Photonic Foundation Library:
Creating Software for Optical Component Tests
Getting Started Guide to
Programming the Photonics Foundation Library

Kazuo Yamaguchi

The Photonic Foundation Library is a library that enables the development of customized software to automate measurement processes, and keep up with the demand for high quality coupled with low cost.

This guide teaches extensive programming techniques using the Photonic Foundation Library and its pre-compiled accessory application which, together offer a complete package for the easy integration of solid test accuracy solutions.

Agilent Technologies

CONTENTS

Introduction

The demand for bandwidth is growing exponentially. The industry trend is similar to that experienced by the semi-conductor business a few years ago; there has been a shift away from the Research and Development phase, which focuses on specification and qualification, to the Manufacturing phase, which now delivers on system integration and flexibility.

- Specification and Qualification: instrument accuracy of tunable lasers and high precision optical sensors provides the basic measurement technology.
- Integration and Flexibility: simplify development of complex and time consuming test procedures by developing the customized software.

Here, the software is the key to the test system automation. Nevertheless, there have been many obstacles to developing such software:

- a lack of programming knowledge
- few resources of test definition
- high development costs

The following sections describe several measurement techniques that are introduced as library functions in the Agilent Photonic Foundation Library (PFL). The guide also features examples to demonstrate the ease of programming details involved in each test procedure. This guide demonstrates how the PFL increases the:
cost effectiveness of application development
time efficiency through test process automation

For more details of measurement techniques behind the programming of the PFL, please see the Application Note "Photonic Foundation Library: Enhancing Swept Loss Measurement" [1].

PFL – Definition

The PFL is a software library product that combines and fully integrates Agilent's test knowledge and experience with Agilent's outstanding optical measurement instruments such as the tunable laser sources, the polarization controller, and the power meter.
PFL includes:
a library for the most-demanding measurement and analysis procedures
ready-to-use and user-friendly function calls
instrument controls
specifications of measurement performance
Photonic Analysis Toolbox as a pre-compiled quick start application
a license-based management tool

Available Functions:

The test definitions of procedures for optical device specification have been gathered from many years of research. Common functions available in the PFL include:
precise insertion loss measurement to accurately capture spectral response for today's near loss-free components
polarization dependent loss measurement for the increasing demands of long distance transmission technology added in the field of DWDM
flexible comprehensive data analysis corresponding to strictly defined ITU-T specification

Programming Environment:

The PFL can be used in combination with a visual engineering environment, such as Agilent-VEE or LabView. You can then easily implement the PFL using its user-friendly function panels without worrying about parameter details. Of course, the PFL can also be integrated with any programming languages that supports our VXI Plug&Play drivers. [2]

Instrument Control:

You can use the functionality of the PFL to control the following Agilent test instruments in your test setup:
lightwave measurement mainframe
tunable laser source
optical power meter
polarization controller

The control of the instruments for certain measurement functions is embedded within the algorithm of the PFL, and the user does not need to know which instruments are needed to perform particular actions at particular times. The PFL takes care of both control and measurement.

Test Setup Specification:

The PFL is a software library with defined test specifications for use with suggested setups. This ensures the repeatability and integrity in different workstations for many test applications. The specifications include among others:
wavelength range
absolute/relative wavelength uncertainty
wavelength repeatability
relative insertion loss uncertainty
PDL uncertainty ... and more

The Photonic Analysis Toolbox:

The Photonic Analysis ToolBox application software, which is included in the PFL installation package, is a ready-to-use program after you have configured both the instruments and the GPIB. It allows the user to interactively see the power of both measurement and post processing of data in the PFL software.

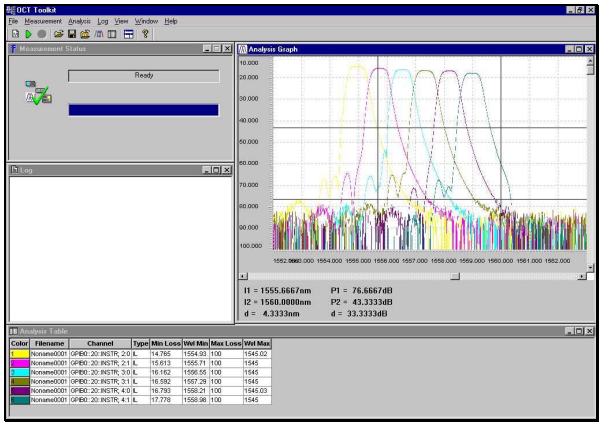


Figure 1: Display of the ToolBox, demonstrating spectrum measurement.

Licensing:

The PFL is a license-based product. A single license is locked to each controller PC used in a test setup. The computer fingerprint that distinguishes the uniqueness of one controller PC from another can be generated using the License Tool in the PFL. Each user is provided with the full capability of the library for their programming tool, and is only required to purchase a license for it's extended usage.

The PFL also offers a server-based license that allows multiple users to work on multiple test stations.

To obtain a single-user license, please order: N4150A Photonic Foundation Library, single user license. For multi-user and site licenses, contact your local Agilent Sales Office for more details.

Measurement Techniques

This section provides a brief overview of the passive optical component measurement techniques most commonly used in today's test routines and the integration of the PFL in each environment.

Insertion Loss Measurement

Polarization Dependent Loss Measurement

Real Time Measurement

Post Data Trace Analysis

Photonic Analysis Toolbox Application

Insertion Loss Measurement :

Among all optical component test and measurement techniques, the most important measurement is the insertion loss. This is especially true for dense wavelength division multiplexing (DWDM) applications where the insertion loss must be determined as a function of wavelength.

Insertion loss is the ratio of two absolute power readings: spliced or connectorized fiber directly linking from an optical laser source to an optical power meter without a device under test (DUT), and the same environment with an inserted DUT. Typical techniques for insertion loss measurement are shown in Figure 2 below.

For efficient long-duration measurements, like temperature and pressure controlled environmental tests that can extend over days, parallel measurements of DUT are desirable using either optical switching or multiple power metes.

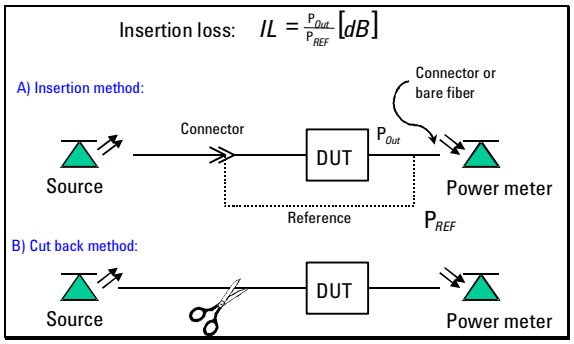


Figure 2: Principle of insertion loss measurement. Picture A) describes the insertion method; B) explains the cut back method.

The PFL integrates insertion loss measurement technique into a few library function calls and calculates the loss over wavelength with user defined precision. The measurement resolution and dynamic depend on the allowable test time that can be adjusted in the function by selecting the wavelength range, sweep speed, and dynamic range. The balance of data volume, dynamic, and resolution is configured in one function, visually showing the choice of parameter by using the function panel available in the visual programming environment. A sample function panel for absolute power measurement configuration is shown in Figure 2.

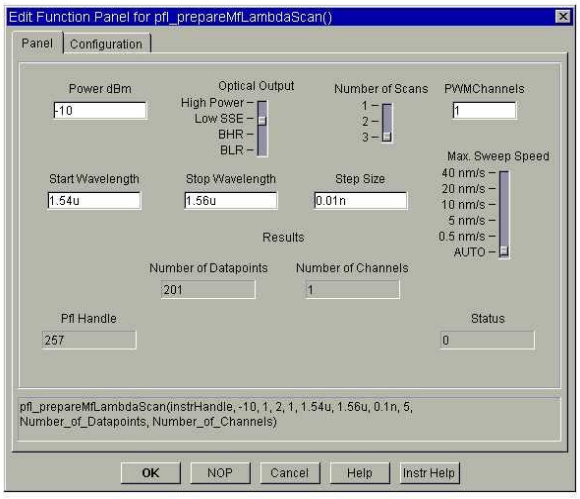


Figure 3: Sample function panel generated in the Agilent VEE Pro 6.0 programming environment.

Polarization Dependent Loss Measurement

: Recent studies into the characteristics of passive optical components in the telecommunications industry indicate that the polarization dependency of the transmission signal is a key influential test parameter. The dramatic increase in interest in polarization dependent loss measurement is due to demanding component requirements of high-speed long-haul data traffic.

Among the various measurement techniques, the PFL uses the "**Mueller Matrix Algorithm**" for the measurement of PDL to cover a wide wavelength range with high accuracy within a short test time. Unlike the "scrambling" method, which uses fixed wavelength points with long test time to test PDL uncertainty, the Mueller method takes advantage of swept-wavelength measurements that are performed by our instruments and handles the mathematics behind PDL calculation without requiring huge data input. A typical PDL measurement using the Mueller method is shown in Figure 4.

This test setup for PDL measurement is not new and has been well characterized. However, the PFL brings ease of system development, predefined specification, and accuracy enhancement to the Lightwave Measurement System 816x.

The optimization of both Agilent hardware, like tunable laser source, and software, like the PFL, ensures highest system accuracy because "we know our instruments best".

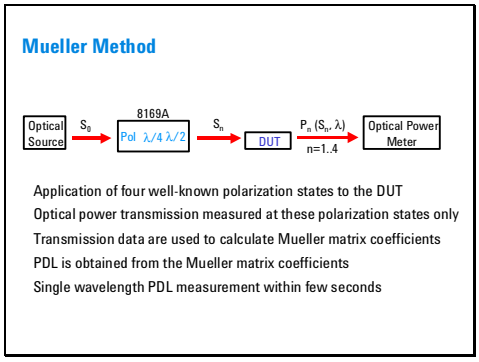


Figure 4: Principle of polarization dependent loss measurement. The setup is similar to insertion loss measurement, but in this case a polarization controller has also been included to measure loss at four predefined polarization states.

Real Time Measurement: Real time (or fast sweep) measurement is a new feature that the PFL makes available to your test system. It allows the user to continuously monitor the result of swept wavelength measurements without needing to individually stop each sweep. Imagine the functionality of an optical spectrum analyzer built into the software!

The capabilities to monitor spectrums for multiple channels and update data within a short measurement time are achieved using the PFL's enhanced technology.

Trace Analysis: Some other device parameters can be determined using the results of an insertion loss measurement. For example, the crosstalk between different channels for a DWDM multiplexer can be calculated by comparing the insertion loss (IL) values of different channels.

Automation Benefit

Over the years, the exponential growth of the optical communication field has been the driving force behind the need for massive increases in production volume, and improvements in accuracy combined with lower test costs. Even a novice programmer can benefit from the PFL's comprehensive collections of library functions, without needing to know the complicated programming algorithms behind the functions.

Such library functions reduce:

Cost of application development time

Cost of engineer or skilled person to operate the system

Cost of test time

The result will bring:

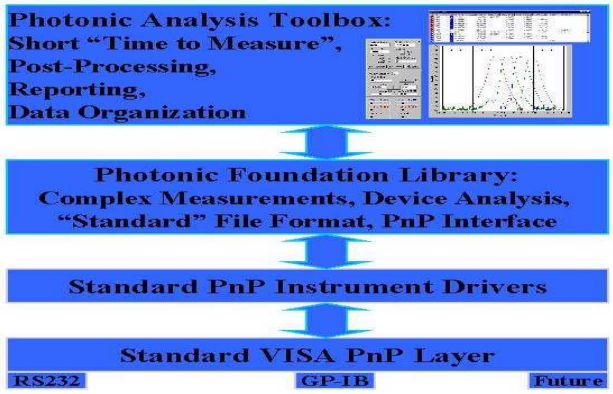
Simple programming and full functional application

Further analysis for loss measurement (i.e. crosstalk, etc.)

Focus on quality and yield

Layer Model of the Software Architecture

The PFL is based on our 816x VXI Plug&Play driver, the instrument control library for the 816x family [3].



Hardware Requirements

For detailed hardware requirements, see the PFL technical specification [2].

Sample Algorithm

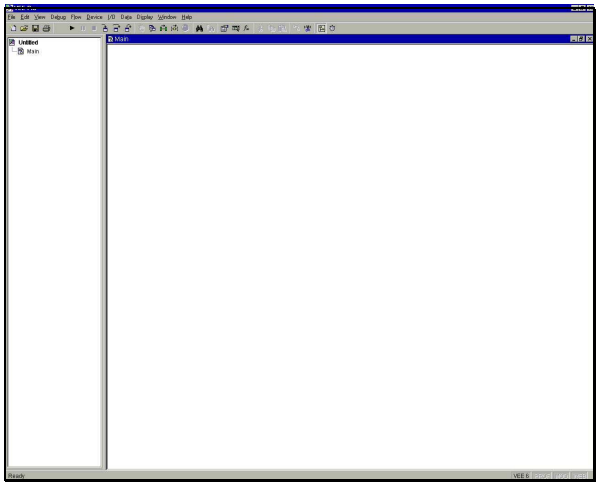
Sample algorithms for the Photonic Foundation Library are shown in various ways to let you comfortably get started with the library. This guide gives you step by step instructions on "How to PFL". The instructions are presented in two different programming environments:
Agilent VEE for visual programming environment using the PFL VXI Plug & Play library
C++ for algorithms using the PFL C API
The following instructions assume that Agilent VEE Pro 6.0 \ Microsoft Visual C++ and the PFL are properly installed. It also assumes that you have configured the GPIB card and installed the VISA driver to communicate with the instruments. [3]
The following step by step programming techniques will be described.
Agilent VEE Pro 6.0
Configuration & Initialization
Insertion Loss Measurement
Polarization Dependent Loss Measurement
Real Time Measurement
Trace Analysis
Microsoft C++
Configuration & Initialization
Insertion Loss Measurement
Polarization Dependent Loss Measurement
Real Time Measurement
Trace Analysis

(Agilent VEE Pro 6.0, VXI PnP library)

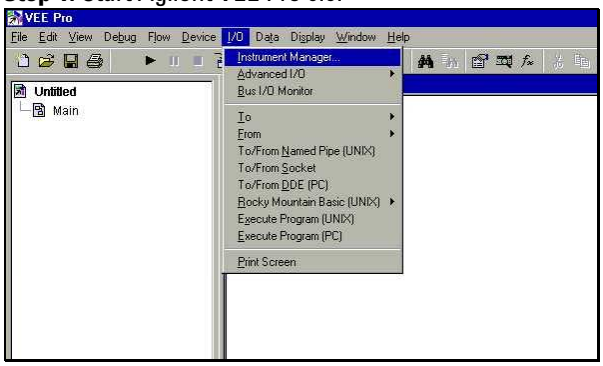
Configuration & Initialization

Agilent VEE Pro 6.0 is a visual engineering environment that allows the user to view the programming flow, the input and output parameters directions, and Help description for available functions.
The programming flow is called a sequence. Agilent VEE controls program sequence with unique input and output sequence pins at the top or bottom of objects (instructions). By connecting this sequence pin between two objects, VEE recognizes which instruction has to be executed first.
The object has unique input and output parameter pins. An input parameter pin can be seen at the left side of the object whereas output parameter pin is located at the right side of the object.
The function panel, also unique to the visual programming environment, provides easy access to the instrument library. The library can be seen as a list of functions that provide the visual representation of required parameters and their types.

