTrigSource_t.afDigitizerDll_tsPXI_TRIG_0);
ErrorCheckDigitizer(errorCode);
// Set a 92µs pre trigger.
errorCode = digitizerObj.Trigger.PreEdgeTriggerSamples_Set(100); // -92µs
ErrorCheckDigitizer(errorCode);

errorCode = digitizerObj.Capture.IQ.TriggerArm(NumberOfIQSamples);
ErrorCheckDigitizer(errorCode);

sw.Reset();
sw.Start();
// Poll until the capture is complete or we're exceeded our timeout value.
do
{
        System.Threading.Thread.Sleep(0); // Don't busy wait.
        errorCode = digitizerObj.Capture.IQ.CaptComplete_Get(ref CaptureComplete);
        ErrorCheckDigitizer(errorCode);
        elapsedTime = sw.ElapsedMilliseconds;
} while (CaptureComplete != AfTrue && (elapsedTime < measTimeout));

// Check to see if we exited the loop due to a timeout.
if (elapsedTime >= measTimeout)
{
        throw new ApplicationException("Timed Out!");
}
```

Capturing IQ

Cobham digitizers specify captures in samples. Therefore the number of samples for a given time period needs to be calculated from the sample rate. Appropriate float arrays can then be defined to hold the I and Q data.

Number of samples = sample rate(Hz) x capture time (s)

'*<INST>.Capture.IQ.CaptMem(...)*' then instructs the digitizer to capture IQ samples.

```
double CaptureTime = 0.005; // 5 ms
uint NumberOfIQSamples = (uint)(CaptureTime * SampleRate);

float[] Idata = new float[NumberOfIQSamples];
float[] Qdata = new float[NumberOfIQSamples];

errorCode = digitizerObj.Capture.IQ.CaptMem(NumberOfIQSamples, Idata, Qdata);
ErrorCheckDigitizer(errorCode);
```

The CaptMem method blocks until the Idata and Qdata buffers are full or an error occurs.

Checking for ADC overload

Once a capture is complete a check on the validity of the data needs to be performed by checking to see if the digitizer overloaded during the capture.

```
int overloaded = -99;
errorCode = digitizerObj.Capture.IQ.ADCOverload_Get(ref overloaded);
ErrorCheckDigitizer(errorCode);

if (overloaded == AfTrue)
{
        throw new ApplicationException("Overloaded");

}
```

Correcting the signal

Depending on the HW configuration and frequency setting, the captured IQ data needs a correction factor applied so that Power can be computed using:

$$\text{Power (dBm)} = 10 \log (I^2 + Q_2) + \text{correction factor (dB)}$$

The correction factor can be retrieved as follows:

```
// Get Correction Factor
double levelcorrection = -99;

errorCode = digitizerObj.RF.LevelCorrection_Get(ref levelcorrection);
ErrorCheckDigitizer(levelcorrection);
```

Aborting the capture

If an error occurs during the capture, it is good practice to call the capture abort method to put the digitizer into a known state:

