EcoGamma Software Development Kit

7072686

Programmer's Manual

Canberra Industries, 800 Research Parkway, Meriden, CT 06450 USA.
Tel: 203-238-2351 FAX: 203-235-1347 http://www.canberra.com/.
Canberra is an AREVA company.

The information in this document describes the product as accurately as possible, but is subject to change without notice.

Printed in the United States of America.

For Technical Assistance, please call our Customer Service Hotline at 800-255-6370 or email techsupport@canberra.com.

Table of Contents

1. Introduction

The EcoGamma Communications Software Development Kit (SDK) allows development of software that communicates with and controls Canberra's EcoGamma dose probes. The SDK provides access to setup, control, and data access functions. The SDK contains libraries, source code, project files, and examples. The SDK consists of two technology implementations Java and .NET.

These two technologies were chosen because of the following:

- They are supported on most platforms and operating systems
- They do not require recompilation for any type of processor. Simply copy the libraries on your platform and consume them.
- They take advantage of managed code environments
- They can be imported into other programming environments like Groovy, Scala, F#, LabView, MatLab, Visual Basic, C++, and more.
- They are object oriented.

The classes and methods provided by the libraries for each technology are very similar to each other.

The Software

The SDK is capable of doing the following:

- Connecting to, configurating and collecting data from an EcoGamma.

System Requirements

Software

The following are required in order to use this software.

For Java development:

- Eclipse JDT 3.6 or greater.
- Java 5.0 or greater.

For Non-Windows .NET development:

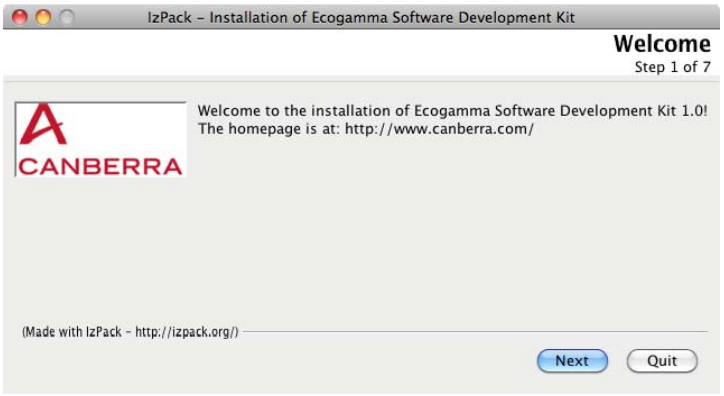
- MonoDevelop 2.4 or greater.
- Mono Framework 2.8 or greater.

For Windows .NET development:

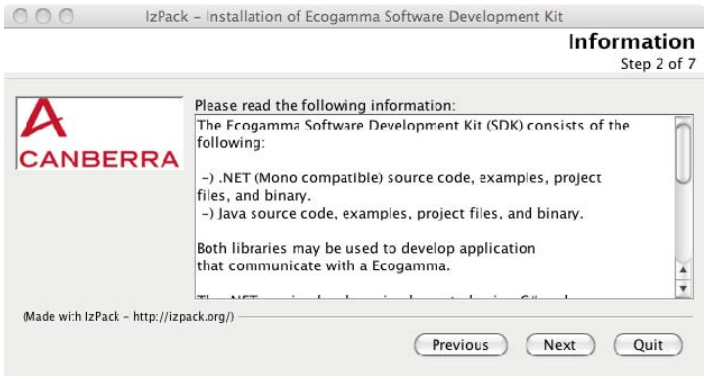
- Visual Studio 2008 or greater.
- .NET 3.5 or greater

Installation

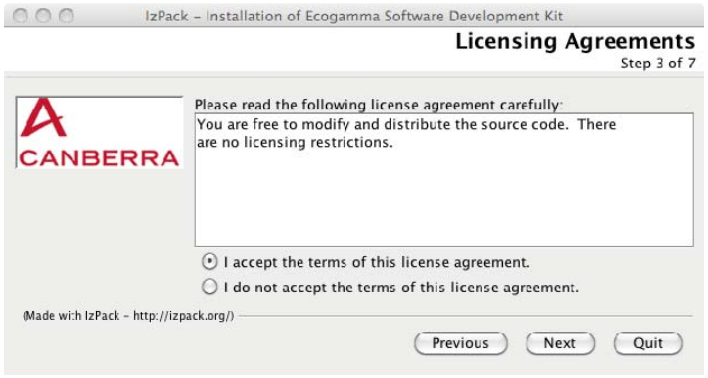
Installation is trivial. Run the installer file, install.jar. Double click on the install.jar file while using Windows, Ubuntu Linux and Mac OS X computers systems. The following dialog will appear.



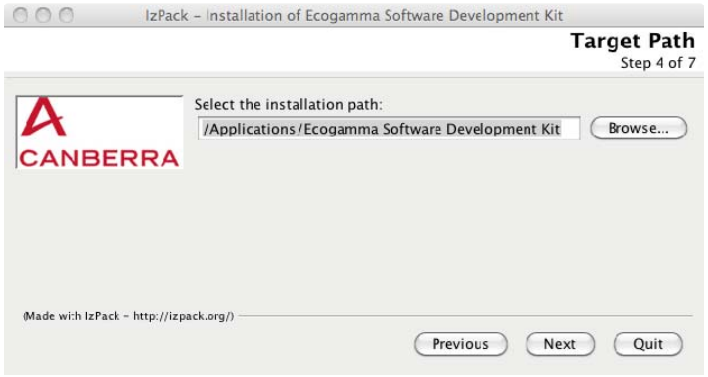
Press the “Next” button.



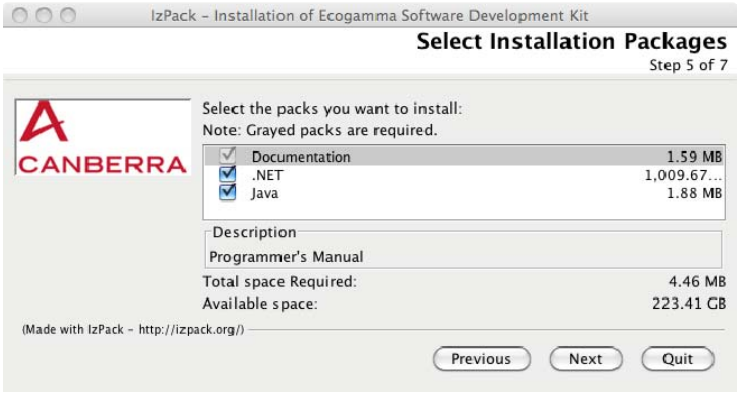
This dialog displays a description of the contents of the installer. Press “Next” button.



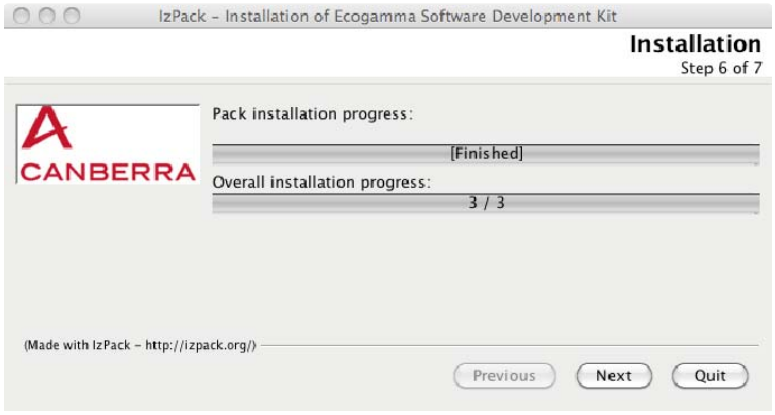
This screen displays the licensing agreement. Choose “I accept the terms of this license agreement” and press “Next.”



Select the location to install the software and press “Next.”



Choose the features you would like to install and press “Next.”



The software is now installed.

Directory Structure

The directory structure for the SDK is as follows:

- sdk/java. Contains the Java specific files
- sdk/c#. Contains the .NET specific files

Under each of these subdirectories there are similar directories that are as follows:

- lib. Contains the programming library
- src. Contains the source code and project files
- examples. Contains examples

Source Code

This SDK contains the source code. This allows you to learn how the library is designed and implemented. Also provides access to resolving any unwanted software features (i.e., bugs). Furthermore, the source code can be used to easily port the code to other object oriented languages like ActionScript, C++, Ruby, Python, and more.

The SDK contains the development projects along with the source code. The Java library was developed using the Eclipse IDE. The .NET library was developed using MonoDevelop, but the project file is compatible with Visual Studio 2008.

The code is extremely well documented; the purpose is to provide as much information as possible so using this source code will be easy.

Before every class or interface definition there exists a comment block that describes the purpose for the class. Also before every method definition there exists a comment block that describes the purpose of the method, method arguments, return value, and exceptions. These comments are displayed with intellisense while coding. Below is an example.

```
/**
   * This method will verify that the regions limits meet the
   * following constraints:
   *  -) left > 0 and right > 0
   *  -) left < right
   * @invariants. left > 0 and right > 0 and left < right
   * @param left.  The left channel
   * @param right.  The right channel
   * @exception.  canberra.Exceptions.IllegalArgumentException
   * error information
   */
  protected void checkRegion(int left, int right) throws Exception
```

2. Getting Started

Development

Overview

One class, GP110i, is used to perform all communications with the device. The following sections will discuss this interface and its associated methods and properties in more detail.