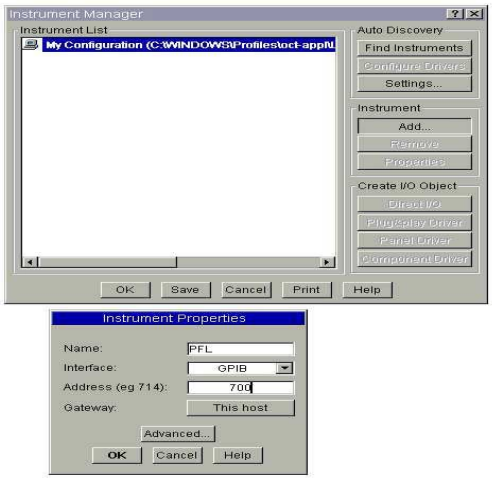
The Plug & Play library for 816x (mainframe with TLS and PWM), 8169 (polarization controller), and the PFL has to be properly configured in Agilent VEE Pro 6.0 in order to use the PFL VXI library functions in the source code.
The following name and GPIB addresses are used for each instrument described in this section:
816xA : Agi8164, addr=20
8169A : Agi8169, addr=22
PFL : PFL, addr=0 (no hardware address)



Step 1: Start Agilent VEE Pro 6.0.



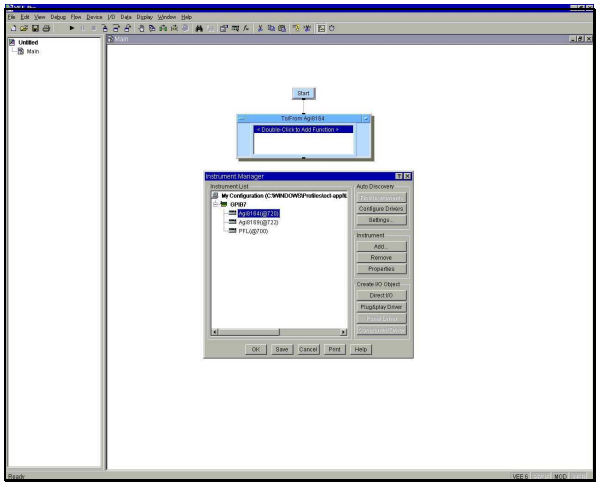
Step 2: All instruments that need to be controlled via software from VEE need to be configured in this **Instrument Manager** window.
Select [**I / O**] from menu. Select [**Instrument Manager**].



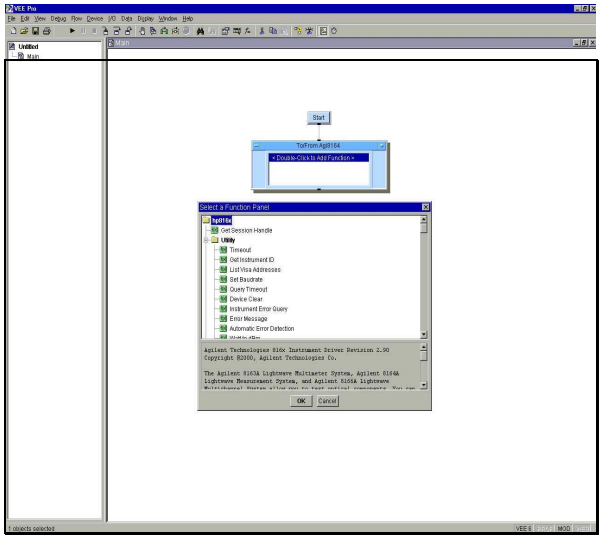
Step 3: This is an **Instrument Manager** window. If no configuration has been done before, the window should looks like this with empty instrument configuration.
To add the instrument, press [**Add...**]. **Instrument Property** window appears for names, interface, and GPIB address.
Enter name of library such as "**PFL**" for the PFL library and select "**GPIB**" for your interface. The GPIB address depends on your instruments and the GPIB card installed in your PC. The combination of primary address (7 in the diagram, specific to the GPIB card) and the secondary address (00 in the diagram, specific to the Instrument) is written as integer (700 in the diagram). Usually the GPIB card from National Instruments is configured as "14" and from Agilent as "7" for its primary address.

Step 4:
Click [Advanced...] on the Instrument Properties window to open the Advanced Windows Properties window, and then click the [Plug&Play Driver] tab.
If the PFL library driver is properly installed, you can select "**pfl**" as your "**Plug&Play Driver Name**".
Press [**OK**] to confirm the configuration.

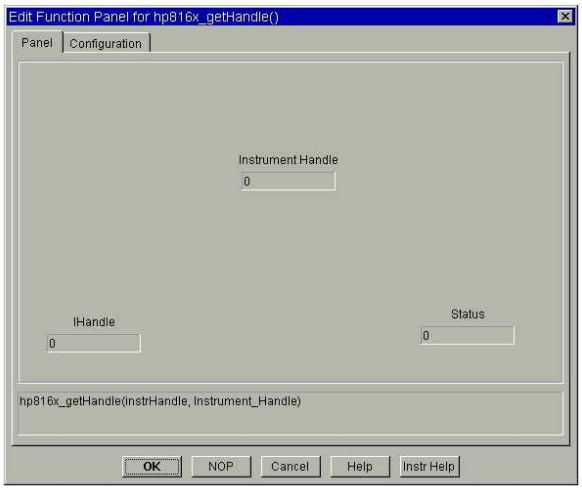
Step 5: Repeat the same configuration procedure to add the other instruments, 816x and 8169.
Select "**816x**" for 816x PnP driver and "**8169**" for 8169 PnP driver.
This completes the configuration procedure for Plug&Play drivers.
The PFL uses a few functions to initialize and close the library. These initialization and close routines are always reused in all the applications described in this document including insertion loss, polarization dependent loss, and real time measurements.



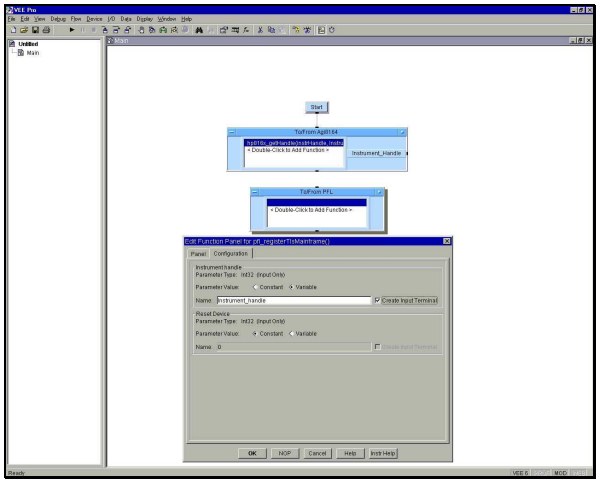
Step 6: The **Plug&Play Driver** button should be enabled in the **Instrument Manager** window after the configuration of the three instrument drivers.
Select **816x** (Agi8164 in the diagram) and press [**Plug&Play Driver**]. This will generate a single object box (**To/From Agi8164**) that you work with in the VEE main page.



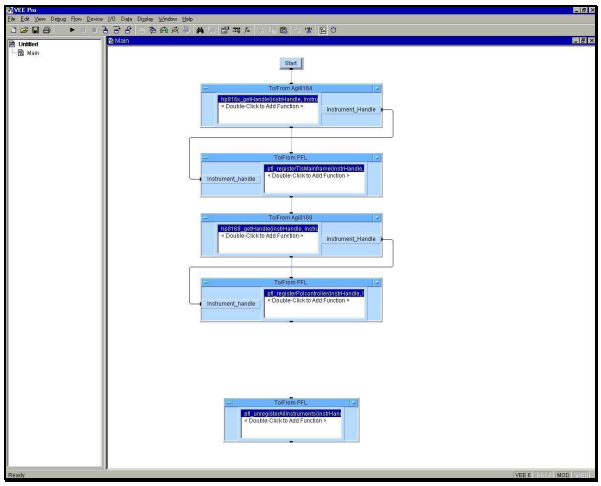
Step 7:
Double-click the (**To/From Agi8164**) object box.
The "**Select a Function Panel**" window appears displaying a list of Plug & Play drivers. These are the list of available functions for 816x. The same window will appear with 8169 and the PFL Plug & Play drivers with different library functions. A short description of each function is displayed at the bottom of the window when you select one from the list.



Step 8: In order to control the instrument via GPIB, the software has to generate an instrument handle to store instrument data such as error status and address. "**hp816x_getHandle**" and "**hp8169_getHandle**" functions of **Plug&Play** driver allow the programmer to generate the instrument handle for both the 8164A mainframe and the 8169A polarization controller for software control. This has to be defined once at the beginning of the program before any command/query to the instrument can be sent.
Select [**hp816x_getHandle**] from 816x Plug&Play driver list and press [**OK**]. The function panel shown in the above diagram appears.
Press [**OK**] without making any configuration changes in the panel.
Note: The [**Configuration**] tab shown in each function panel is used to change the name of input/output parameters, to redefine the size of parameter if it is an array, and/or to make parameter as input/output variable. Unless specified in the instruction of this guide, no change is required. Proceed with the change in function panel configuration for enhanced usage of the PFL library in the program.

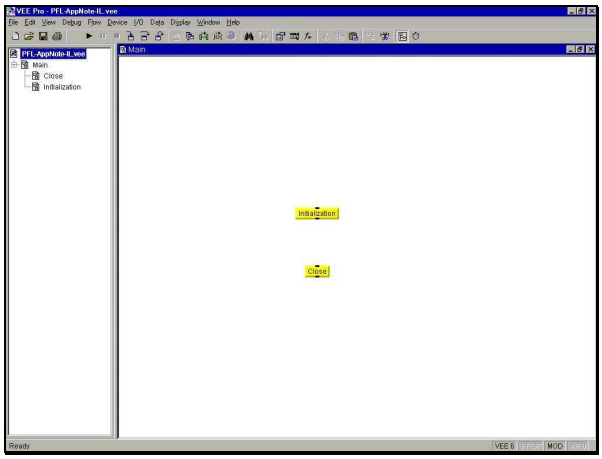


Step 9: Create a PnP driver object box for the PFL.
Select [pfl_registerTlsMainframe].
Check "**Instrument handle**" parameter as variable.
Check "**Create Input Terminal**" in [**Configuration**] of the function panel.
Press [**OK**] to confirm the use of this function and to enable the configuration change in the panel.



Step 10: Create PnP and PFL objects for the 8169 and use [**hp8169_getHandle**] and [**pfl_registerPolcontroller**] to register 8169 by following the same instruction steps described in 816x.
Create [**pfl_unregisterAllInstruments**] PFL object for close routine.

Connect output parameter pin of [**hp816x_getHandle**] to input parameter pin of [**pfl_registerTlsMainframe**].
Connect output parameter pin of [**hp8169_getHandle**] to input parameter pin of [**pfl_registerPolcontroller**].
The connecting sequence pin guarantees the sequence of library execution. These four functions are connected consecutively so that the order of execution is
[hp816x_getHandle]
[pfl_registerTlsMainframe]
[hp8169_getHandle]
[pfl_registerPolcontroller]



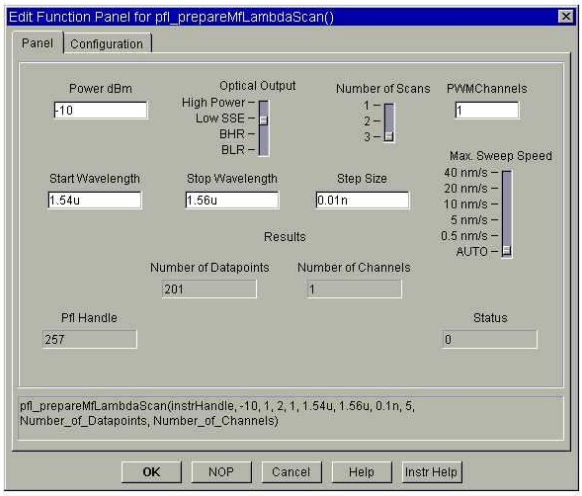
Step 11: Make Initialization and Close routine as user objects.
The instruction for creating a user object:
Drag the screen and highlight multiple objects by pressing ctrl+right mouse+mouse movement
Release mouse movement and right-click the screen
Select [**Create User Object**] in the list
Enter the name of the user object.