```
// If an error occured abort the current capture to maintain a known state for the digitizer.
errorCode = digitizerObj.Capture.IQ.Abort();
ErrorCheckDigitizer(levelcorrection);
```

Making an analysis library measurement

The analysis libraries take IQ data as input, perform a number of user-defined measurements and then return the results. Cobham supports a different DLL/library for each format. Some formats also support loop-back BER and are supported with a different library.

The analysis libraries are installed as part of installing a measurement suite. All measurement suites can be downloaded from the following location:

http://www.aeroflex.com/ats/products/product/PXI/Application_Software/Measurement_Suites~531

Adding a reference to the GSM Analysis Library in Visual Studio

To start programming against the .NET analysis libraries a reference needs to be added for the .NET assemblies within Visual Studio. This can be accomplished:

1 Load Visual Studio 2008.

2 Create a new project.

3 Make sure the Solution explorer is visible, right-click on References and click on 'Add Reference'. See Figure 7.

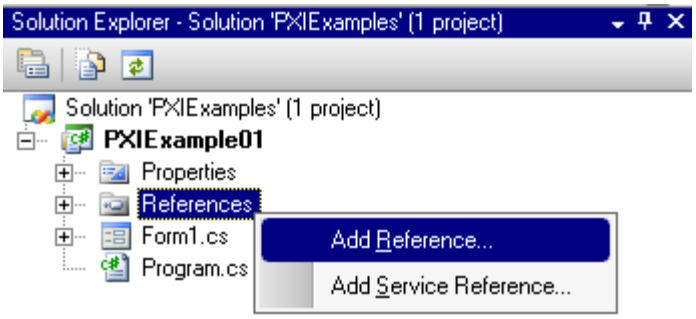


Figure 7 Adding a reference to the GSM Analysis Library DLL

4 From the dialog box that pops up click on the 'browse' tab and navigate to and add the following DLLs. See Figure 8.

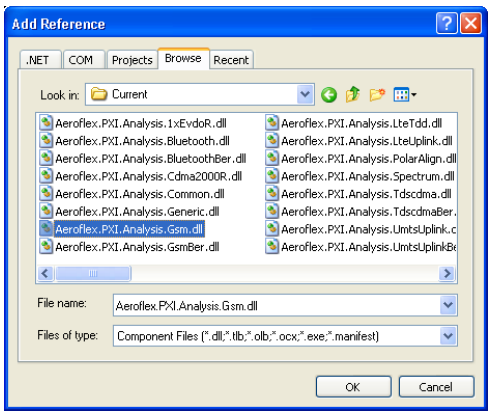


Figure 8 Selecting the GSM Analysis Library .NET DLL

Adding a reference and instantiating the GSM analysis library

Add the namespace for the newly added GSM analysis library:

```
using Aeroflex.PXI.Analysis;
```

Now add a private member variable to a class to represent the GSM analysis library:

```
private Gsm         GsmObj = null;
```

Now call the GSM analysis library constructor to instantiate the private member variable GsmObj you added above:

```
GsmObj = new Gsm();
```

Error handling

All Cobham .NET analysis libraries support .NET exceptions and therefore no explicit error checking needs to be made. All code should be written to handle .NET exceptions though.

Configuring the GSM analysis library

Before analysis can take place a number of GSM analysis library specific settings need to be made. All configuration settings can be found in the GSM analysis library Help file that is installed when the GSM measurement suite is installed. As a minimum, for the example the sample rate is set to that used to capture the signal (1.08 MHz, see 'Digitizer basics' on page 25).

AutoAnalysisBurstType sets the analysis library to determine automatically if the burst is a GSM burst or an EDGE burst, and AutoAnalysisTSC sets the analysis library to detect the training sequence (TSC) of the analyzed burst rather than use a specific value.

```
GsmObj.Configuration.SamplingFreq           = 1.08e6;
GsmObj.Configuration.AutoAnalysisBurstType  = true;
GsmObj.Configuration.AutoAnalysisTSC        = true;
```

Making a GSM analysis library measurement

All analysis libraries perform the measurement(s) by calling the Analyse method. The inputs to this method are the measurements to perform and the I and Q data retrieved from the digitizer. The method is overloaded with additional parameters to provide an offset, and length parameters that allow only portions of the I and Q data to be analyzed. This of particular interest when making large captures where you want to limit the amount of analysis performed. In the example below a modulation measurement is performed on the I and Q data.

```
GsmObj.Analyse(AnalysisType.Modulation | AnalysisType.LocateBurst, Idata, Qdata);
```

Retrieving results

Once the `Analyse` method returns, the results are available. No polling is required. Results are available under the results property from the instantiated analysis library object.

```
BurstType bursttype;
long powerprofile     = 999;
float freqError       = 999;
float modError        = 999;
float power           = 999;
char detectedTsc      = 'z';

power          = GsmObj.Results.BurstPower;
freqError      = GsmObj.Results.FrequencyError;
bursttype      = GsmObj.Configuration.BurstType;
modError       = GsmObj.Results.ModulationErrorRms;
powerprofile   = GsmObj.Results.PowerProfileComplete;
detectedTsc    = GsmObj.Results.DetectedTSC;
```

***Note**: analysis libraries are not thread-safe so only one thread should call Analyse at any given time. If analysis is to done concurrently then multiple instances of analysis libraries are required.*

Useful links

Cobham PXI : http://www.aeroflex.com/ats/products/category/PXI.html

VISA IVI Foundation : http://www.ivifoundation.org/docs/vpp43.pdf