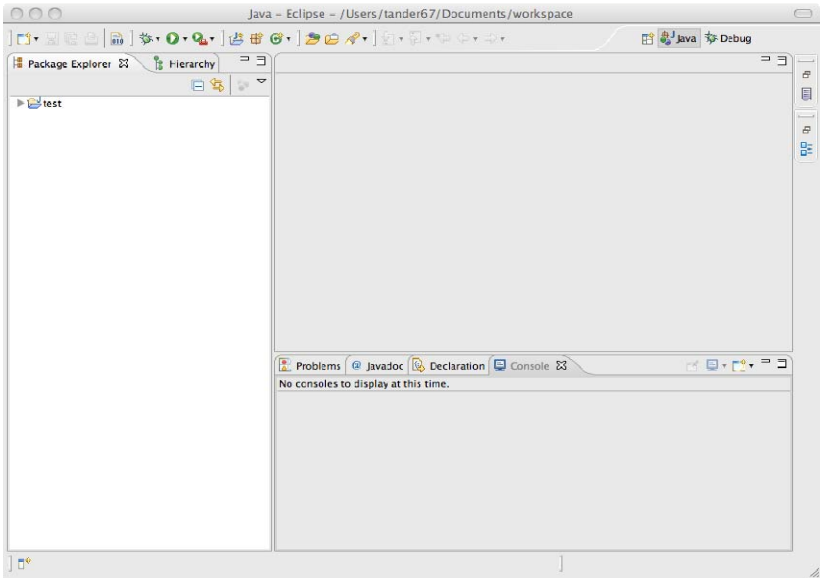
Using the library

Java

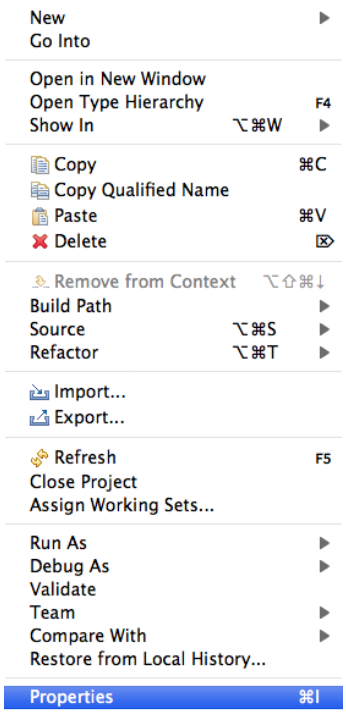
The package for this archive is gp110i-2.jar.

You can import this archive into your Eclipse environment by following the steps below:

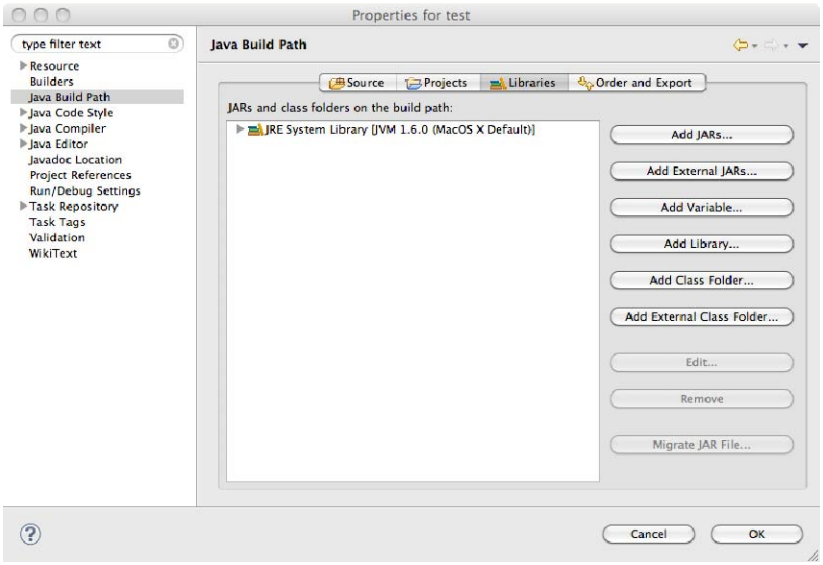
- Create a Java project.



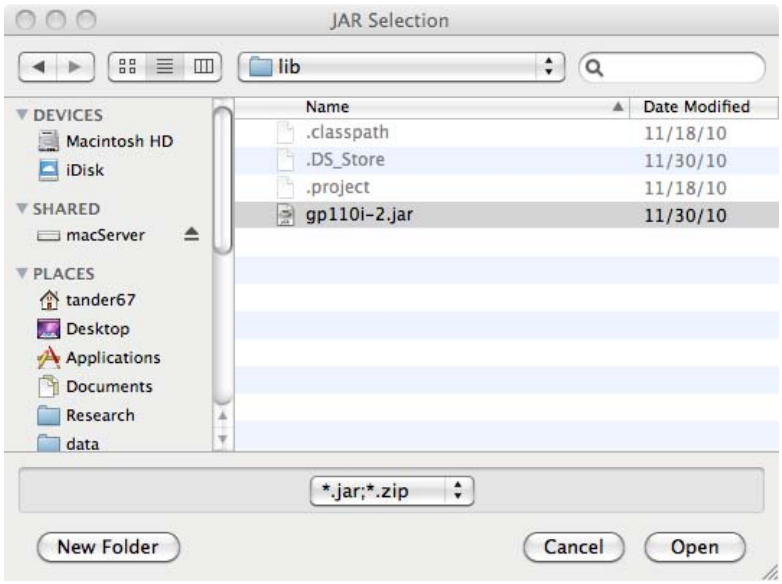
- Right mouse click over your Java project in the “Package Explorer” tab and choose “Properties.”



- Highlight “Java Build Path”



- Select the “Libraries” tab
- Choose “Add External JARS”



- Browse to the location of “gp110i-2.jar” and select it
- Now you are ready to writing your program.

.NET

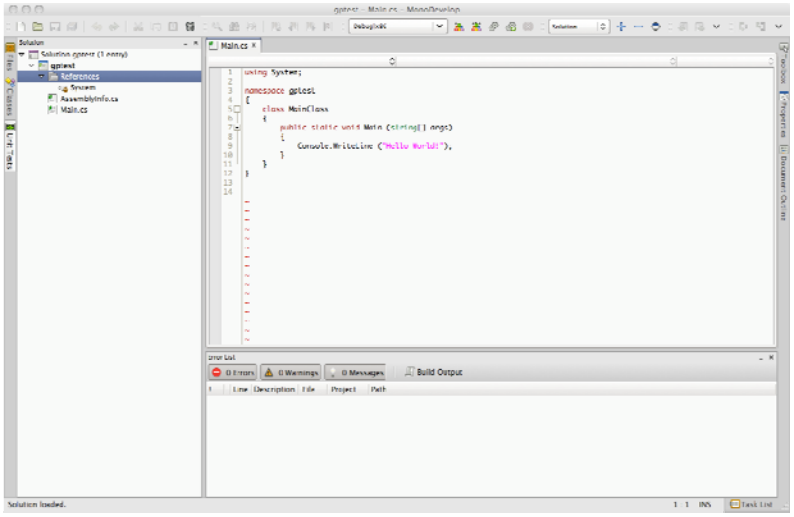
The assembly to use is gp110i-2.dll.

MonoDevelop

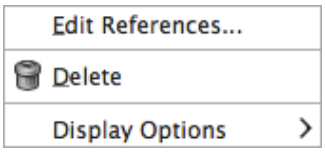
This section will discuss how to setup the MonoDevelop environment.

You can import this archive into your MonoDevelop environment by following the steps below:

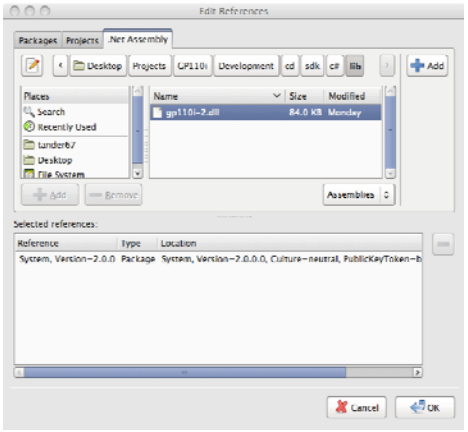
- Create a C#, Visual Basic, or F# solution



- Right mouse click over your References project in the “Solution Explorer” tab and choose “Edit References.”



- Select the “.Net Assembly” tab



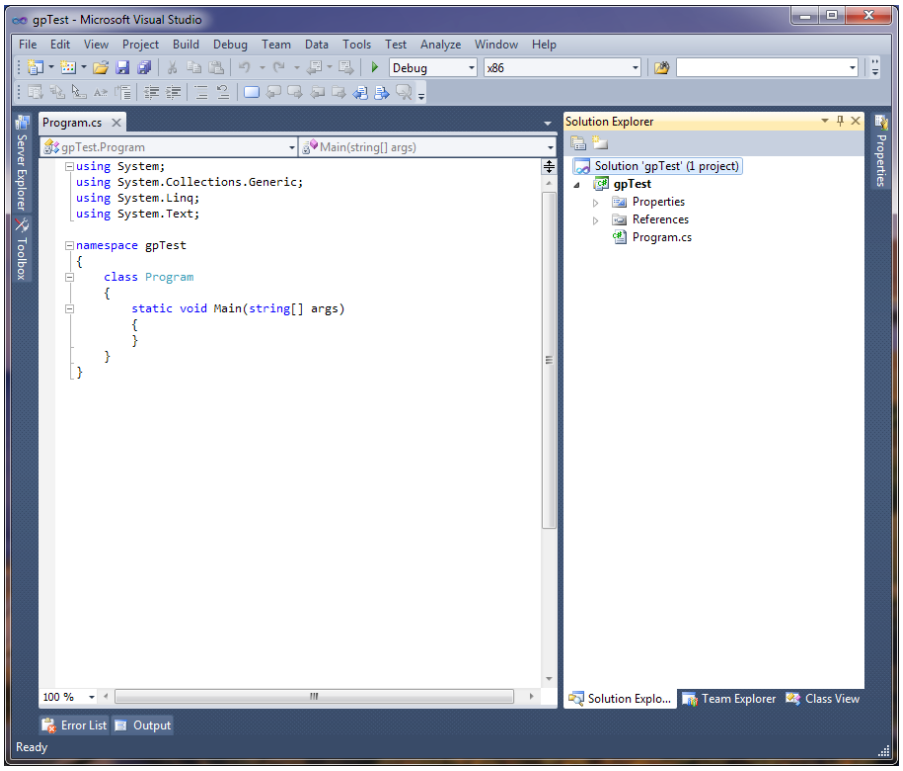
- Navigate to the gp110i-2.dll file and select it.
- Now you are ready to write your program.