Module Summary
This is the end of Initialization and Close routine for Agilent VEE Pro 6.0. In this step sequence you have learned how to:
Configure the Plug&Play drivers
Initialize instruments in the test setup
Use a function from the PFL to close the library
For enhanced usage of the function, please review the sample programs provided in the PFL installation package.

(Agilent VEE, VXI PnP library)

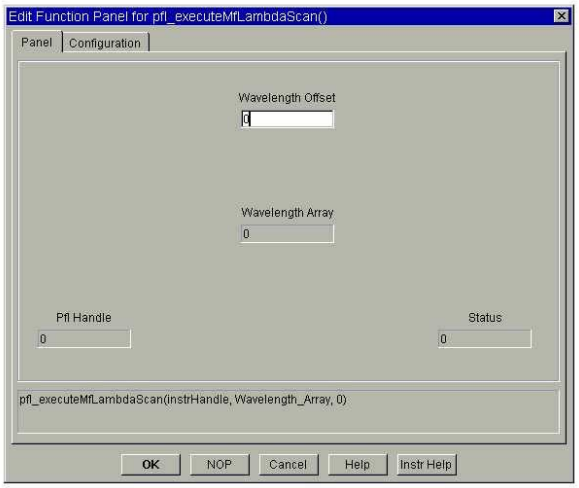
Insertion Loss Measurement:

Swept wavelength measurement is based around three PFL functions:



Step 1:

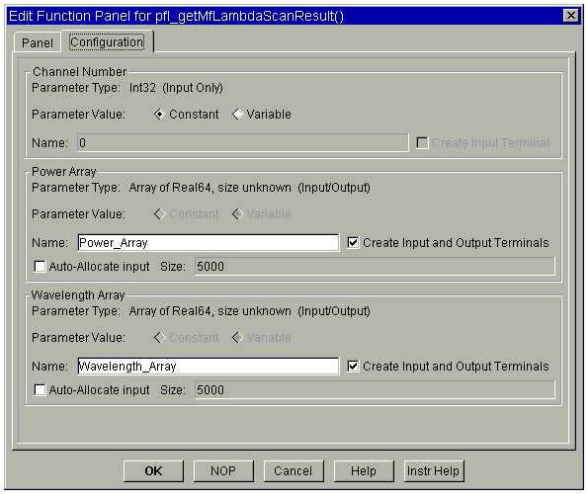
Create a PFL PnP object and then select "**pfl_prepareMfLambdaScan**" from function panel list. This function panel provides users with the required parameter setting for swept wavelength measurement. Edit the parameter settings according to your test system. Some of the parameters, such as **Low SSE** output and different wavelength range, may only apply to certain types of tunable laser sources.



Step 2:

Create a PFL PnP object and then select "**pfl_executeMfLambdaScan**" from the function panel list. This function starts executing measurements for the setting configured in "**pfl_prepareMfLambdaScan**". Think of this function as triggering the TLS when executed.

A parameter "**Wavelength Offset**" is set to 0 as default. To further improve the wavelength accuracy and repeatability, the PFL function, "**pfl_measureWavelengthOffset**", combined with an absorption gas cell provide a one step wavelength calibration procedure.[1]

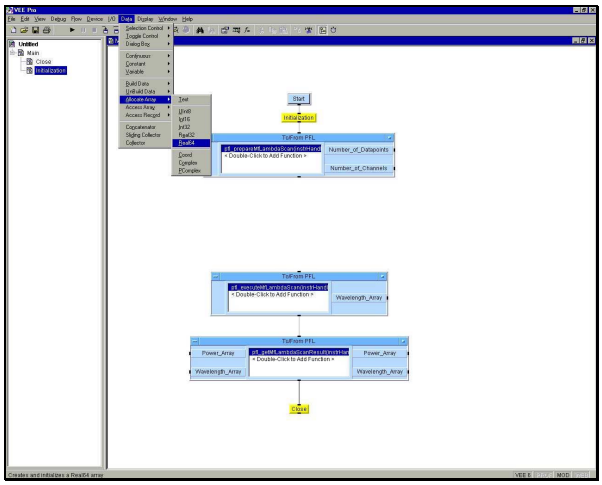


Step 3:

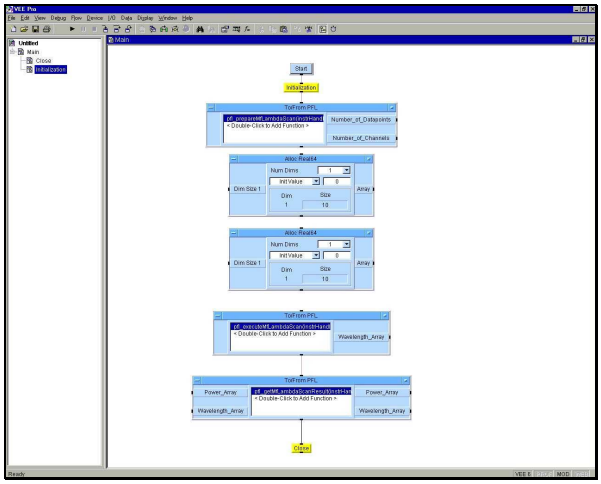
Create a PFL PnP object and then select "**pfl_getMfLambdaScanResult**" from the function panel list.

In the [Configuration] tab, check [Create Output Terminals] for "Power Array" and "Wavelength Array". A single click gives you a change in text from [**Create Output Terminal**] to [**Create Input and Output Terminal**] when unchecked.

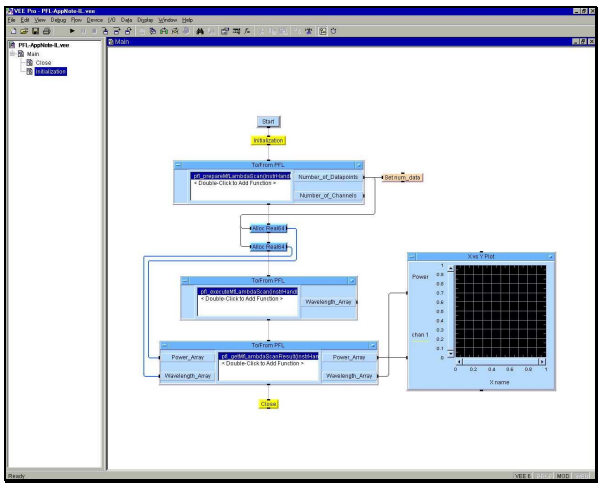
Single-click again to enable [**Create Input and Output Terminal**]. This will create input parameter pins for power array and wavelength array in this function object.
Note: To measure multiple channels at the same time, you must define "**Channel Number**" as a variable by selecting [**Variable**] in the **Configuration** tab. With this input variable, the algorithm has to handle a loop where it provides the counter to repeatedly execute this function until data from all channels has been read. You can find the programming algorithm in the PFL sample programs, provided in the installation package.
As an instruction of the first approach to the PFL, the number of channel used for the measurement is 1 (a default value of the function panel) and specified as a constant.



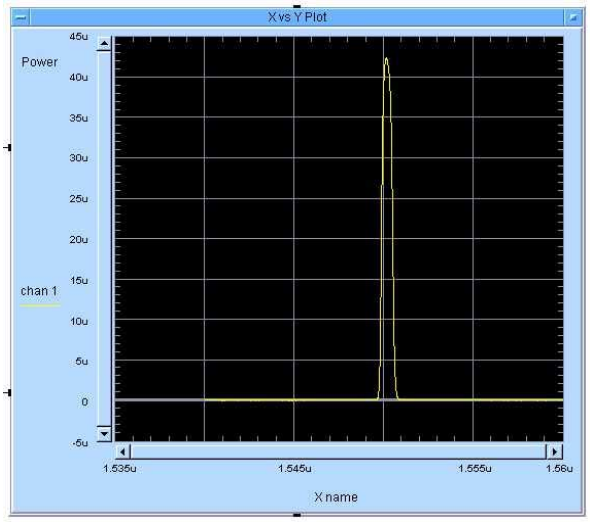
Step 4: The VEE **Main** screen will look similar to the graphic above if you have followed the sequential procedures described in previous steps.
Here is the current algorithm:
Initialization *
pfl_prepareMfLambdaScan
pfl_executeMfLambdaScan
pfl_getMfLambdaScanResult
Close *
*(user objects defined in the "**Initialization & Configuration**" module)
From the menu bar, select [**Data**], [**Allocate Array**], then [**Real64**] to allocate memory for a 1 dimensional array of 64 bits float. These memory spaces are needed for measured power data and wavelength data.



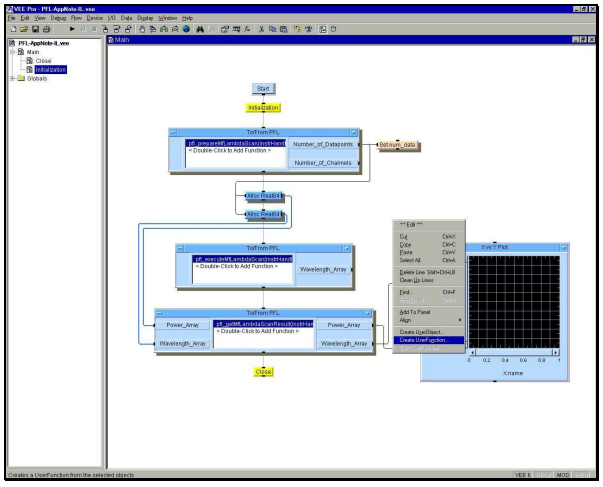
Step 5: The "**Alloc Real 64**" object box should look like above.
To make an array size variable, add data input terminal and select "**Dim Size 1**".
From drop-down list, select "**Init Value**" (the default is "**Lin Ramp**").



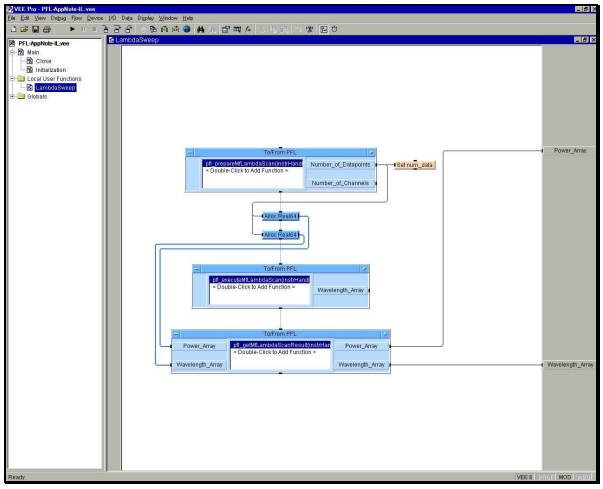
Step 6:
Connect sequence lines and output parameter pins of allocated memory to input variables of "**pfl_getMfLambdaScanResult**".
Generate an X Vs Y Plot object to see the results of swept wavelength measurement. These results only give an absolute power reading over wavelength, not insertion loss. A variable "**num_data**" (the pale orange object in the graphic above) is used to save number of data.



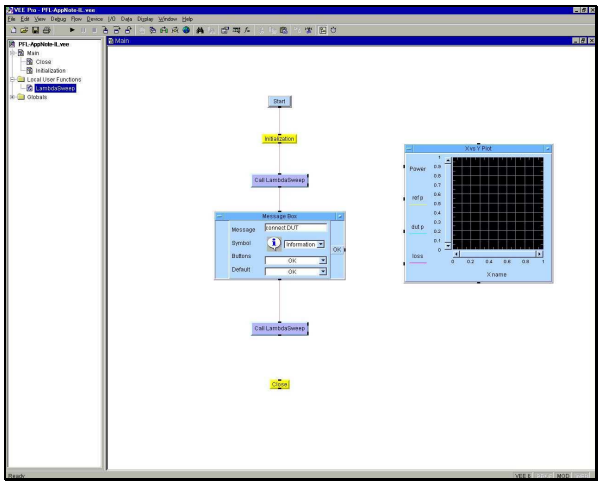
Step 7:
If the instruments are ready to be used, i.e. power is on and GPIB is set according to software setting, run the program.
A spectrum generated in the X Vs Y Plot graph is an absolute power reading (dBm) over wavelength of the characteristics of some passive optical device.
Note: This guide assumes that the user has a similar hardware setup to that described in Photonic Foundation Library Manual.



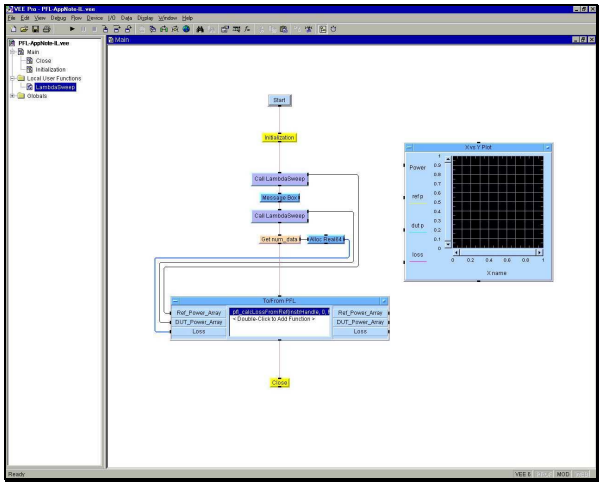
Step 8:
To advance from power measurement to loss measurement, the program has to be modified in a way that relative data can be calculated from the two absolute data: the results of reference and DUT measurements. Therefore, you need to twice execute the absolute power measurement routine and then calculate the loss from two data set.
First, create a user function by selecting:
[pfl_prepareMfLambdaScan],
[pfl_executeMfLambdaScan],
[pfl_getMfLambdaScanResult],
allocate array function, and variable to store number of data.