Visual Studio

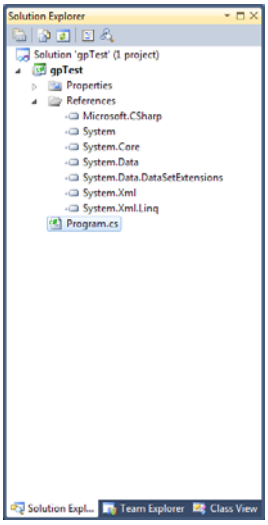
This section will discuss how to setup the Visual Studio 2010 environment.

You can import this archive into your MonoDevelop environment by following the steps below:

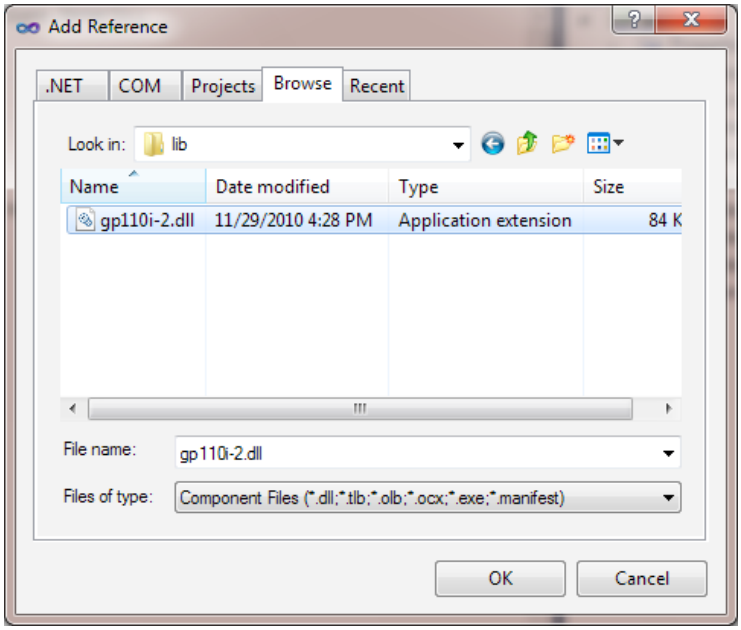
- Create a C#, Visual Basic, or F# solution



- Expand the “References” folder in the “Solution Explorer”



- Right click the mouse over the “References” folder and select “Add Reference”



- Select the “Browse” tab and navigate to the gp110i-2.dll assembly.
- Press “OK”
- Now you are ready to write your program.

Entry Point

The main entry point for this Software Development Kit is the GP110i class.

Namespace

The namespace that contains the Software Development Kit is tabulated below.

Namespace	Language
com.canberra.communications	Java
Canberra.Communications	C#

Thread Safety

All methods of the GP110i class are thread-safe. This means that you can share the same instance between multiple threads without synchronization problems.

Methods

This section will discuss the methods associated with the GP110i class.

Discovery

This method is used to discover other EcoGamma devices on your network. The return value will update periodically as new devices are discovered. Therefore, it is best to periodically invoke this method.

Java Format:

```
public List<String> discoveredDevices();
```

.NET Format:

```
public List<String> DiscoveredDevices();
```

Arguments:

None.

Java Exceptions:

IOException	Errors during connection process.

.NET Exceptions:

SocketException	Errors during connection process.

Return:

List<String>	A list of network addresses; one for each device that has been discovered.

Open

This method is used to establish a connection between your application and the device.

Java Format:

public void open(String Device);

.NET Format:

public void Open(String Device);

Arguments:

String Device	The network address of the device.

Java Exceptions:

IOException	Errors during connection process.

.NET Exceptions:

SocketException	Errors during connection process.

Common Exceptions:

DatasourceAlreadyOpenException	Indicates that this instance is already connected to a device.

Open State

This method is used to determine whether the class instance is already connected to a device.

Java Format:

public boolean isOpen();

.NET Format:

public bool IsOpen;

Return:

State indicating whether a connection exists

Close

This method is used to terminate a connection.

Java Format:

public void close();

.NET Format:

public void Close();

Java Exceptions:

IOException	Errors during connection process.

.NET Exceptions:

SocketException	Errors during connection process.

Common Exceptions:

DatasourceNotOpenException	This error is thrown if this instance is not connected to a device.

Enumerating Parameters

This method is used to enumerate all of the parameters the device supports. The information that is returned will contain parameter metadata. The metadata consists of information like name, description, minimum, maximum, and default value. All parameters are defined in the namespace canberra.protocols.gp110i.datatypes.parameters.

The first method will list all parameters. The second method will list read-only parameters. The last method returns a list of read-write parameters. See the "Parameters" on page 49 for further information.

Java Format:

```
public List<IParameterMetaData> listAllParameters();

public List<IParameterMetaData> listReadOnlyParameters();

public List<IParameterMetaData> listReadWriteParameters();
```

.NET Format:

```
public List<IParameterMetaData> ListAllParameters();

public List<IParameterMetaData> ListReadOnlyParameters();

public List<IParameterMetaData> ListReadWriteParameters();
```

Java Exceptions:

IOException	Errors during connection process.

.NET Exceptions:

SocketException	Errors during connection process.

Common Exceptions:

DatasourceNotOpenException	This error is thrown if this instance is not connected to a device.
DeviceErrorException	The device returned an error.
InvalidResponseException	An incorrect response was returned.
ChecksumException	The checksum received does not match the computed checksum.
IllegalArgumentException	An incorrect argument was supplied to a method

Return:

A list of parameters.

Getting Parameters

This method is used to get parameter values from the device. See the “Parameters” on page 49 for further information.

Java Format:

```
public List<IParameterMetaData> getParameters(List<IParameterMetaData>
                                                pars);
```

.NET Format:

```
public List<IParameterMetaData> GetParameters(List<IParameterMetaData>
                                                pars);
```

Arguments:

List<IParameterMetaData> pars	The parameters to get

Java Exceptions:

IOException	Errors during connection process.

.NET Exceptions:

SocketException	Errors during connection process.

Common Exceptions:

DatasourceNotOpenException	This error is thrown if this instance is not connected to a device.
DeviceErrorException	The device returned an error.
InvalidResponseException	An incorrect response was returned.
ChecksumException	The checksum received does not match the computed checksum.

Return:

A list of parameters requested from the device.

Setting Parameters

This method is used to get parameter values from the device. See the “Parameters” on page 49 for further information.

Java Format:

public void setParameters(List<IParameterMetaData> pars);

.NET Format:

public void SetParameters(List<IParameterMetaData> pars);

Arguments:

List<IParameterMetaData> pars	The parameters to write

Java Exceptions:

IOException	Errors during connection process.

.NET Exceptions:

SocketException	Errors during connection process.

Common Exceptions:

DatasourceNotOpenException	This error is thrown if this instance is not connected to a device.
DeviceErrorException	The device returned an error.
InvalidResponseException	An incorrect response was returned.
ChecksumException	The checksum received does not match the computed checksum.
IllegalArgumentException	An incorrect argument was supplied to a method

Getting Acquisition Data

This method is used to get acquisition values like total dose, dose rate, and temperature.

Java Format:

public List<IParameterMetaData> getAcquiredData();

.NET Format:

public List<IParameterMetaData> GetAcquiredData();

Java Exceptions:

IOException	Errors during connection process.

.NET Exceptions:

SocketException	Errors during connection process.

Common Exceptions:

DatasourceNotOpenException	This error is thrown if this instance is not connected to a device.
DeviceErrorException	The device returned an error.
InvalidResponseException	An incorrect response was returned.
ChecksumException	The checksum received does not match the computed checksum.
IllegalArgumentException	An incorrect argument was supplied to a method

Return:

The acquisition parameters.

Getting Diagnostic Information

This method is used to get diagnostic information like GM tube voltage and LVPS voltage.

Java Format:

```
public List<IParameterMetaData> getDiagnosticInformation();
```

.NET Format:

```
public List<IParameterMetaData> GetDiagnosticInformation();
```

Java Exceptions:

IOException	Errors during connection process.

.NET Exceptions:

SocketException	Errors during connection process.

Common Exceptions:

DatasourceNotOpenException	This error is thrown if this instance is not connected to a device.
DeviceErrorException	The device returned an error.
InvalidResponseException	An incorrect response was returned.
ChecksumException	The checksum received does not match the computed checksum.
IllegalArgumentException	An incorrect argument was supplied to a method

Return:

The diagnostic information.

Getting Push List

The device supports the ability for pushing data to clients at periodic intervals. The data that may be pushed is specified in the following table:

Value	Description	Notes
01	Full Dose	Returns the full dose in SSN-2 format with dose units.
19	Temperature	Returns the probe temperature in degrees C in signed fixed point format as follows: <s><vv>.<n> Where the sign <s> is always present, the value <vv> can be 1 or 2 digits depending on the magnitude, and the fractional value <n> is always present
40	Filtered Dose Rate	Returns the filtered dose rate in SSN-2 format with dose rate units.
41	Unfiltered Dose Rate	Returns the unfiltered dose rate in SSN-2 format with dose rate units.
50	Time	Returns the device time in HH:MM:SS format
49	Alarms	Returns the alarm indication for Rate and Total as follows: A:** no alarm A:R* Rate High or Rate Alert alarm asserted A:*D Total High or Total Alert alarm asserted A:RD Both Rate High or Rate Alert alarm asserted and Rate high or Rate Alert alarm asserted
53	Append Checksum	Appends the checksum of the entire push list string to the end of the string. The checksum format is as follows: XX<term> Where XX is the 8-bit XOR of all the characters in the returned string excluding the checksum and terminator characters, represented in hex by two hex/ASCII digits. The <term> character is the string terminator character '}'
54	Date & Time	Returns the device date and time in DD/MM/YY HH:MM:SS format

The Value column indicates the value to specify in the CustomPushList parameter list. The Description column describes the information associated with that custom push list value. The Notes describe the format of the return value.

The Push List Code Table contains the Push codes that can be sent to the device through the CustomPushList parameter. Once defined, the Custom Push List must be activated through the PushListSelect parameter. Lastly, you need to make sure streaming is enabled, see parameter StreamingEnabled.

Up to 16 individual codes can be specified in a custom push list. Each code must be specified as two ASCII numeric digits in decimal notation with no separating characters. Push list codes can range from “00” to “99”

Dose rates and Dose values are returned with appropriate units based on the DoseUnits parameter as follows:

- With US units selected, the returned units for Dose values are in rems “R”, millirems “mR”, or microrems “uR”

- With SI units selected, the returned units for Dose values are in sieverts “Sv”, millisieverts “mSv”, or microsieverts “uSv”

Rates are returned in dose units per hour “/h”

Typical string returned by the push list for dose rates and dose values are as follows:

7.39E+1uR/h 7.51E+1uR/h 5.06E+2uR

7.29E+1uR/h 7.50E+1uR/h 5.06E+2uR

7.39E+1uR/h 7.49E+1uR/h 5.06E+2uR

After setting up the CustomPushList, PushListSelect , and StreamingEnabled parameters, you can access the pushed data by iteratively invoking the method below.

Java Format:

public String getPushedList();

.NET Format:

public String GetPushedList();

Java Exceptions:

IOException	Errors during connection process.

.NET Exceptions:

SocketException	Errors during connection process.

Common Exceptions:

DatasourceNotOpenException	This error is thrown if this instance is not connected to a device.
DeviceErrorException	The device returned an error.
InvalidResponseException	An incorrect response was returned.
ChecksumException	The checksum received does not match the computed checksum.
IllegalArgumentException	An incorrect argument was supplied to a method

Return:

The pushed list data, see above for the format.

Getting History Summary

This method is used to get a summary of the historical data stored on the device. See the “Logging Details” on page 61 for further information.

Java Format:

```
public List<Summary> getLogSummary();
```

.NET Format:

```
public List<Summary> GetLogSummary();
```

Java Exceptions:

IOException	Errors during connection process.

.NET Exceptions:

SocketException	Errors during connection process.

Common Exceptions:

DatasourceNotOpenException	This error is thrown if this instance is not connected to a device.
DeviceErrorException	The device returned an error.
InvalidResponseException	An incorrect response was returned.
ChecksumException	The checksum received does not match the computed checksum.
IllegalArgumentException	An incorrect argument was supplied to a method

Return:

A list of log summaries. Each summary contains information like file/element number, start time and end time of the data stored in the element/file, and more. This information is used to retrieve a historical data stored on the device.

Getting History Data

This method is used to get historical data stored on the device. See the “Logging Details” on page 61 for further information.

Java Format:

public List<Data> getLogData(int element, Date start, Date end);

.NET Format:

public List<Data> GetLogData(int element, Date start, Date end);

Arguments:

int element	The element or file number.
Date start	The start time of the data to retrieve.
Date end	The end time of the data to retrieve.

Java Exceptions:

IOException	Errors during connection process.

.NET Exceptions:

SocketException	Errors during connection process.

Common Exceptions:

DatasourceNotOpenException	This error is thrown if this instance is not connected to a device.
DeviceErrorException	The device returned an error.
InvalidResponseException	An incorrect response was returned.
ChecksumException	The checksum received does not match the computed checksum.
IllegalArgumentException	An incorrect argument was supplied to a method

Return:

The data associated with this element/file and time span.

Modes

These methods are used to enable/disable different modes of operation. You can determine whether a specific mode is enabled by requesting for the DeviceStatus parameter.

The first method enables acquisition. The second method enables calibration. The last method enables logging data.

Java Format:

```
public void enableCalibration(boolean state, boolean forLowRange);

public void enableLinearityCheck(boolean state, int index, int selRange);

public void enableLogging(boolean state);
```

.NET Format:

```
public void EnableCalibration(bool state, bool forLowRange);

public void EnableLinearityCheck(bool state, uint index, uint selRange);

public void EnableLogging(bool state);
```

Arguments:

boolean state	The enable state.
int index	The array index (option base 0) of the linearity check point. See next section for accessing checkpoints.
boolean forLowRange	State that indicates whether to enable calibration for low or high range tube. A value of 'true' indicates enable for low range.
int selRange	The selected range to use. Values are as follows: 0 = Automatic. The device automatically determines which range to use 1 = Low. The device will use the low range tube. 2=High. The device will used the high range tube.

Java Exceptions:

IOException	Errors during connection process.

.NET Exceptions:

SocketException	Errors during connection process.

Common Exceptions:

DatasourceNotOpenException	This error is thrown if this instance is not connected to a device.
DeviceErrorException	The device returned an error.
InvalidResponseException	An incorrect response was returned.
ChecksumException	The checksum received does not match the computed checksum.
IllegalArgumentException	An incorrect argument was supplied to a method

Linearity Test

These methods are used to get and set the linearity checkpoints. The values that are specified must be between the values specified by the parameters LowGMLimit and HighGMLimit.

Java Format:

```
public float[] getLinearityCheckPoints();

public void setLinearityCheckPoints(float[] pts);
```

.NET Format:

```
public Single[] GetLinearityCheckPoints();

public void SetLinearityCheckPoints(Single[] pts);
```

Arguments:

float[] pts	The linearity check points. You must specify 12 points otherwise an exception is thrown, IllegalArgumentException.

Java Exceptions:

IOException	Errors during connection process.

.NET Exceptions:

SocketException	Errors during connection process.

Common Exceptions:

DatasourceNotOpenException	This error is thrown if this instance is not connected to a device.
DeviceErrorException	The device returned an error.
InvalidResponseException	An incorrect response was returned.
ChecksumException	The checksum received does not match the computed checksum.
IllegalArgumentException	An incorrect argument was supplied to a method.

Clear

This method is used to clear different device conditions.

Java Format:

public void clear(int option);

.NET Format:

public void Clear(ClearCommands option);

Arguments:

int option	The clear option. The clear options are as follows: AlarmCondition. Will clear the alarm or warning DiagnosticCounters. Will clear the diagnostic counters LogData. Will clear the log data ResetFactoryDefaults. Will reset to factory defaults. TotalDose. Will clear the total dose. ResetCalibration. Will reset the calibration due date. ResetLinearityTest. Will reset the linearity test information. See the ClearCommands interface for these definitions.

Java Exceptions:

IOException	Errors during connection process.

.NET Exceptions:

SocketException	Errors during connection process.

Common Exceptions:

DatasourceNotOpenException	This error is thrown if this instance is not connected to a device.
DeviceErrorException	The device returned an error.
InvalidResponseException	An incorrect response was returned.
ChecksumException	The checksum received does not match the computed checksum.
IllegalArgumentException	An incorrect argument was supplied to a method.

Examples

This section will list several examples. All examples assume that the USB connection and the associated factory default address are being used.

Discovering devices

This example will show you how to discover EcoGamma devices on your network.

Java

```
package com.canberra.examples;

import java.util.List;
import com.canberra.protocols.gp110i.communications.GP110i;


public class Main {


 private Main() {};


 public static void main(String[] args) {

     //Create a device instance
     GP110i dev = new GP110i();
     try {
            List<String> list=null;
            int cnt=0;
            do {

                //Query for the latest discovered devices
                list = dev.discoveredDevices();

                //Indicate whether any were found
                if (list.size() > 0) System.out.println("Found a device.");
                else System.out.println("No devices found.");

                //Pause for 1sec
                java.lang.Thread.sleep(1000);

                cnt++;
                //Loop until find a device
            } while((list.size()==0) || (cnt < 30));

            //Display the list of discovered devices
            for(String add : list) {
                System.out.println("Discovered device: " + add);
            }
     }catch(Exception ex) {
        System.out.println("Exception: " + ex);
     }
     finally {
        //Close any open connections
        try {
           if (dev.isOpen()) dev.close();
        }
        catch(Exception ex) {System.out.println("Exception: " + ex);}
     }
     System.exit(0);


 }}
```

.NET

```
using System;
using System.Collections.Generic;

using Canberra.Protocols.GP110i.Communications;
```

```
namespace gp110i2
{
  class MainClass
  {
      public static void Main (string[] args)
      {
          GP110i dev = new GP110i();
          try {
             List<String> list=null;
             int cnt=0;
             do {

                  //Query for the latest discovered devices
                  list = dev.DiscoveredDevices();

                  //Indicate whether any were found
                  if (list.Count > 0) Console.WriteLine("Found a device.");
                  else Console.WriteLine("No devices found.");

                  //Pause for 1sec
                  System.Threading.Thread.Sleep(1000);

                  cnt++;

              //Loop until find a device
              } while((list.Count==0) || (cnt < 30));

              //Display the list of discovered devices
              foreach(String add in list) {
                  Console.WriteLine("Discovered device: " + add);
              }
          }
          catch(Exception ex) {
              Console.WriteLine("Error: " + ex);
          }
          finally {
              try {
                 if (dev.IsOpen) dev.Close();
              }
              catch { }
          }

          Console.WriteLine("Done...");
      }
  }
}
```

Getting device parameters

This example will show you how to connect and get parameters from the device.