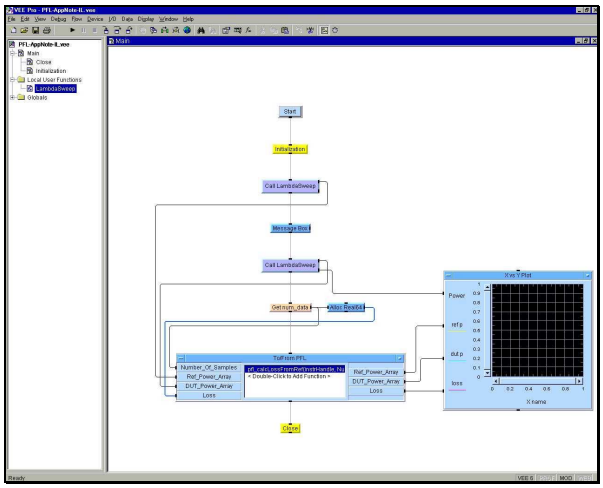
Step 9: The above graphic displays a created user function. Every object from Main is copied except **Initialization**, **Close**, and **Graph**.
Name this user function (the above diagram uses a name "**LambdaScan**", which will be used for descriptive purposes throughout the rest of this guide)
Create two output terminals: one for wavelength data and the other for absolute power data.



Step 10:
After creating a function to measure absolute power, call this function twice in Main.
In between the two measurements, position a message box to interrupt programming execution and give the user time to set up the DUT connection. The message description can be anything that instructs users how to connect up the device in the test system. Graphic inputs are also added to show "absolute reference power", "absolute DUT power", and insertion loss.

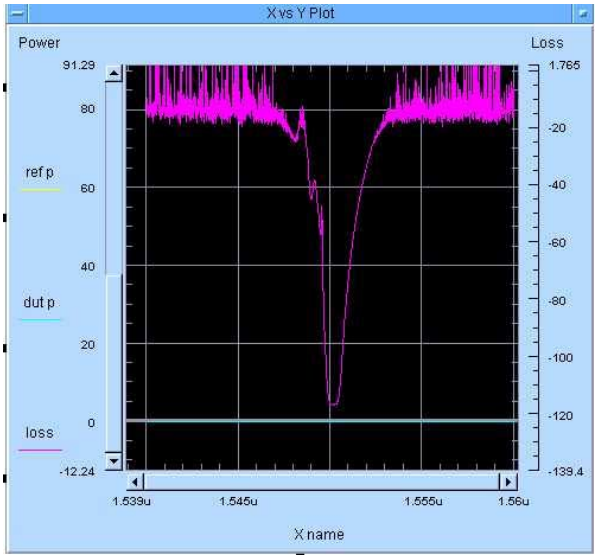


Step 11:
Create a new PFL VXI object and then select [**pfl_calLossFromRef**] from the function panel list. This function alone calculates the insertion loss for two absolute measurements.
Create input and output terminals and disable auto-allocate memory, which is configurable in the function panel.
A variable, "**Number_Of_Data**" from "**pfl_prepareMfLambdaScan**", is used to allocate memory for one of the inputs from [**pfl_calLossFromRef**]. The output parameter pin of the memory allocating object should be connected to "**Loss**" (default parameter name) in [**pfl_calLossFromRef**].



Step 12: The VEE **Main** screen looks similar to the above graphic after you have connected the sequence and data parameter to the graph. Here is the algorithm used for insertion loss measurement in this instruction.
call Initialization *
call LambdaScan **
message dialog for DUT connection
call LambdaScan **
allocate memory to store data from instrument
calculate an insertion loss
call **Close** *
display to **X Vs Y Plot** graph

* (user object defined in "**Init & Config**" section)
**(user function defined in step 8-9)



Step 13: After executing the insertion loss measurement for some passive optical DUT using the above program, the spectrum shown in the **X Vs Y Plot** displays the absolute reference power, the absolute DUT power, and the insertion loss.
The PFL not only calculates the difference between two absolute powers, but it also conducts post processing for signal delay and distortion effects to enhance the accuracy of Lightwave Measurement System 816x. [1]
Module Summary
You have now completed the programming instruction for Insertion Loss measurements. In this module you have learned how to apply the functionality of the PFL to:
Measure insertion loss using swept wavelength measurements for a DUT
Calculate the loss property of the device
For enhanced usage of this function, please review the sample programs provided in the PFL installation package.

(Agilent VEE, VXI PnP library)

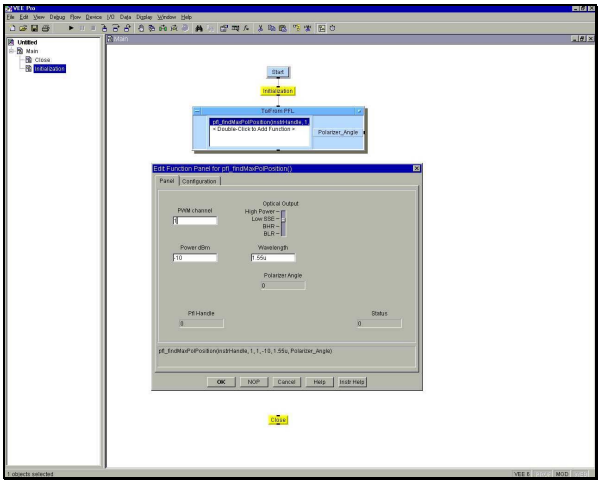
Polarization Dependent Loss Measurement:

The Mueller method is a commonly used method of Polarization Dependent Loss (PDL) measurement in test environments for passive optical devices with high wavelength resolution. This method yields the polarization dependency of loss, but also of other parameters such as passband ripple or crosstalk.

The Mueller method is advantageous to high volume manufacturers of passive optical components because it increases the manufacturing yield. This improved yield results from higher test accuracy and reduced test time over a wide wavelength range.

The complexity of the PDL measurement algorithm lies in the calculation that optimizes the polarizer used on the polarization controller, by choosing four well defined polarization states, and then calculates the PDL value. The defined polarized states are already programmed in the VXI Plug & Play driver of the PFL and are ready to use.

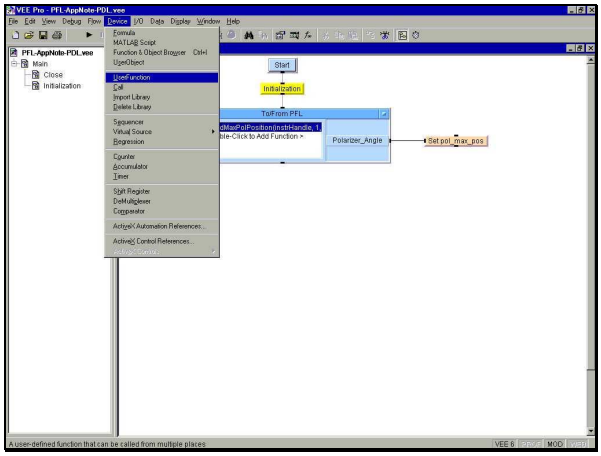
Also, the accuracy enhancement of the PDL correction algorithm is implemented in the PFL to overcome the wavelength dependency of the waveplates (λ/2 & λ/4 waveplates), in the polarization controller (8169A), when used in the wavelength outside of the instrument specification.



Step1:

Use the "**Initialization**" and "**Close**" user objects (yellow objects) defined in the Initialization and Configuration module of this guide.

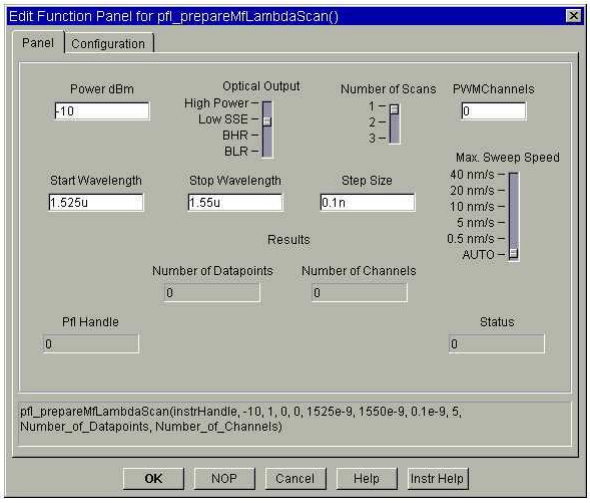
Add the PFL function panel, [**pfl_findMaxPolPosition**]. The polarizer in the polarization controller needs to be adjusted in such a way that the maximum dynamic range of optical sources can be used for the measurement. Configure the parameters accordingly.



Step 2: Similar to the core algorithm introduced in the insertion loss program, a user function is used to program the core algorithm of PDL measurement.

Select "**Device**" and "**User Function**" from the menu bar

Name the new user function. (The example diagram uses a name "**PdlLambdaScan**", which will be used for further descriptions in this guide)



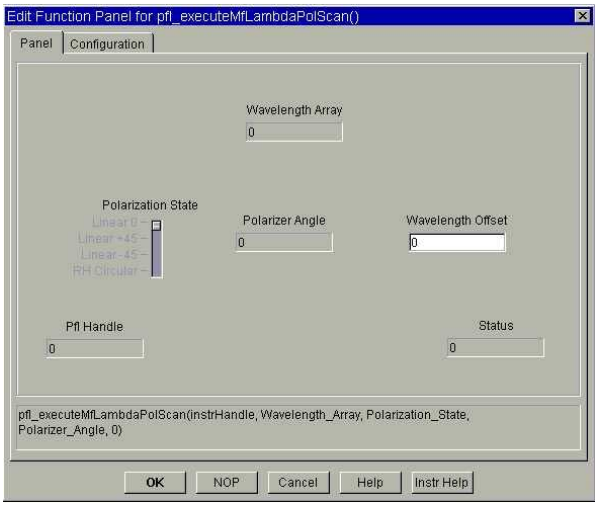
Step 3: Three core PFL Plug & Play functions are used:

[pfl_prepareMfLambdaScan]

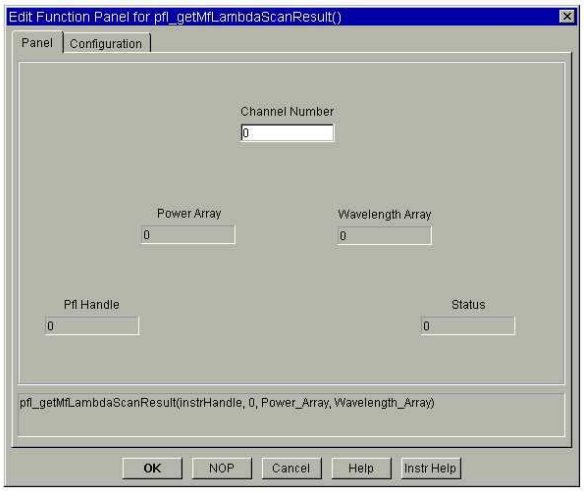
[pfl_executeMfLambdaPolScan]

[pfl_getMfLambdaScanResult]

This function panel provides users with the required parameter settings for swept wavelength measurement. Edit the parameter settings according to your test system. Some of the parameters, such as **LowSSE** output and different wavelength range, may not apply depending on the tunable laser source.

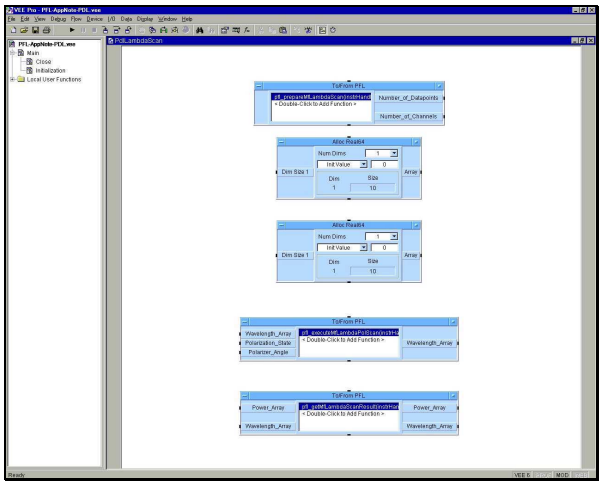


Step 4: The functionality of [**pfl_executeMfLambdaPolScan**] is similar to that of [**pfl_executeMfLambdaScan**] (described in step 13 of the Insertion Loss module) except that it has a polarization state and a polarizer angle for its additional parameters.
A polarization state parameter is an integer that defines the polarization states
A polarizer angle parameter is the angle of polarizer in 8169A searched by calling "**pfl_findMaxPolPosition**"
Adjust wavelength offset if wavelength calibration is done and needed for more accurate measurement.

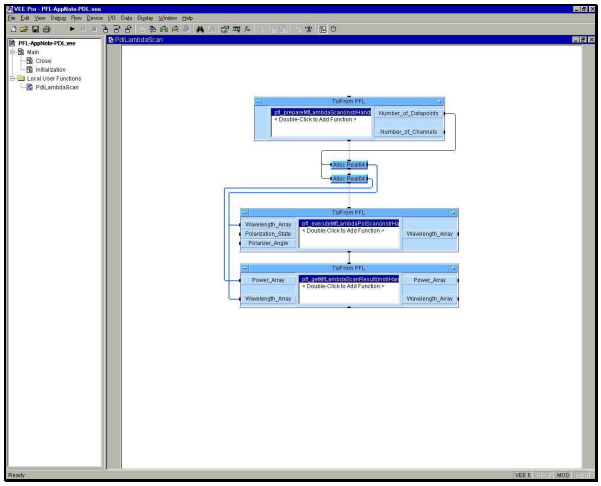


Step 5:
Create the PFL PnP object and select "**pfl_getMfLambdaScanResult**" from function panel list.
In [Configuration] tab, click [Create Output Terminal] for [Power Array] and [Wavelength Array].
Single click [Create Output Terminal] check box to give you a change in text from [Create Output Terminal] to [Create Input and Output Terminal].
Single click check box again to enable [**Create Input and Output Terminal**]. This will create input parameter pins for power array and wavelength array in this function object.

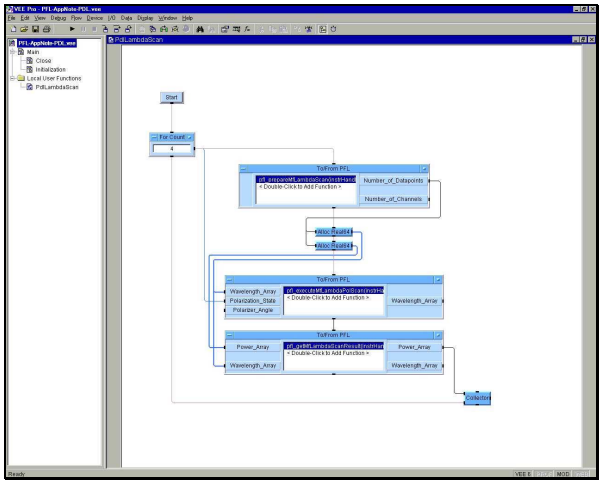
Note: To measure multiple channels at the same time, you must make "**Channel Number**" a variable by selecting "**Variable**" in the "**Configuration**" tab. With this input variable, the algorithm has to handle a loop where it provides the counter to repeatedly execute this function until data from all channels has been read. You can see this programming algorithm in the PFL sample programs that are included in the installation package.



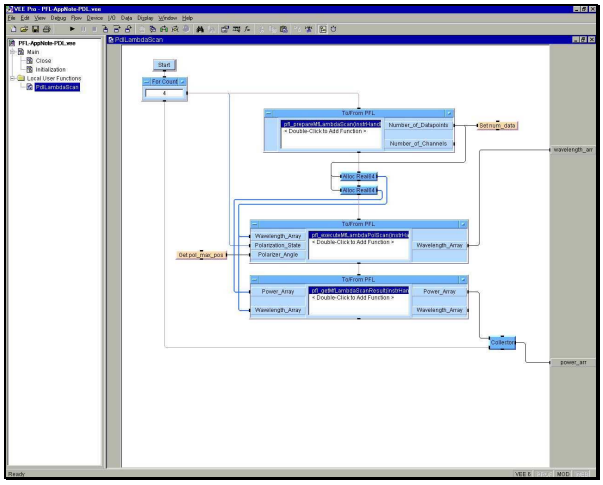
Step 6: The [**Alloc Real 64**] object box should look like the above graphic.
To create an array size variable, add the data input terminal and select [**Dim Size 1**].
From the drop-down list, select [**Init Value**] (the default is [**Lin Ramp**].



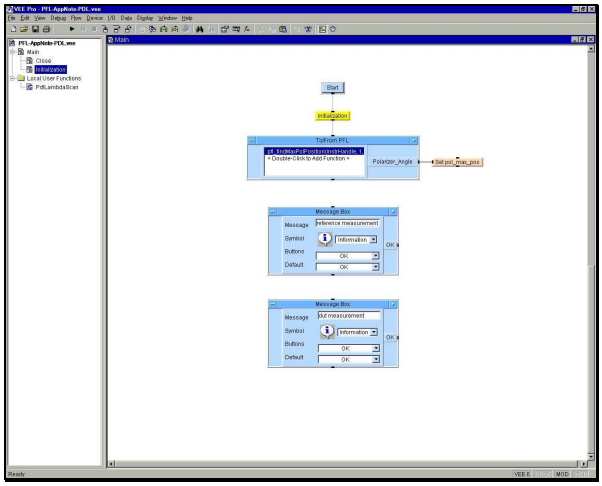
Step 7: The output parameter of "**pfl_prepareMfLambdaScan**", number of channel, is connected to the [**Allocate Memory**] object to make storage size appropriate to data size.



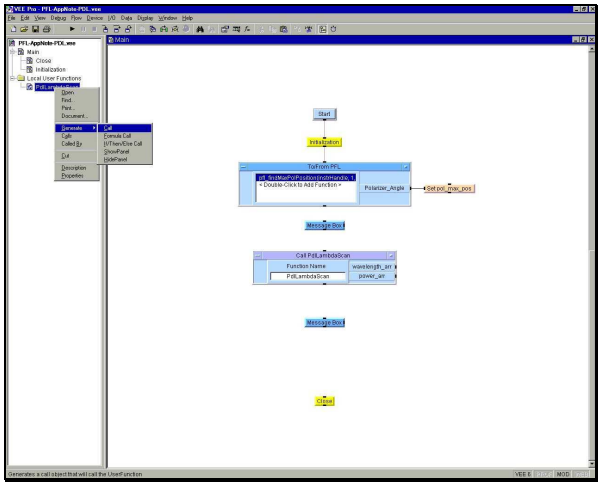
Step 8: As described in the definition of Mueller method, four absolute powers are measured at four different polarization states.
"For loop" is appropriate to give index to polarization states, an input parameter used in "**pfl_executeMfLambdaPolScan**". 1 dimensional array of result data is temporary stored in collector object, which then will output 2 dimensional array (4 polarization states, data size).



Step 9: A variable polarizer angle (pale orange object) is used as another input parameter of "**pfl_executeMfLambdaPolScan**". This variable is set by calling "**pfl_findMaxPolPosition**" as already described in step 1. By convention, data size is stored in a variable.
Create two outputs: wavelength data and 2 dimensional array power data
Connect the wavelength output parameter pin to the wavelength data output of this function
Connect the power data output parameter pin from the collector object to the power data output of this function

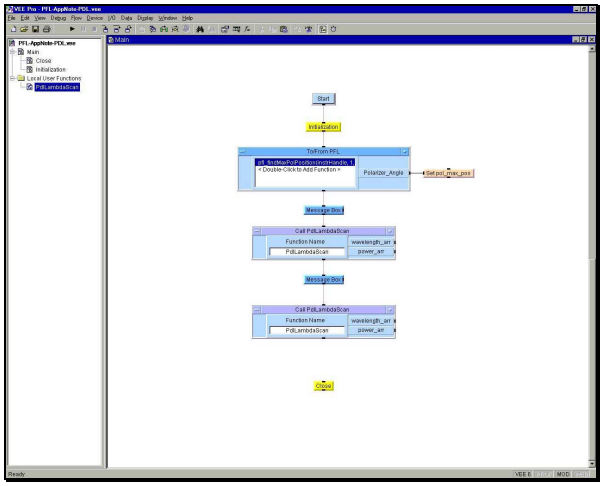


Step 10:
Back to main.
Introduce two message box. One describes user to connect reference fiber and one to connect DUT (device under test) in the test system.
Like insertion loss measurement, polarization dependent loss measurement also compares data referenced with patch cable and test device inserted.



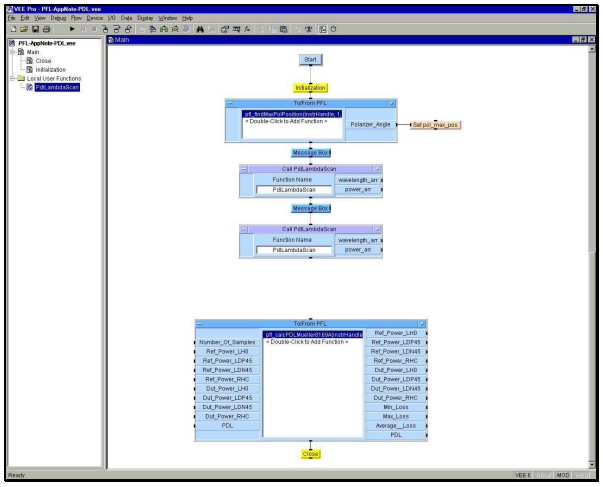
Step 11: Call a user function created in step 2-9 ("PdlLambdaScan" in the diagram of this paper).
Under [**Local User Function**] of program explore in VEE screen, select your user function and right click.
From menu, choose [**Generate**] then [**Call**].
Place the user function object box below two message box.

Reusing the same user function is possible here since the same procedure applies to reference and DUT measurement.



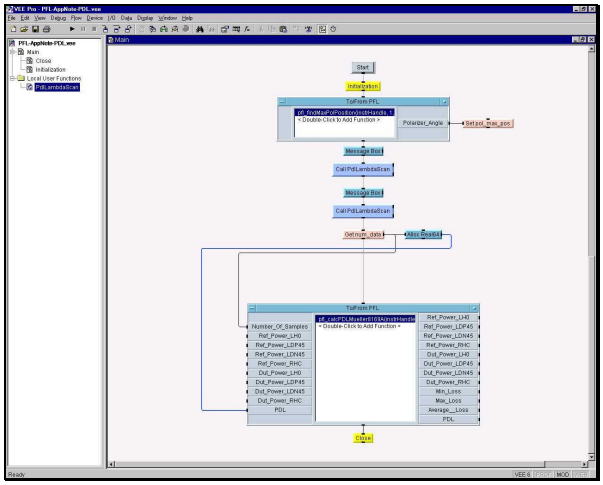
Step 12: After connecting the sequence to control the program flow, the program looks like the diagram above.
The sequence is:
Initialization *
"pfl_findMaxPolPosition" **
message box for reference
"PdlLambdaScan" ***
message box for DUT
"PdlLambdaScan"
close *
* (user object defined in Init. & Config. module)
** PFL function
***(user function defined in steps 2-9)

The measurement part of the program has been completed. The rest of the instruction for PDL measurement describes the calculation of PDL value derived from reference and DUT data.

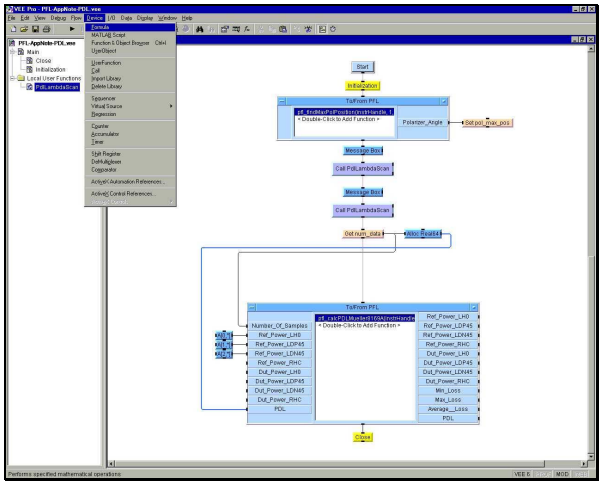


Step 13:
Create a PFL PnP object and then call the "**pfl_calcPDLMueller8169**" function.
The following input parameter pins have to be created to pass data to the function. The function configuration tab allows the user to choose each parameter as a variable.
The parameters to create are:

Number_Of_Sample
Ref_Power_LH0
Ref_Power_LDP45
Ref_Power_LDN45
Ref_Power_RHC
Dut_Power_LH0
Dut_Power_LDP45
Dut_Power_LDN45
Dut_Power_RHC
PDL
(default names for input parameters)
Bot start and step wavelength parameters can also be input variable if defined previously.

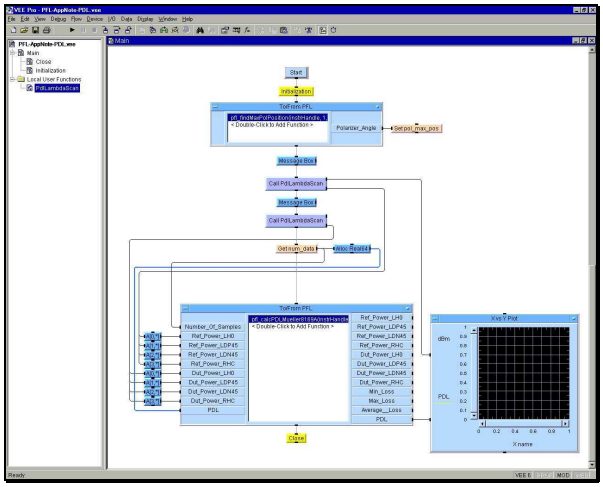


Step 14:
A variable, number of data, set in the user function is called and used as an input [**Number_Of_Sample**] to the "**pfl_calPDLMueller8169**" function.
Allocate memory, 1 dimensional array for 62 bits float, and use it as input for "**PDL**" input parameter.



Step 15: A [**Formula**] object is used to distribute data for each input parameter to calculate PDL.
The [**Formula**] object is generated by choosing [**Device**] in the menu bar, then [**Formula**] from drop-down list.

The output of the user object, "**PdlLambdaScan**", is a 2 dimensional array, (4 polarization states by data size). Therefore, *refarray [0,*],* gives reference data measured at a linear horizontal polarized state.
"*" in the index is used by VEE to select all rows. Here is the list of formulae used in the program:
Ref array [0,*] – LH0 (linear horizontal)
Ref array [1,*] – LDP45 (linear diagonal positive 45)
Ref array [2,*] – LDN45 (linear diagonal negative 45)
Ref array [3,*] – RHC (right hand circle)
Dut array [0,*] – LH0
Dut array [1,*] – LDP45
Dut array [2,*] – LDN45
Dut array [3,*] – RHC



step 16:
Connect the reference measurement output and the absolute power data generated from the first user function call, to the first four formulae listed in step 14. Connect the DUT measurement output and the absolute power data generated from the second user function call, to the next four formulae in the list
Create **X Vs Y Plot** graph to display PDL over wavelength. Take the wavelength data from the wavelength array output generated by the user function. You can also view minimum loss, maximum loss, and average loss by adding input to the **X Vs Y Plot**. (Remember to allocate memory with proper data size for each loss parameter to be displayed)
Notice that only five PFL VXI commands, excluding **Initialization** and **Close**, are used to demonstrate PDL measurement. No complex calculation and instrument communication algorithm is added.

Module Summary
You have completed the programming instruction for PDL measurement. In this module you have learned how to apply the functionality to:
Measure PDL
Calculate the PDL value derived from reference and DUT data.

For enhanced usage of the function, please review the sample programs provided in the PFL installation package.