Java

```
package com.canberra.examples;


import com.canberra.datasources.parameters.IParameterMetaDataBase;
import com.canberra.protocols.gp110i.communications.GP110i;
import com.canberra.protocols.gp110i.datatypes.parameters.Parameter;


public class Main {


 private Main() {};

 @SuppressWarnings("rawtypes")
 public static void main(String[] args) {
     //Create a device instance
     GP110i dev = new GP110i();

     try {

        //Open a connection to the device
        dev.open("10.0.1.4");

        //Loop through all device parameters
        for (IParameterMetaDataBase par :
             dev.getParameters(dev.listAllParameters())) {
             Parameter p = (Parameter) par;
             System.out.println("Name: " + p.getName() + "; value: " +
                                p.getValue());
         }
     }
     catch(Exception ex) {
        System.out.println("Exception: " + ex);
     }
     finally {
        //Close any open connections
        try {
           if (dev.isOpen()) dev.close();
        }
        catch(Exception ex) {System.out.println("Exception: " + ex);}
     }
     System.exit(0);


 }}
```

.NET

```
using System;
using System.Collections.Generic;

using Canberra.Datasources.Parameters;
using Canberra.Protocols.GP110i.Communications;
using Canberra.Protocols.GP110i.Datatypes.Parameters;

namespace gp110i2
{
  class MainClass
  {
      public static void Main (string[] args)
      {
          GP110i dev = new GP110i();
          try {
              dev.Open("10.0.1.4");

              List<IParameterMetaDataBase> pars = ParameterFactory.GetParameters();
              pars = dev.GetParameters(pars);

              foreach(IParameterMetaDataBase p in pars) {
                  Console.WriteLine(p.ToString());
              }
          }
          catch(Exception ex) {
              Console.WriteLine("Error: " + ex);
          }
          finally {
              try {
                 if (dev.IsOpen) dev.Close();
              }
              catch { }
          }

          Console.WriteLine("Done...");
      }
  }
}
```

Setting device parameters

This example will show you how to connect and set parameters from the device.

Java

```
package com.canberra.examples;

import java.util.ArrayList;

import com.canberra.datasources.parameters.IParameterMetaDataBase;
import com.canberra.protocols.gp110i.communications.GP110i;
import com.canberra.protocols.gp110i.datatypes.parameters.Parameter;


public class Main {


 private Main() {};

 @SuppressWarnings("rawtypes")
 public static void main(String[] args) {
     //Create a device instance
     GP110i dev = new GP110i();

     try {
        ArrayList<IParameterMetaDataBase> wpars = new
        ArrayList<IParameterMetaDataBase>();

        //Open a connection to the device
        dev.open("10.0.1.4");

        //Loop through all device parameters
        for (IParameterMetaDataBase par :
             dev.getParameters(dev.listAllParameters())) {
             Parameter p = (Parameter) par;
             System.out.println("Name: " + p.getName() + "; value: " +
                                p.getValue());
             if (dev.isReadOnlyParameter(par)) continue;
             wpars.add(par);
         }
         //Write the values back to the device
         dev.setParameters(wpars);

     }
     catch(Exception ex) {
        System.out.println("Exception: " + ex);
     }
     finally {
        //Close any open connections
        try {
           if (dev.isOpen()) dev.close();
        }
        catch(Exception ex) {System.out.println("Exception: " + ex);}
     }
     System.exit(0);


 }}
```

.NET

```
using System;
using System.Collections.Generic;

using Canberra.Datasources.Parameters;
using Canberra.Protocols.GP110i.Communications;
using Canberra.Protocols.GP110i.Datatypes.Parameters;

namespace gp110i2
{
  class MainClass
  {
      public static void Main (string[] args)
      {
          GP110i dev = new GP110i();
          try {
              dev.Open("10.0.1.4");

              List<IParameterMetaDataBase> pars = ParameterFactory.GetParameters();
              List<IParameterMetaDataBase> wpars =new List<IParameterMetaDataBase>();
              pars = dev.GetParameters(pars);

              foreach(IParameterMetaDataBase p in pars) {
                  Console.WriteLine(p.ToString());
                  if (GP110i.IsReadOnlyParameter(p)) continue;
                  wpars.Add(p);

              }
              //Write the values back to the device
              dev.SetParameters(wpars);
          }
          catch(Exception ex) {
              Console.WriteLine("Error: " + ex);
          }
          finally {
              try {
                 if (dev.IsOpen) dev.Close();
              }
              catch { }
          }

          Console.WriteLine("Done...");
      }
  }
}
```

Getting historical data

This example will show you how to connect and get historical data from the device.

Java

```
package com.canberra.examples;
import com.canberra.states.IState;
import com.canberra.protocols.gp110i.communications.GP110i;
import com.canberra.protocols.gp110i.datatypes.historian.Data;
import com.canberra.protocols.gp110i.datatypes.historian.Summary;


public class Main {

 private Main() {};


 public static void main(String[] args) {
     //Create a device instance
     GP110i dev = new GP110i();

     try {
        //Open a connection to the device
        dev.open("10.0.1.4");
        //Enable logging
        dev.enableLogging(true);


        //Read the history summary
        for (Summary sum : dev.getLogSummary()) {
             System.out.println("Element: " + sum.getElement() + "; numEntries: " +
             sum.getNumberOfEntries() + "; startDate: " + sum.getStartDate()+ ";
             endDate: " + sum.getEndDate());

             //Get the history data
             if (sum.getNumberOfEntries() > 0) {
                 for (Data data : dev.getLogData(sum.getElement(),
                      sum.getStartDate(), sum.getEndDate())) {
                 String alarms="";
                 for (IState state : data.getAlarms()) {
                      alarms += state.getName() + ", ";
                 }
                 String status="";
                 for (IState state : data.getStatus()) {
                      status += state.getName() + ", ";
                 }
                 System.out.println("Dose rate: " + data.getDoseRate() + "; total
                 dose: " + data.getTotalDose() + "; temperature: " +
                 data.getTemperature() + "; Date: " + data.getTime() + "; Alarms: "
                 + alarms + "; Status: " + status);
                 }
             }
         }
     }
     catch(Exception ex) {
        System.out.println("Exception: " + ex);
     }
     finally {
        //Close any open connections
        try {
           if (dev.isOpen()) dev.close();
        }
        catch(Exception ex) {System.out.println("Exception: " + ex);}
     }
     System.exit(0);}}
```

.NET

```
using System;
using System.Collections.Generic;

using Canberra.Datasources.Parameters;
using Canberra.Protocols.GP110i.Communications;
using Canberra.Protocols.GP110i.Datatypes.Parameters;
using Canberra.Protocols.GP110i.Datatypes.Historian;

using System.Globalization;
using Canberra.States;

namespace gp110i2
{
    class MainClass
    {
        public static void Main (string[] args)
        {
            GP110i dev = new GP110i();
            try {

                dev.Open("10.0.1.4");

                //Enable logging
                dev.EnableLogging(true);


                //Read the history summary
                foreach (Summary sum in dev.GetLogSummary()) {
                     Console.WriteLine("Element: " + sum.Element + "; numEntries: "
                        + sum.NumberOfEntries + "; startDate: " + sum.StartDateTime+
                        "; endDate: " + sum.EndDateTime);

                    //Get the history data
                    if (sum.NumberOfEntries > 0) {
                        foreach (Data data in dev.GetLogData(sum.Element,
                            sum.StartDateTime, sum.EndDateTime)) {

                        String alarms="";
                        foreach (IState state in data.Alarms) {
                            alarms += state.Name + ", ";
                        }
                        String status="";
                        foreach (IState state in data.Status) {
                            status += state.Name + ", ";
                        }
                        Console.WriteLine("Dose rate: " + data.DoseRate + "; total
                            dose: " + data.TotalDose + "; temperature: " +
                            data.Temperature + "; Date: " + data.Time + "; Alarms: "
                            + alarms + "; Status: " + status);
                        }

                    }
                }
            }
            catch(Exception ex) {
                Console.WriteLine("Error: " + ex);
            }
            finally {
                try {
                    if (dev.IsOpen) dev.Close();
                }
                catch { }
            }

            Console.WriteLine("Done...");
        }
    }
}
```

Calibration

This example will show you how to perform a calibration. This example requires two radioactive sources. One source is, 0.1 R/h, is used to calibrate the low range tube. The other, 140 R/h, is used to calibrate the high range tube.

Java

```
package com.canberra.examples;

import java.util.ArrayList;
import java.util.List;

import com.canberra.datasources.parameters.IParameterMetaDataBase;
import com.canberra.states.IState;
import com.canberra.protocols.gp110i.communications.GP110i;
import com.canberra.protocols.gp110i.datatypes.parameters.acquisition.DoseUnits;
import com.canberra.protocols.gp110i.datatypes.parameters.calibration.*;
import com.canberra.protocols.gp110i.datatypes.parameters.calibration.states.*;


public class Main {


 private Main() {};

 public static void main(String[] args) {
     //Create a device instance
     GP110i dev = new GP110i();

     try {
        List<IParameterMetaDataBase> wpars = new
        ArrayList<IParameterMetaDataBase>();
        List<IParameterMetaDataBase> par= new ArrayList<IParameterMetaDataBase>();

        //Open a connection to the device
        dev.open("10.0.1.4");

        //Disable calibrations (for both tubes)
        dev.enableCalibration(false, true);
        dev.enableCalibration(false, false);

        //Disable linearity testing
        for (int i=0; i<12; i++) {
           try {
               dev.enableLinearityCheck(false, i, 0);
           }
           catch(Exception ex) {}
        }

        //Disable logging
        dev.enableLogging(false);

        //Do the low range tube (units are in R/h so specify units)
        wpars.add(new DoseUnits(DoseUnits.REM));
        wpars.add(new LowRangeCalibrationSource(.1));
        dev.setParameters(wpars);

        //Enable calibration for low range tube
        dev.enableCalibration(true, true);

        //Loop until calibration is done
        boolean loop=false, success=false;
        par.add(new CalibrationStatus());
        par.add(new LowRangeCalibrationFactor());
        par.add(new HighRangeCalibrationFactor());

        do {
           java.lang.Thread.sleep(2000);
           loop = false;
           par = dev.getParameters(par);
           for (IState state : ((CalibrationStatus) par.get(0)).getState()) {
              if (state instanceof Busy) loop=true;
              if (state instanceof Success) success=true;
           }
           if (loop)
             System.out.println("Busy calibrating.  Time Remaining: " +
             ((CalibrationStatus) par.get(0)).getTimeRemaining() + " (sec)");
        } while(loop);
```

```
            if (!success) System.out.println("Calibration did not succeed");
            else System.out.println("Calibration factor is: " + par.get(1));

            //Do the high range tube
            wpars.clear();
            wpars.add(new HighRangeCalibrationSource(140));
            dev.setParameters(wpars);

            //Enable calibration for high range tube
            dev.enableCalibration(true, false);

            //Loop until calibration is done
            do {
               java.lang.Thread.sleep(2000);
               loop = false;
               par = dev.getParameters(par);
               for (IState state : ((CalibrationStatus) par.get(0)).getState()) {
                  if (state instanceof Busy) loop=true;
                  if (state instanceof Success) success=true;
               }
               if (loop)
                 System.out.println("Busy calibrating.  Time Remaining: " +
                 ((CalibrationStatus) par.get(0)).getTimeRemaining() + " (sec)");

            } while(loop);

            if (!success) System.out.println("Calibration did not succeed");
            else System.out.println("Calibration factor is: " + par.get(2));

         }
         catch(Exception ex) {
            System.out.println("Exception: " + ex);
         }
         finally {
            //Disable calibrations (for both tubes)
            try {
               if (dev.isOpen()) dev.enableCalibration(false, true);
            }
            catch(Exception ex) {}
            try {
               if (dev.isOpen()) dev.enableCalibration(false, false);
            }
            catch(Exception ex) {}

            //Enable logging
            try {
               if (dev.isOpen()) dev.enableLogging(true);
            }
            catch(Exception ex) {}

            //Close any open connections
            try {
               if (dev.isOpen()) dev.close();
            }
            catch(Exception ex) {System.out.println("Exception: " + ex);}
         }
         System.exit(0);


      }}
```

.NET

```
using System;
using System.Collections.Generic;

using Canberra.Datasources.Parameters;
using Canberra.States;
using Canberra.Protocols.GP110i.Communications;
using Canberra.Protocols.GP110i.Datatypes.Parameters;
using Canberra.Protocols.GP110i.Datatypes.Parameters.Acquisition;
using Canberra.Protocols.GP110i.Datatypes.Parameters.Calibration;
using Canberra.Protocols.GP110i.Datatypes.Parameters.Calibration.States;

namespace gp110i2
{
  class MainClass
  {
      public static void Main (string[] args)
      {
          GP110i dev = new GP110i();
          try {
              List<IParameterMetaDataBase> wpars = new
                                                   List<IParameterMetaDataBase>();
              List<IParameterMetaDataBase> par= new List<IParameterMetaDataBase>();

              //Open a connection to the device
              dev.Open("10.0.1.4");

              //Disable calibrations (for both tubes)
              dev.EnableCalibration(false, true);
              dev.EnableCalibration(false, false);

              //Disable linearity
              for (uint i=0; i<12; i++) {
                  try {dev.EnableLinearityCheck(false, i, 0);} catch{}
              }

              //Disable logging
              dev.EnableLogging(false);

              //Do the low range tube (units are in R/h so specify units)
              wpars.Add(new DoseUnits(DoseUnits.REM));
              wpars.Add(new LowRangeCalibrationSource(.1));
              dev.SetParameters(wpars);

              //Enable calibration for low range tube
              dev.EnableCalibration(true, true);

              //Loop until calibration is done
              bool loop=false, success=false;
              par.Add(new CalibrationStatus());
              par.Add(new LowRangeCalibrationFactor());
              par.Add(new HighRangeCalibrationFactor());

              do {
                  System.Threading.Thread.Sleep(2000);
                  loop = false;
                  par = dev.GetParameters(par);
                  foreach (IState state in (par[0] as CalibrationStatus).State) {
                      if (state is Busy) loop=true;
                      if (state is Success) success=true;
                  }
                  if (loop)
                     Console.WriteLine("Busy calibrating.  Time Remaining:" + (par[0]
                     as CalibrationStatus).TimeRemaining + " (sec)");
              } while(loop);

              if (!success) Console.WriteLine("Calibration did not succeed");
              else Console.WriteLine("Calibration factor is: " + par[1]);
```

```
                //Do the high range tube
                wpars.Clear();
                wpars.Add(new HighRangeCalibrationSource(140));
                dev.SetParameters(wpars);

                //Enable calibration for high range tube
                dev.EnableCalibration(true, false);

                //Loop until calibration is done
                do {
                    System.Threading.Thread.Sleep(2000);
                    loop = false;
                    par = dev.GetParameters(par);
                    foreach (IState state in  (par[0] as CalibrationStatus).State) {
                        if (state is Busy) loop=true;
                        if (state is Success) success=true;
                    }
                    if (loop)
                       Console.WriteLine("Busy calibrating.  Time Remaining:" + (par[0]
                       as CalibrationStatus).TimeRemaining + " (sec)");

                } while(loop);

                if (!success) Console.WriteLine("Calibration did not succeed");
                else Console.WriteLine("Calibration factor is: " + par[2]);
            }
            catch(Exception ex) {
                Console.WriteLine("Error: " + ex);
            }
            finally {
                //Disable calibrations (for both tubes)
                try { if (dev.IsOpen) dev.EnableCalibration(false, true);} catch { }
                try { if (dev.IsOpen) dev.EnableCalibration(false, false);} catch { }

                //Enable logging
                try { if (dev.IsOpen) dev.EnableLogging(true);}catch { }

                //Close any open connections
                try {
                   if (dev.IsOpen) dev.Close();
                }
                catch { }
            }

            Console.WriteLine("Done...");
        }
    }
}
```

Linearity Check

This example will show you how to perform a linearity check that is typically done after a calibration. This example requires 5 radioactive sources with emission rates of 0.000463, 16.4, 118, 160, and 325 R/h.

Java

```
package com.canberra.examples;

import java.util.ArrayList;
import java.util.List;

import com.canberra.states.IState;
import com.canberra.datasources.parameters.IParameterMetaDataBase;
import com.canberra.protocols.gp110i.communications.GP110i;
import com.canberra.protocols.gp110i.datatypes.parameters.calibration.*;
import com.canberra.protocols.gp110i.datatypes.parameters.calibration.states.*;

public class Main {


 private Main() {};

 public static void main(String[] args) {
     //Create a device instance
     GP110i dev = new GP110i();

     try {
        List<IParameterMetaDataBase> par= new ArrayList<IParameterMetaDataBase>();

        //Open a connection to the device
        dev.open("10.0.1.4");

        //Disable calibrations (for both tubes)
        dev.enableCalibration(false, true);
        dev.enableCalibration(false, false);

        //Disable linearity
        for (int i=0; i<12; i++) {
           try {
               dev.enableLinearityCheck(false, i, 0);
           }
           catch(Exception ex) {System.out.println("Exception: " + ex);}
        }


        //Disable logging
        dev.enableLogging(false);

        //Set the dose units to REM because the linearity points below
        //are specified in R/h
        List<IParameterMetaDataBase> units = new
                                          ArrayList<IParameterMetaDataBase>();
        units.add(new DoseUnits(DoseUnits.REM));
        dev.setParameters(units);

        //Set the linearity points
        float []pt=new float[]{0.000463F, 16.4F, 118, 160, 325};
        dev.setLinearityCheckPoints(pt);

        //Set the parameter we want to monitor into the list
        par.add(new CalibrationStatus());

        //Create a parameter list to query for the check points once
        //the test is complete
        LinearityCheckPoints pts;
        List<IParameterMetaDataBase> lin=new ArrayList<IParameterMetaDataBase>();
        lin.add(new LinearityCheckPoints());

        //Perform linearity test on each point
        for(int i=0; i<pt.length; i++) {
           dev.enableLinearityCheck(true, i, 0);

           //Loop until linearity is done
           boolean loop=false, success=false;
           do {
              java.lang.Thread.sleep(2000);
              loop = false;
```

```
            par = dev.getParameters(par);
            for (IState state : ((CalibrationStatus) par.get(0)).getState()) {
               if (state instanceof LinearityCheckBusy) loop=true;
               if (state instanceof LinearityCheckSuccess) success=true;
            }
            if (loop) System.out.println("Busy performing linearity check.  Time
                                          Remaining: "+((CalibrationStatus)
                                          par.get(0)).getTimeRemaining() + "
                                          (sec)");

         } while(loop);

         //Get the check points to display the information
         pts=((LinearityCheckPoints) dev.getParameters(lin).get(0));

         //Display status for this test
         if (!success)
            System.out.println("Linearity check did not succeed for: " + pt[i] +
                               ", Average rate: " +
                               pts.getPoints().get(i).getAverageRate() + " Percent
                               Error: " +
                               pts.getPoints().get(i).getPercentError());
         else
            System.out.println("Linearity check did succeed for: " + pt[i] + ",
                               Average rate: " +
                               pts.getPoints().get(i).getAverageRate() + " Percent
                               Error: " +
                               pts.getPoints().get(i).getPercentError());

       }
    }
    catch(Exception ex) {
       System.out.println("Exception: " + ex);
    }
    finally {
       //Disable linearity
       try {
          if (dev.isOpen())
             for (int i=0; i<12; i++) dev.enableLinearityCheck(false, i, 0);
       }
       catch() { }


       //Enable logging
       try {
          if (dev.isOpen()) dev.enableLogging(true);
       }
       catch() { }


       //Close any open connections
       try {
          if (dev.isOpen()) dev.close();
       }
       catch(Exception ex) {System.out.println("Exception: " + ex);}
    }
    System.exit(0);

}}
```