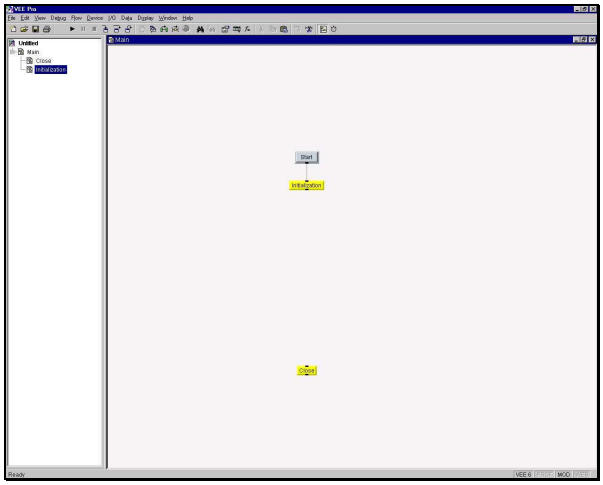
(Agilent VEE, VXI PnP library)

Real Time Measurement:

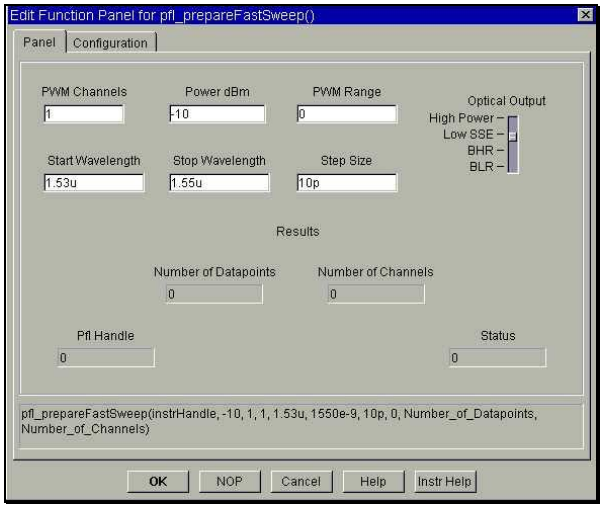
With a short program development time, the user is able to implement a powerful tool to continuously monitor the absolute power reading, while adjusting the position of test devices like thin film filter or writing fiber brag grating.

This real time sweep is an enhanced feature that synchronizes swept wavelength measurement technology with upgraded firmware of tunable laser source and mainframe, by improving both the hardware and software trigger method.

The software consists of only four PFL VXI commands, excluding "**Initialization**" and "**Close**" routines, and two of them are included within a loop to continuously activate directional sweep without interruption.



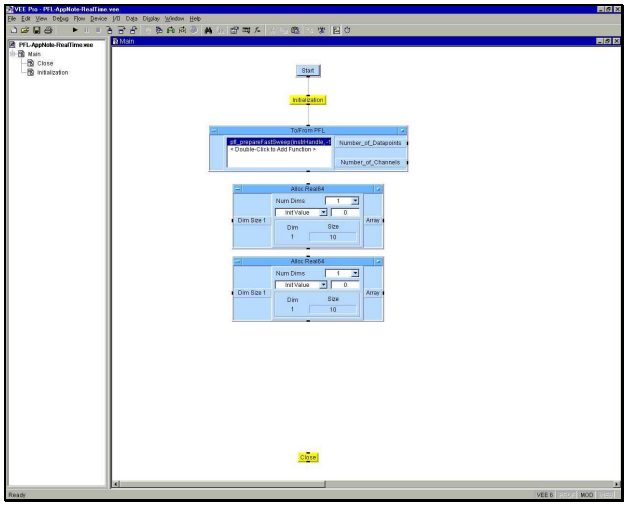
Step 1: Use the "**Initialization**" and "**Close**" user objects defined in the Initialization and Configuration module earlier in this guide.



Step 2:

Create the PFL VXI object and then select the "**pfl_prepareFastSweep**" function.

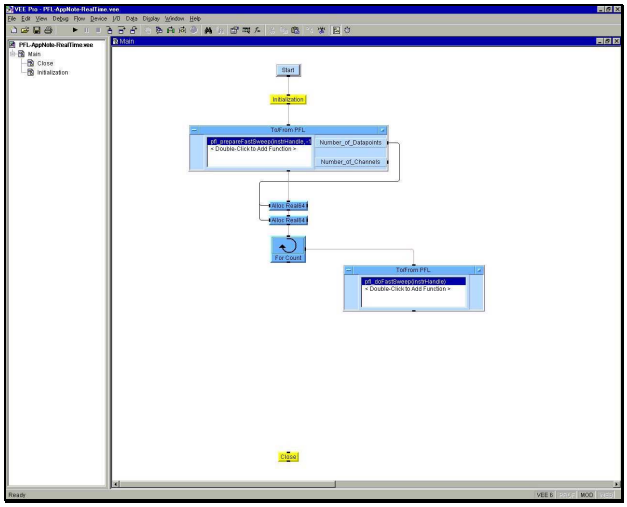
Set parameters according to your test setup.



Step 3: You need to allocate memory for wavelength data and absolute power data reading. To allocate memory:

Select [**Data**] from VEE menu bar and then select [**Allocate Array**]

Select [**Real64**] to allocate memory for 1 dimensional array of 64 bits float.



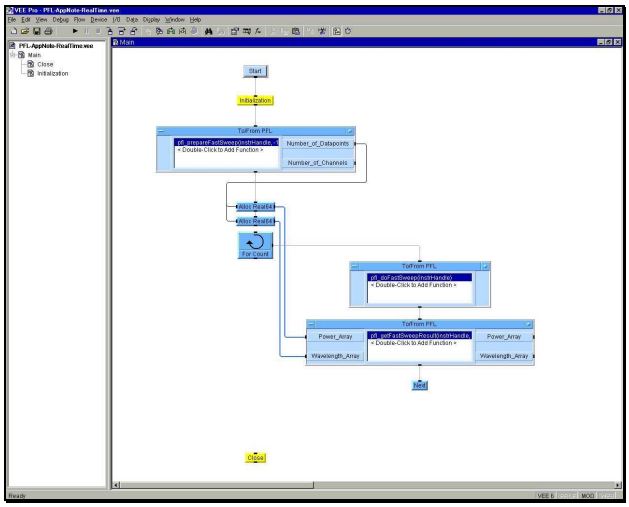
Step 4: A loop object is used to repeatedly execute the PFL function for real time measurement. To select the **FOR** loop functionality to execute a defined number of repeat:

Select [**Flow**] in menu bar and [**Repeat**]

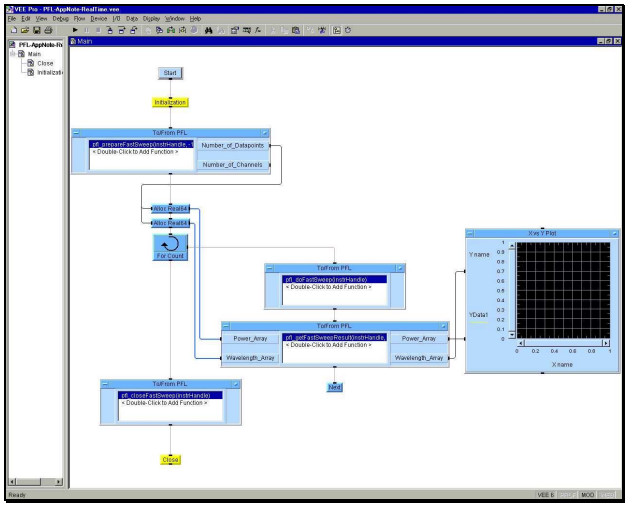
Select [**For Count**] from the list. The default execution counter, 10, is used for the purposes of this demonstration.

The repeat procedure described here is **FOR** counter but other repeat procedure such as **UNTIL BREAK** or **ON CYCLE** loops can also be implemented according to your requirement. In these cases, some program halt procedure must be programmed in order to execute the PFL closing routine.

Create the PFL VXI object and select "**pfl_doFastSweep**" function.
Connect the sequence pin from the repeat object to the function to include "**pfl_doFastSweep**" in the loop.



Step 5: Create the PFL VXI object and select the "**pfl_getFastSweepResult**" function from the list.
This function allows the retrieval, via the GPIB cable, of data stored in the memory buffer of the power meter. Number of channel, number of slot, number of data to be measured, and wavelength range are some of the factors that affect total measurement time. Nevertheless, one sweep takes less than one second per channel.



Step 6: Create a PFL VXI object and select "**pfl_closeFastSweep**". This is a closing routine not included in the normal lambda scan procedure described in insertion loss and PDL loss measurement. The trigger to the hardware has to be carefully handled by the software in order to achieve maximum communication speed with the instruments and minimum loss time.
The **X Vs Y Plot** graph is added to display the swept measurement update.

The VEE main screen looks similar to above after connecting the sequence and data parameters to the graph. Here is the algorithm used for real time measurement in this instruction:
call Initialization *
call "pfl_prepareFastSweep"
allocate memory for wavelength and power data array
loop
call "pfl_doFastSweep"
call "pfl_getFastSweepResult"
display to **X Vs Y Plot** graph
end loop
call "pfl_closeFastSweep"
call Close *
* (user object defined in "Init & Config" section)

Module Summary
This completes the programming instruction for real time sweep measurement. In this module you have learned how to:
Perform a real time measurement using the PFL fast sweep functionality
For enhanced usage of the function, please review the sample programs provided in the PFL installation package.

(Agilent VEE, VXI PnP library)

Trace Analysis:

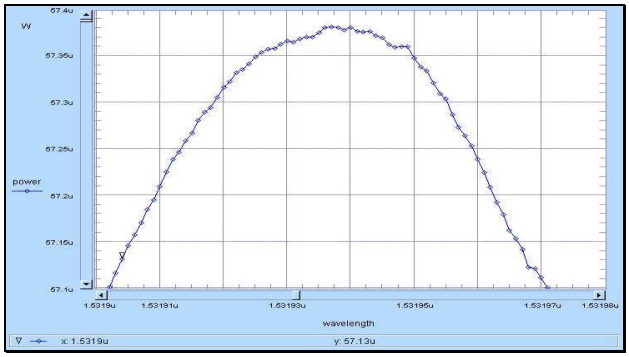
Some of today's most demanding device characteristics can be easily analyzed using trace analysis functions provided by the PFL. Such functions include among others:

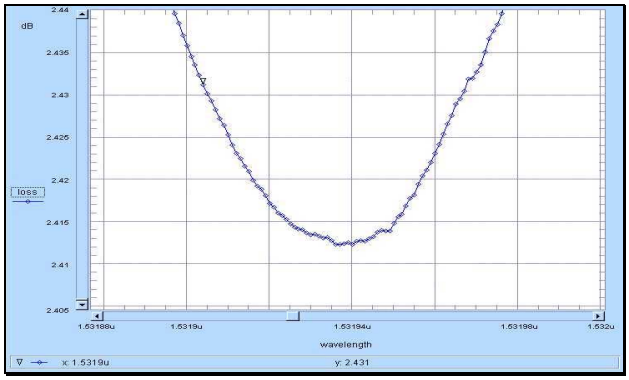
losses at ITU / peak of the spectrum
n dB bandwidth
wavelength at minimum / maximum loss
ripple

With conjunction to insertion loss measurement, post data processing is one more added value provided with the library.

This module features a programming example followed by a description diagram. The PnP capability that gives a programmer ready-to-use function parameters applies here also. The description of each input and output parameters in the function is shown in the description diagram. Comparing the description diagram to the actual function library can guide the user in deciding what values to set and what data to expect.

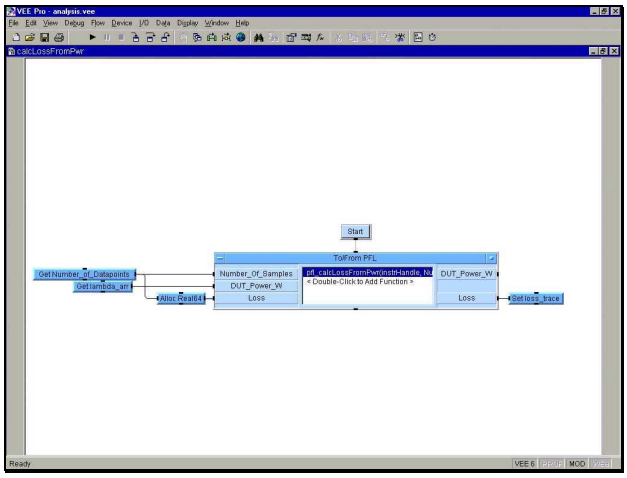
Some trace analysis functions take an input parameter that is an output of other function. This parameter conjunction can distinguish difference in the device characterization based on its given input parameters. See "**pfl_ndBPeakAnalysis**" as an example. Unless noted, the following description assumes that each step is not a sequential order.





The two diagrams show the result of insertion loss measurement. The top graph is an absolute power reading, and the bottom graph is an insertion loss with a reference at TLS power level. The following trace information is used for the rest of the description.

trace start wavelength : 1530nm
trace stop wavelength : 1535nm
trace step wavelength : 1pm
laser power : -10 dBm
sweep speed : 0.5 nm/s
DUT channel spacing : 100GHz
DUT ITU specification : 26 (1531.90nm)



Step 1: The "**calcLossFromPwr**" function calculates loss from the given absolute power data. This function uses the TLS output power level as a reference value, which means it does not take into account any loss occurred in the connection or the fiber. Instead, "**calcLossFromRef**" takes two absolute power data readings, one with a device and the other with a reference cable. Both functions provide calculated loss data that can be used in all trace analysis functions.

At least three input parameters have to be created in the configuration tab of the function (assuming the reference power level (in dBm) is written in the function panel of "**calcLossFromPwr**"):

number of data point
DUT power reading (in W)
allocated memory space for loss data

The number of data point is data provided by "**pfl_prepareMfLambdaScan**" and DUT power reading (in W) is a result of absolute power data generated by "**pfl_getMfLambdaScanResult**" function, both of which are introduced in the insertion loss measurement routine.