.NET

```
using System;
using System.Collections.Generic;

using Canberra.Datasources.Parameters;
using Canberra.States;
using Canberra.Protocols.GP110i.Communications;
using Canberra.Protocols.GP110i.Datatypes.Parameters;
using Canberra.Protocols.GP110i.Datatypes.Parameters.Calibration;
using Canberra.Protocols.GP110i.Datatypes.Parameters.Calibration.States;

namespace gp110i2
{
  class MainClass
  {
      public static void Main (string[] args)
      {
          GP110i dev = new GP110i();
          try {
              List<IParameterMetaDataBase> par= new List<IParameterMetaDataBase>();

              //Open a connection to the device
              dev.Open("10.0.1.4");

              //Disable calibrations (for both tubes)
              dev.EnableCalibration(false, true);
              dev.EnableCalibration(false, false);

              //Disable linearity
              for (uint i=0; i<12; i++) {
                  try {dev.EnableLinearityCheck(false, i, 0);} catch{}
              }

              //Disable logging
              dev.EnableLogging(false);

              //Set the dose units to REM because the linearity points below
              //are specified in R/h
              List<IParameterMetaDataBase> units = new
              List<IParameterMetaDataBase>();
              units.Add(new DoseUnits(DoseUnits.REM));
              dev.SetParameters(units);

              //Set the linearity points
              float []pt=new float[]{0.000463F, 16.4F, 118, 160, 325};
              dev.SetLinearityCheckPoints(pt);

              //Set the parameter we want to monitor into the list
              par.Add(new CalibrationStatus());

              //Create a parameter list to query for the check points
              //once the test is complete
              LinearityCheckPoints pts;
              List<IParameterMetaDataBase> lin=new List<IParameterMetaDataBase>();
              lin.Add(new LinearityCheckPoints());

              //Perform linearity test on each point
              for(uint i=0; i<pt.Length; i++) {
                  dev.EnableLinearityCheck(true, i, 0);

                  //Loop until linearity is done
                  bool loop=false, success=false;
                  do {
                      System.Threading.Thread.Sleep(2000);
                      loop = false;
                      par = dev.GetParameters(par);
                      foreach (IState state in (par[0] as CalibrationStatus).State) {
                          if (state is LinearityCheckBusy) loop=true;
                          if (state is LinearityCheckSuccess) success=true;
                      }
                      if (loop)
                         Console.WriteLine("Busy performing linearity check.  Time
                                   Remaining: "+(par[0] as
                                   CalibrationStatus).TimeRemaining + " (sec)");

                  } while(loop);



                  //Get the check points to display the information
                  pts=(dev.GetParameters(lin)[0] as LinearityCheckPoints);
```

```
                    //Display status for this test
                    if (!success)
                      Console.WriteLine("Linearity check did not succeed for: " + pt[i]
                                    + ", Average rate: " + pts.Points[i].AverageRate +
                                    " Percent Error: " + pts.Points[i].PercentError);
                    else
                      Console.WriteLine("Linearity check did succeed for: " + pt[i] +
                                    ", Average rate: " + pts.Points[i].AverageRate + "
                                    Percent Error: " + pts.Points[i].PercentError);


                }

            }
            catch(Exception ex) {
                Console.WriteLine("Error: " + ex);
            }
            finally {
                //Disable linearity
                try {
                   if (dev.IsOpen)
                      for (uint i=0; i<12; i++)
                         dev.EnableLinearityCheck(false, i, 0);
                }
                catch{}


                //Enable logging
                try {
                   if (dev.IsOpen) dev.EnableLogging(true);
                }
                catch{ }

                //Close any open connections
                try {
                   if (dev.IsOpen) dev.Close();
                }
                catch { }
            }

            Console.WriteLine("Done...");
        }
    }
}
```

Parameters

This section will list all of the parameters the device supports. A class instance represents each parameter. All parameters are contained in the canberra.protocols.gp110i.datatypes.parameters namespace. Each class contains all of the metadata associated with the parameter such as minimum value, maximum value, data type, description, name, read-only, read-write, etc…

This section will discuss the parameters and the associated namespaces.

Acquisition

The acquisition parameters are in the com.canberra.protocols.gp110i.datatypes.parameters.acquisition namespace. The table below lists the parameters.

Name	Description
DataLogRate	The rate in entries/second at which data is logged to the non-volatile memory. This parameter should never be written by a client
DataPushRate	This value is expressed in seconds and represents the period at which data is pushed out when 'Push' mode is enabled
DataStoreCount	DataStoreRate and this parameter determines the interval at which data is transferred from the non-volatile memory into the file system. This parameter should never be written by a client
DataStoreFilemax	Controls the number of files to be allocated for storage in the file system
DataStoreRate	DataStoreCount and this parameter determines the interval at which data is transferred from the non-volatile memory into the file system. This parameter should never be written by a client
DigitalAlarmState	The alarms and failures. See “Alarm” on page 57 for details. You can clear any alarms using the Clear() method.
DoseRateAlertAlarmSetpoint	The warning setpoint for the dose rate
DoseRateFiltered	The dose rate will be filtered using the 'Exponential Filtering' method. There are 2 GM tubes present and the device will switch between them based on the dose rate.

DoseRateHighAlarmSetpoint	The alarm setpoint for the dose rate
DoseRateUnfiltered	The raw, or unfiltered, dose rate. There are 2 GM tubes present and the device will switch between them based on the dose rate.
DoseTotalAlertAlarmSetpoint	The warning setpoint for the total dose
DoseTotalHighAlarmSetpoint	The alarm setpoint for the total dose
DoseUnits	The units the device uses to return the dose (0=uSv, 1=mR, uR=2, R=3, mSv=4, Sv=5). The dose rate will be returned using the same units but per hour values.
HighRangeTubeActive	State indicating that the high-energy tube is in use.
IntegratedDose	The total dose since it was last reset
Temperature	The ambient temperature in Celsius
TotalHighTubeCounts	Represents the number of events captured by the high-energy tube since the high range tube was assembled
TotalHighTubeSecs	It represents the number of seconds that the high-energy tube has been active since the high range tube was assembled
TotalLowTubeCounts	Represents the number of events captured by the low-energy tube since the low range tube was assembled
TotalLowTubeSecs	It represents the number of seconds that the low-energy tube has been active since the low range tube was assembled
LowTubeLifetimeDose	The total dose recorded since the low range tube was assembled
HighTubeLifetimeDose	The total dose resources since the high range tube was assembled
CustomPushList	The list of parameters to push to clients connected to the data push port. Push mode (Stream) is enabled via the StreamEnable parameter.

PushListSelect	0= Use Default Push List. Refer to the Push List section for more information 1= Use Custom Push List established by the operator through the CustomPushLIst parameter
AlarmStatusLatch	This parameter tells the device whether to latch or not-latch the alarm status once the condition that caused it clears. The LED will follow the alarm status. 0= not-latch 1= latch

Calibration

The calibration parameters are in the com.canberra.protocols.gp110i.datatypes.parameters.calibration namespace. The table below lists the parameters.

Name	Description
CalibrationDueDate	This parameter is indirectly set to the LastCalibrationDate + 2 years when the LastCalibrationDate parameter is written. It can be changed.
CalibrationStatus	The calibration status, see the “Calibration Status” on page 58.
HighRangeCalibrationFactor	The calibration factor for the high-energy tube
HighRangeCalibrationSource	The dose rate for the high-energy calibration source
LastCalibrationDate	The date the calibration was last performed
LowRangeCalibrationFactor	The calibration factor for the low-energy tube
LowRangeCalibrationSource	The dose rate for the low-energy calibration source
TubeBaseSensitivity	The tube sensitivity. This parameter is currently reserved.
MaximumLinearityCheckPoints	The maximum number of linearity checkpoints supported by the device.
LowGMlimit	The value represents the minimum dose range supported by the device
HighGMlimit	The value represents the maximum dose range supported by the device
LinearityCheckPoints	The linearity checkpoints.
LastAverageRate	The intermediate dose rate determined during linearity check
LastPercentError	The intermediate percent error determined during linearity check

Network

The calibration parameters are in the com.canberra.protocols.gp110i.datatypes.parameters.network namespace. The table below lists the parameters.

Name	Description
ConfigurationPort	The socket port used to configure the device
ConfigurationSessionTimeout	The number of milliseconds within which if there is no communication with the device over the socket it will be closed due to time out error.
EthernetAssignedIP	The assigned Ethernet network address
EthernetDhcpEnabled	The Ethernet DHCP enable state
EthernetGateway	The Ethernet gateway address
EthernetMacAddress	The Ethernet MAC address
EthernetStaticIP	The Ethernet static IP address
EthernetSubnetMask	The subnet mask for the Ethernet interface
MaxClients	The maximum number of simulataneously connected clients
StreamEnabled	The stream enable state
StreamingPort	The socket port used for streaming data to clients
StreamSessionTimeout	The number of milliseconds within which if there is no communication over the streaming port with the device over the socket it will be closed due to time out error.
TftpEnabled	The enable state for the Trivial File Transfer Protocol server
UPnPEnable	The enable state for Universal Plug and Play discovery
UPnPFriendlyName	The UPnP discovery name
UPnPTimeToLive	The UPnP Time to Live (number of network hops)

USBAssignedIP	The assigned USB network address
USBDhcpEnabled	The USB DHCP enable state
USBGateway	The USB gateway address
USBMacAddress	The USB MAC address
USBStaticIP	The USB static IP address
USBSubnetMask	The subnet mask for the USB interface
WebserverEnabled	The web server enable state
KeepAliveEnable	The enable state for network keep alive.
KeepAliveInterval	This value, expressed in seconds, determines the rate of the KeepAlive commands issued by the device's network interface to detect valid connections once the KeepAlive logic in the device has been activated. The KeepAlive logic remains inactive until the timeout determined by the Network_KeepAliveTimeout parameter has expired.
KeepAliveTimeout	This value, expressed in milliseconds, represents the amount of transaction-less time that the device allows to expire before activating its network interface KeepAlive logic. Once activated, the KeepAlive logic issues Keep Alive commands at the rate determined by the Network_KeepAliveInterval and terminates any open connections that do not acknowledge the command after nine un-acknowledged attempts.

System

The system parameters are in the com.canberra.protocols.gp110i.datatypes.parameters.system namespace. The table below lists the parameters.