For loss variable, memory size is allocated to the number of data points each with 64 bits real number using [**Alloc Real64**] object found under [**Data**] -> [**Allocate Array**] -> [**Real 64**] of VEE menu bar.

Any other input parameters used in the description need to be adjusted according to the device characteristics.

For convenience, the following input variables are

defined prior to their use in the sample program shown in this guide:

Number_Of_Datapoints – number of data points measured

lambda_arr – absolute power reading for device test

loss_trace – loss data, referenced at TLS power level

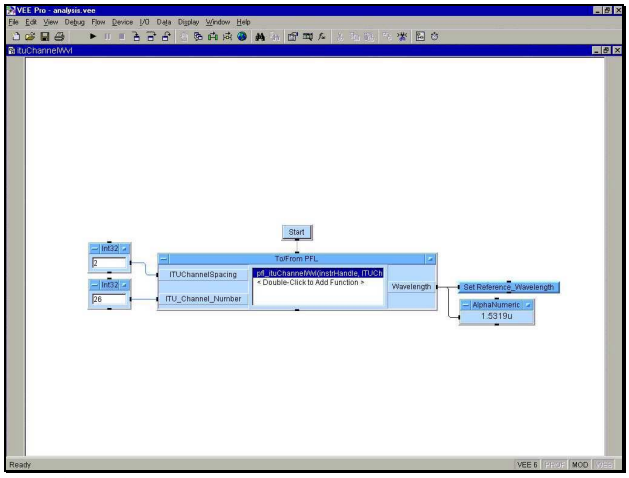
Trace_StartWvl – start wavelength of the whole trace

Trace_StepWvl – step wavelength of the trace

Start_Wavelength – user defined start wavelength within the trace*

Stop_Wavelength – user defined stop wavelength within the trace*

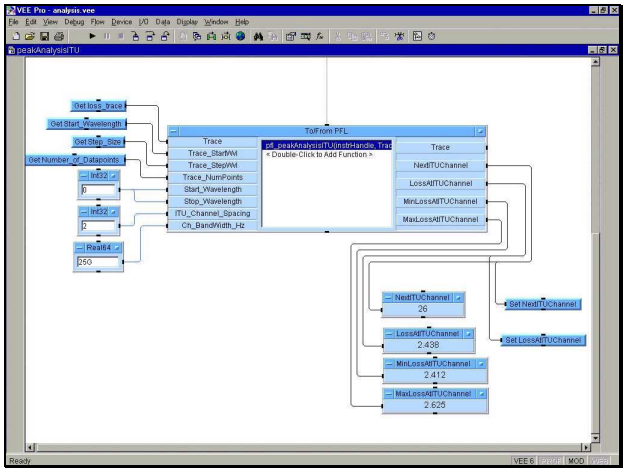
* Inputs parameters, "**Start_Wavelength**" and "**Stop_Wavelength**", specify the wavelength range approximated around ITU wavelength. For convenience, these parameters can be 0 if the whole trace range contains only one spectrum peak to be analyzed. Yet, it is highly recommended to specify the passband of target range to avoid noise in its calculation.

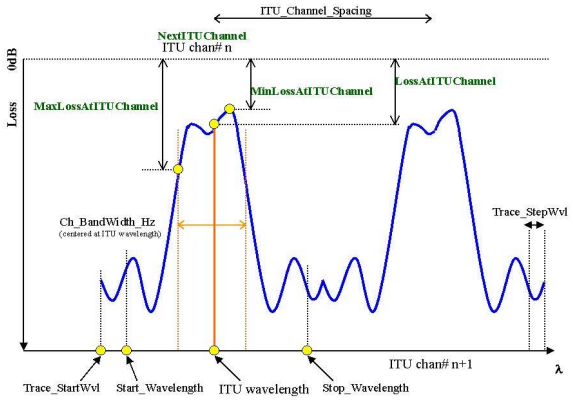


pfl_ituChannelWvl:

The "**pfl_ituChannelWvl**" function utilizes ITU information by selecting proper channel spacing and the ITU channel number. An output "**Wavelength**" is an ITU channel wavelength specific to its ITU channel number. The channel spacing is an enumerate type defined as follow:

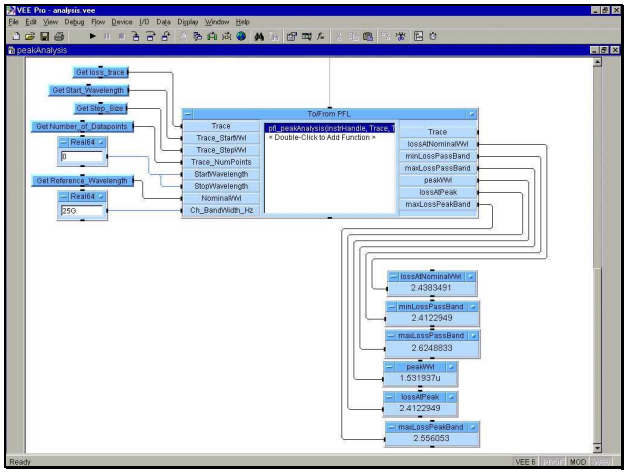
0 – 25 GHz	5 – 400 GHz
1 – 50 GHz	6 – 500 GHz
2 – 100 GHz	7 – 600 GHz
3 – 200 GHz	8 – 1000 GHz
4 – 300 GHz	

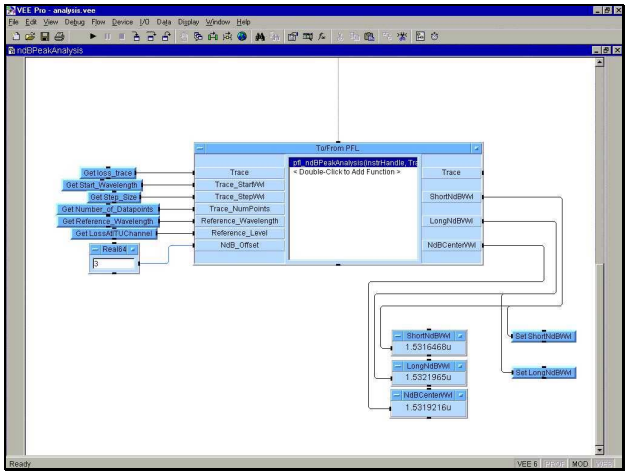


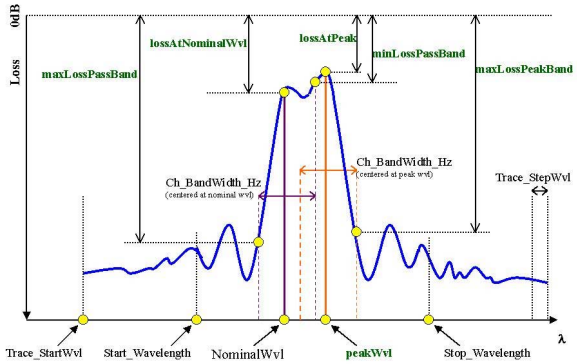


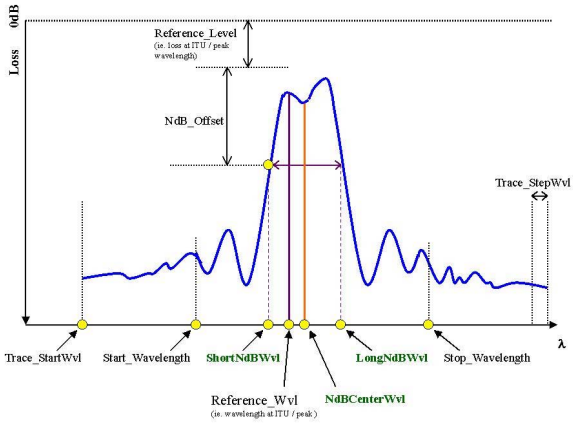
pfl_peakAnalysisITU:

The "**pfl_peakAnalysisITU**" function calculates the loss spectrum information according to its given ITU specification. "**NextITUChannel**" is an ITU channel number of the trace. The loss at the ITU wavelength, "**LossAtITUChannel**", the wavelengths at minimum / maximum losses, "**MinLossAtITUChannel**" and "**MaxLossAt ITUChannel**", are all searched. Minimum loss means that the device has a peak point within the passband centered at ITU wavelength. It is most likely to be different from the loss at the ITU channel wavelength due to interference and other device peak distortion effects in the optical transmission.









pfl_peakAnalysis:

The "**pfl_peakAnalysis**" function searches peak information at nominal wavelengths. Unlike the "**pfl_peakAnalysisITU**" function, "**pfl_peakAnalysis**" is based on the nominal wavelength, which can be defined as any wavelength within the trace.

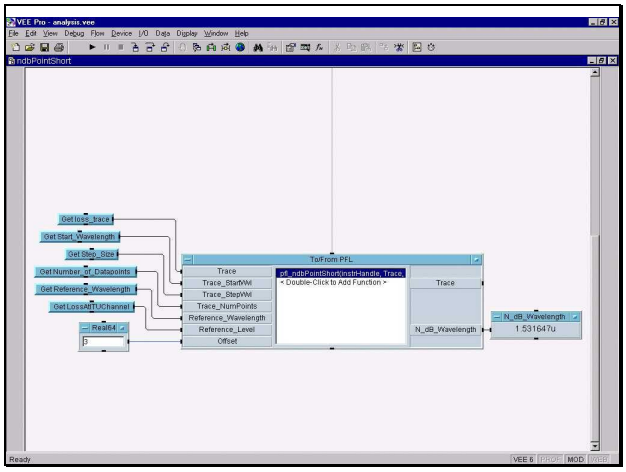
"**peakWvl**" and "**lossAtPeak**" are the spectrum peak wavelength and loss information, within the pre-defined wavelength range of "**Start_Wavelength**" and "**Stop_Wavelength**", which can be "**0**" (32 bit integer) for the whole trace.

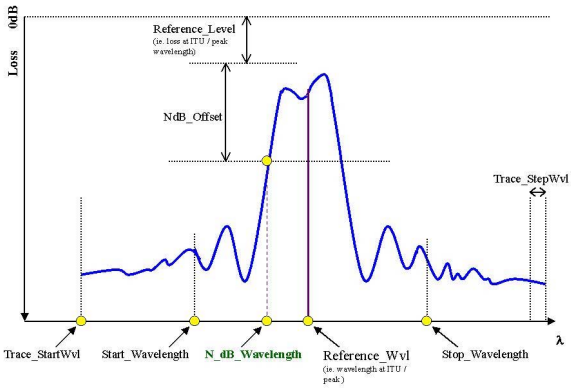
Two maximum losses occurring in the bandwidth centered at two different wavelengths, nominal and peak wavelengths, are calculated along with the minimum loss of passband of nominal wavelength.

pfl_ndBPeakAnalysis:

The device channel bandwidth at "**n dB**" down from either ITU wavelength or peak wavelength can be analyzed using the "**pfl_ndBPeakAnalysis**" function. "**n dB**" can be specified according to its test requirement such as 1, 3... dB. The difference between "**LongNdBWvl**" and "**ShortNdBWvl**" is the bandwidth of the spectrum.

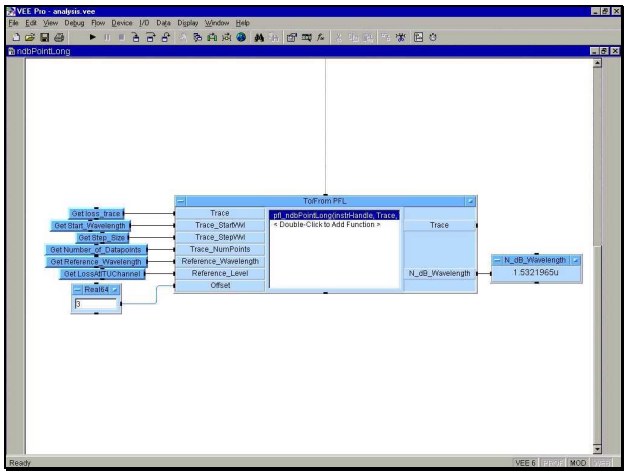
The "**Reference_Level**" input, can be either loss at peak wavelength or loss at ITU wavelength depending on the user specifications. From this reference loss, offset is added to its calculation.

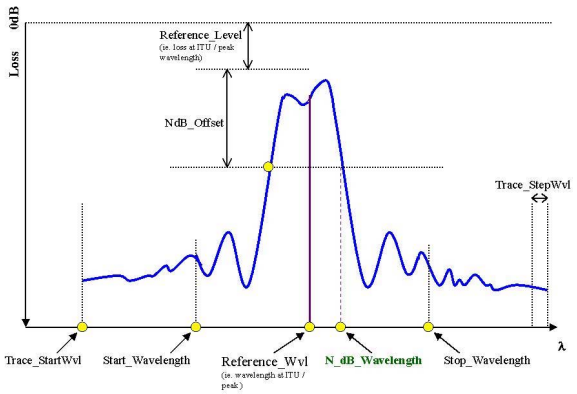




pfl_ndbPointShort:

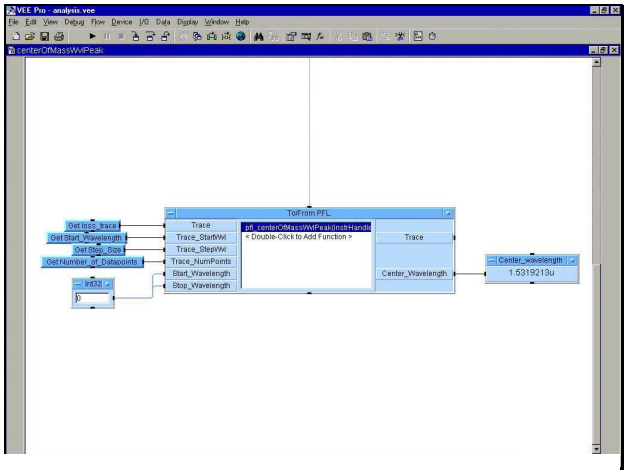
The "**pfl_ndbPointShort**" function is similar to the previously described "**pfl_ndBPeakAnalysis**" function, except that it only searches **for shorter** wavelengths from "**Reference_Wvl**".