Name	Description
DeviceDateTime	The date/time of the device
DeviceStatus	The device status, see the “Device Status” on page 59. You can clear any faults using the clear() method.
DiagnosticCounters	The diagnostic counters. This parameter provides information about the various diagnostic tests that are always running. You can clear the diagnostic counters using the clear() method.
FirmwareVersion	The firmware version
LastError	The last error that occurred. This will contain a text description of the error condition
Location	The location of the device
ProbeList	A list of other tubes discovered on the network
SerialNumber	The serial number of the device
UnitNumber	The unit number associated with the device.
WatchdogEnable	The watchdog enable state
WebUIversion	The web user interface version
GMtubeVoltage	The GM tube voltage
LvpsVoltage	The LVPS voltage
DiagnosticValue	An unnamed value returned from a diagnostic information request. Reserved for factory use.
StorageTotal	Total number of bytes for storage in device’s mass storage
StorageRemaining	Percent of device mass storage left

Name	Description
ManufacturingInfo	Information specific to the manufacturer

Alarm

This section will discuss the alarms and failures that may be returned from the DigitalAlarmState parameter. Each alarm and failure is represented by an instance of a class. These classes are contained in the com.canberra.protocols.gp110i.datatypes.alarms namespace. These classes are tabulated below.

Name	Description
DoseRateAlarm	Dose rate alarm limit has been exceeded.
DoseRateAlarm	Dose rate warning limit has been exceeded.
TotalDoseAlarm	Total dose alarm limit exceeded
TotalDoseWarning	Total dose warning limit exceeded
GMtubeFailure	A GM tube has failed
NonVolatileRAMFault	The nonvolatile RAM has a memory error. Settings may have been lost.
InstrumentOverrange	Over-range condition detected indicating that one or more readings have exceeded the device's maximum dose range.
RtcBatteryLow	The real time clock battery is low.

The items in blue are defined in a different namespace because they are used in other aspects of the system besides alarmings. This namespace is com.canberra.protocols.gp110i.datatypes.faults.

Calibration Status

This section will discuss the states that occur during a calibration. These states are reported through the CalibrationStatus parameter. Each state is represented by an instance of a class. These classes are contained in the com.canberra.protocols.gp110i.datatypes.parameter.calibration.states namespace. These classes are tabulated below.

Name	Description
Aborted	The calibration was aborted.
Busy	The tube is busy calibrating.
Failure	The tube failed to calibrate.
Success	The calibration process completed successfully.
TimedOut	The calibration process timed out.
LinearityCheckAborted	The linearity check was aborted.
LinearityCheckBusy	The tube is busy performing linearity check.
LinearityCheckFailure	The linearity check failed.
LinearityCheckSuccess	The linearity check succeeded.
Complete	Indicates that both GM tubes have reported that their respective calibration has been completed successfully.
HighTubeActive	Indicates the high GM tube is inuse for calibration or linearity.
FailedTooLow	Indicates the calibration or linearity check failed because the count rate is too low.
FailedTooHigh	Indicates the calibration or linearity check failed because the count rate is too high.
HighTubeCalibrated	The high range tube is calibrated.
LowTubeCalibrated	The low range tube is calibrated.

Device Status

This section will discuss the device states. These states are reported through the DeviceStatus parameter. Each state is represented by an instance of a class. These classes are contained in the com.canberra.protocols.gp110i.datatypes.status and com.canberra.protocols.gp110i.datatypes.faults namespace. These classes are tabulated below.

Name	Description
AcquisitionInactive	Acquisition is inactive.
CalibrationActive	The calibration or linearity check is active. It can be enabled by using the enableCalibration() method.
LogDisabled	Data logging is disabled. It can be enabled by using the enableLogging() method
HardwareInit	Error occurred while initializing hardware
LogInit	A critical error developed during initialization of the data log system. This can occur at init or during run-time as storage elements are manipulated.
LogSystem	A critical error was detected during data logging
LogTime	The timestamp for the last-logged entry was less than the previously logged entry
ParameterInit	Error occurred while initializing parameters
DeviceBusy	Asserted by the device after a hardware reset. It remains set for the duration of the initialization sequence. The initialization sequence can take up to about a minute as the device catalogs the entire historical database into memory. During DeviceBusy the device can be queried for status and other data but not for historical data because the catalog is incomplete.
Rebooting	Indicates the device is processing a Soft Boot or Factory Default command.
Updating	Indicates an upload or download transfer is in progress
CalibrationFault	The calibration or linearity check failed
SystemFault	A system fault has occurred. Look at LastError for a detailed description

Name	Description
RtcBatteryLow	The real-time clock battery is low and should be replaced as soon as possible.
GMtubeVoltageFault	The GM voltage is out of tolerance indicating a hardware power supply problem
LvpsVoltageFault	The internal LVPS voltage is out of tolerance indicating a hardware power supply problem

Logging Details

This section will discuss the different data types involved in data logging. There are basically two types. One is the log summary that provides summary information about the various files stored on the device. The other is the log data that is the data stored in the file.

Log Summary

The log summary is a class in the namespace com.canberra.protocols.gp110i.datatypes.historian. The class consists of the following properties. The log summary is retrieved by using the getLogSummary() method.

Name	Description
element	This is a number that represents the file on the device
numberOfEntries	The number of entries in the file.
startDate	The start date which is the date of the first entry
endDate	The date of the last entry
startSequenceNumber	The start sequence number.
endSequenceNumber	The end sequence number.

The sequence numbers primarily exists as an integrity check within the device. It is also used in conjunction with the start time stamp when retrieving logged data. If we consider a situation where the RTC was changed to an earlier time, the device could end up with new entries having an older time stamp than previously logged entries. That can remain that way, or go on for some time until the RTC is changed back. If the search used only the time stamp it would not return those entries whose time is not incremental. Instead, using the sequence number guarantees that all entries are returned and are returned in the proper order.

Log Data

The log data is a class in the namespace com.canberra.protocols.gp110i.datatypes.historian. The class consists of the following properties. This data is retrieved using the getLogData() method.

Name	Description
time	The date/time of the data
doseRate	The dose rate in the units of the device Units are specified by canberra.protocols.gp110i.datatypes.parameters.acquisition.DoseUnits
totalDose	The total dose in the units of the device. Units are specified by canberra.protocols.gp110i.datatypes.parameters.acquisition.DoseUnits
temperature	The temperature in Celsius
alarms	A list of alarms, see Alarms section
status	A list of device states, see Device Status section

Exceptions

This section will discuss the custom exceptions that may the thrown from any of the methods described in this document.

Exceptions have been defined in two namespaces canberra.exceptions and com.canberra.protocols.gp110i.exceptions. The exceptions that are thrown from the methods of the GP110i class are tabulated below.

Namespace	Name	Description
canberra.exceptions	DatasourceAlreadyOpenException	You invoked open() before closing an already opened connection
	DatasourceNotOpenException	You invoked close() without ever calling open()
	IllegalArgumentException	An illegal argument was passed to a method
canberra.protocols.gp110i.datatypes.exeptions	ChecksumException	The checksum received from the device does not match the computed checksum
	DeviceErrorException	The device returned an error. Invoke toString() method on this execption for details.
	InvalidResponseException	The device returned an invalid response.

Notes

Warranty

Canberra (we, us, our) warrants to the customer (you, your) that for a period of ninety (90) days from the date of shipment, software provided by us in connection with equipment manufactured by us shall operate in accordance with applicable specifications when used with equipment manufactured by us and that the media on which the software is provided shall be free from defects. We also warrant that (A) equipment manufactured by us shall be free from defects in materials and workmanship for a period of one (1) year from the date of shipment of such equipment, and (B) services performed by us in connection with such equipment, such as site supervision and installation services relating to the equipment, shall be free from defects for a period of one (1) year from the date of performance of such services.

If defects in materials or workmanship are discovered within the applicable warranty period as set forth above, we shall, at our option and cost, (A) in the case of defective software or equipment, either repair or replace the software or equipment, or (B) in the case of defective services, reperform such services.

LIMITATIONS

EXCEPT AS SET FORTH HEREIN, NO OTHER WARRANTIES OR REMEDIES, WHETHER STATUTORY, WRITTEN, ORAL, EXPRESSED, IMPLIED (INCLUDING WITHOUT LIMITATION, THE WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE) OR OTHERWISE, SHALL APPLY. IN NO EVENT SHALL CANBERRA HAVE ANY LIABILITY FOR ANY SPECIAL, EXEMPLARY, PUNITIVE, INDIRECT OR CONSEQUENTIAL LOSSES OR DAMAGES OF ANY NATURE WHATSOEVER, WHETHER AS A RESULT OF BREACH OF CONTRACT, TORT LIABILITY (INCLUDING NEGLIGENCE), STRICT LIABILITY OR OTHERWISE. REPAIR OR REPLACEMENT OF THE SOFTWARE OR EQUIPMENT DURING THE APPLICABLE WARRANTY PERIOD AT CANBERRA'S COST, OR, IN THE CASE OF DEFECTIVE SERVICES, REPERFORMANCE AT CANBERRA'S COST, IS YOUR SOLE AND EXCLUSIVE REMEDY UNDER THIS WARRANTY.

EXCLUSIONS

Our warranty does not cover damage to equipment which has been altered or modified without our written permission or damage which has been caused by abuse, misuse, accident, neglect or unusual physical or electrical stress, as determined by our Service Personnel.

We are under no obligation to provide warranty service if adjustment or repair is required because of damage caused by other than ordinary use or if the equipment is serviced or repaired, or if an attempt is made to service or repair the equipment, by other than our Service Personnel without our prior approval.

Our warranty does not cover detector damage due to neutrons or heavy charged particles. Failure of beryllium, carbon composite, or polymer windows, or of windowless detectors caused by physical or chemical damage from the environment is not covered by warranty.

We are not responsible for damage sustained in transit. You should examine shipments upon receipt for evidence of damage caused in transit. If damage is found, notify us and the carrier immediately. Keep all packages, materials and documents, including the freight bill, invoice and packing list.

Software License

When purchasing our software, you have purchased a license to use the software, not the software itself. Because title to the software remains with us, you may not sell, distribute or otherwise transfer the software. This license allows you to use the software on only one computer at a time. You must get our written permission for any exception to this limited license.

BACKUP COPIES

Our software is protected by United States Copyright Law and by International Copyright Treaties. You have our express permission to make one archival copy of the software for backup protection. You may not copy our software or any part of it for any other purpose.