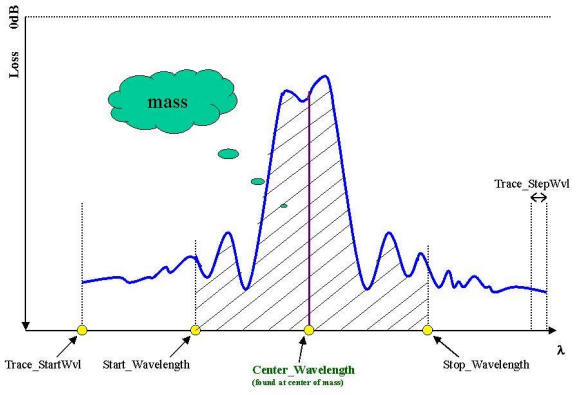




pfl_ndbPointLong:

The "**pfl_ndbPointLong**" function is similar to the previously described "**pfl_ndBPeakAnalysis**" function, except that it only searches **for longer** wavelengths from "**Reference_Wvl**".

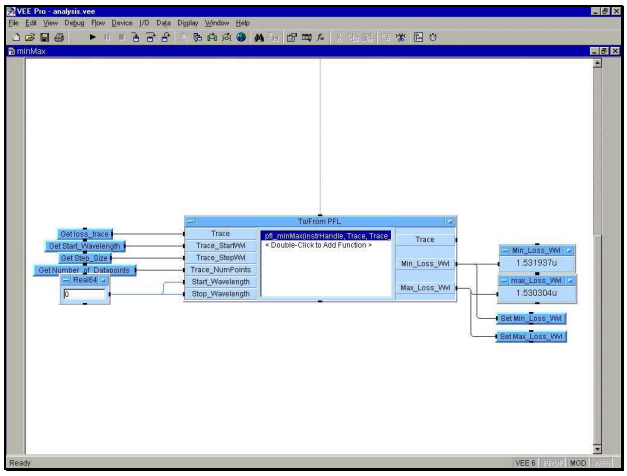


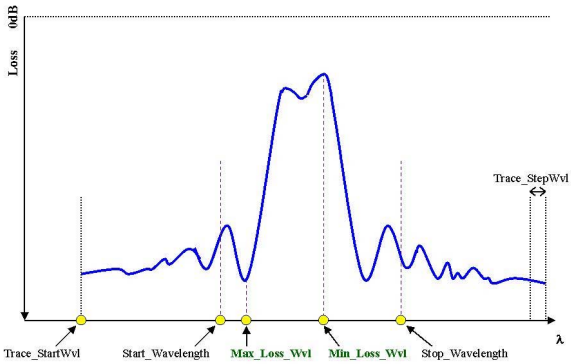


pfl_centerOfMassWvlPeak:

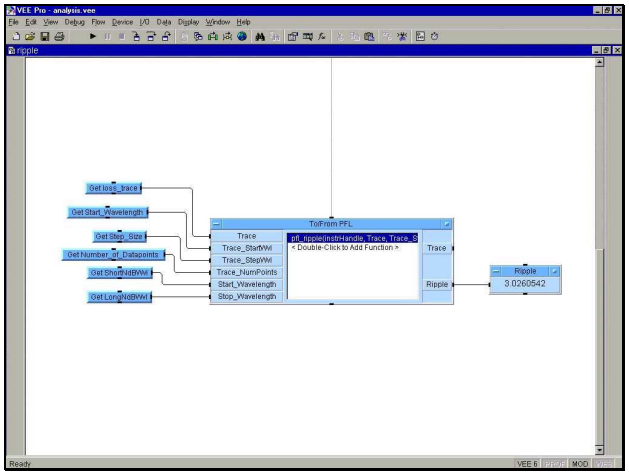
One way of finding the center wavelength is by calculating the total mass (volume) of spectrum between the user defined start and stop wavelength and then

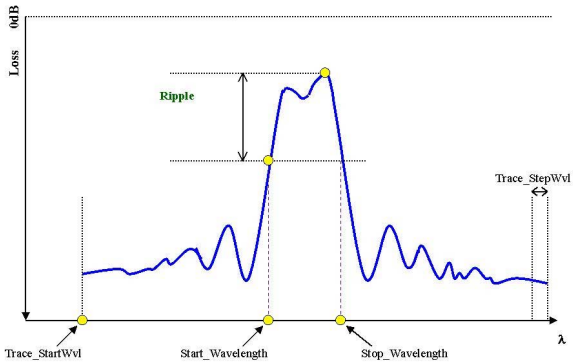
calculating the middle. "**pfl_centerOfMassWvlPeak**" takes this approach.
For example, the center of mass of a circle is a very center point of a circle. If one defines left coordinate of circle as (0,-1), right as (0,1), top as (1,0) and bottom as (-1,0), then the center coordinate is (0,0).



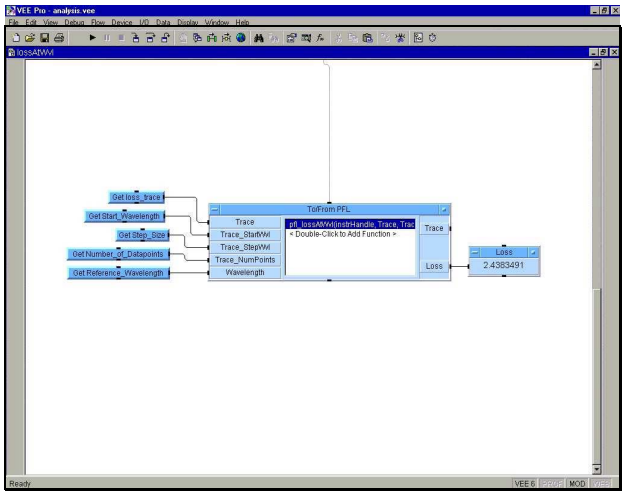


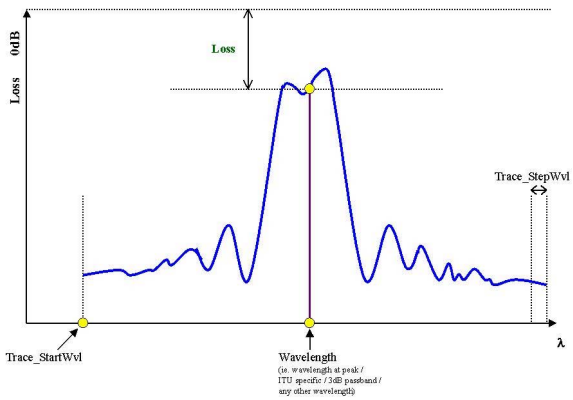
pfl_minMax:
A useful way to analyze ripple is by calculating maximum variation in insertion loss over operating wavelength range for unpolarized light. "**pfl_minMax**" allows users to find the minimum loss (peak) and the maximum loss (bottom) of the predefined wavelength range.
Instead of providing ripple value, this function generates only wavelength where maximum and minimum loss occurs.





pfl_ripple:
The "**pfl_ripple**" function is used to find ripple over operating wavelength range. Output parameter, ripple, is returned in dB.





pfl_lossAtWvl:
Any loss within the spectrum can be found using "**pfl_lossAtWvl**" by specifying wavelength. The linear interpolation calculation is used within the function at the wavelength outside of step size.

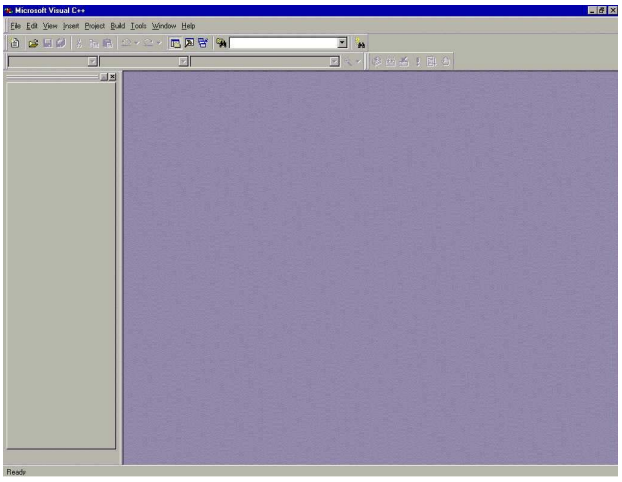
Module Summary
This completes the programming instruction for trace analysis function. In this module you have learned how to use trace analysis functions provided by the PFL, such as:
losses at ITU / peak of the spectrum
n dB bandwidth
wavelength at minimum / maximum loss
ripple

(ANSI C, API library)

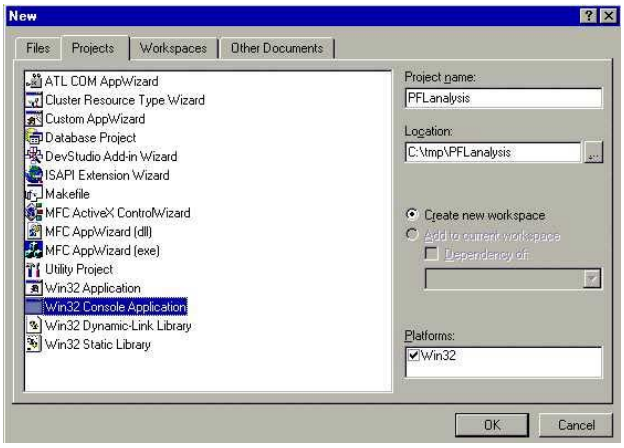
Configuration & Initialization

To identify the functional differences in the PFL, the C++ API offers four header files:
"stdpfl.h"
"pflmeasurement.h"
"pfltransformation.h"
"pflanalysis.h".

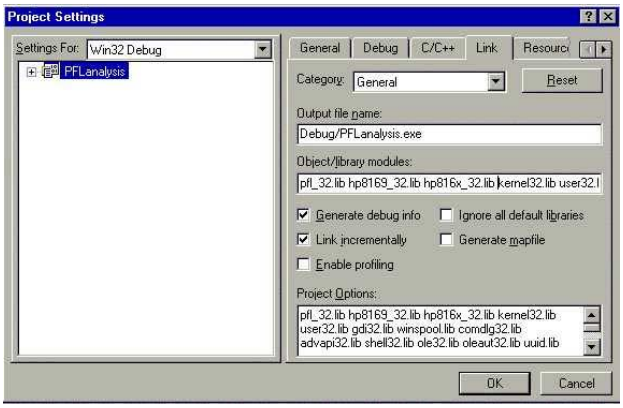
The functions provided by the PFL completely replace equivalent functions in the Agilent 816x and 8169 API by hiding the implementation details from users. With such information hidden, the programmer can focus more on the application itself in major implementations.
"**stdpfl.h**" includes the essential initialization, error handling, and version related interface.
"**pflmeasurement.h**" is a measurement related interface, such as lambda scan, real time sweep, and PDL related measurement using the Mueller method. It adds extensive capability for measurement of passive optical components to the test system.
"**pfltransformation.h**" is not only the calculus involved in IL and PDL, but also includes the accuracy enhancement algorithms which account for the instrument specific properties.
"**pflanalysis.h**" defines calculus of today's most demanding spectrum loss parameters such as bandwidth, isolation, ripple, crosstalk, and center wavelength.
Microsoft Visual C++ is the programming environment used to describe and demonstrate the PFL C++ API in a clear, visual, top-down manner. By first considering measurement structure on a high level using the PFL without concern for the implementation details, the user obtains a powerful tool that simplifies the process of handling data and naturally extends the concept of measurement.
The setup described in the following steps shows users how they must configure the MS VC++ application prior to compilation and execution. This guide touches briefly on the configuration that links the PFL and the PnP driver for 816x and 8169.



Step 1: Start the MS VC++ 6.0 environment and you should see the screen shown above.



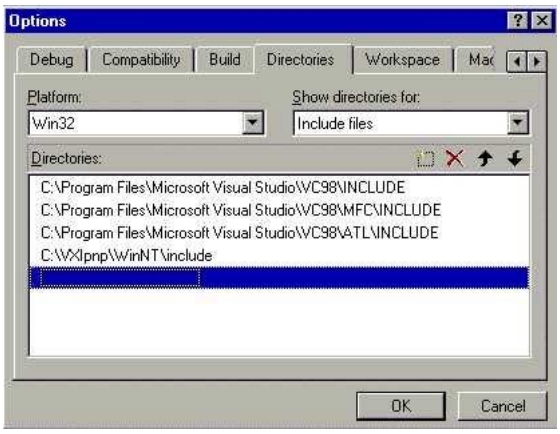
Step 2: To create a new working environment, select **[File]->[New]** in the start up screen. This prompts the above window to appear asking for the application that the programmer intends to use for development. Choose "**Win32 Console Application**" and enter the project name. The following window "**Win32 Console Application – Step 1 of 1**" should be checked with "**An empty project**" (depending on how to implement the PFL in the software, the definition of application might change).



Step 3: To link other libraries in MS VC++ 6.0:
Select [**Project**]-> [**Settings**] to open the **Project Settings** window
Enter the following libraries (used by the PFL and PnP) in the **Object/Library modules:** field.

"pfl_32.lib"
"hp816x_32.lib"
"hp8169_32.lib"

Select [**Tools**] -> [**Options**] to prompt the **Options** window.
Select the **[Directories]** tab
Enter the path (either by typing directly or browsing) of the following files in the **[Show directories for**:] field:
"Include files" – header folder (*.h for PFL & PnP)
"Library files" – library folder (*.lib for PFL & PnP)
"Source files" – dll folder (*.dll for PFL & PnP)



Step 4: You want to define PFL and PnP functions within the program:

The setup for MS VC++ 6.0 should be configured and it is ready to write the source code that endures full capability of instrument control and post data transaction simply by calling build in function of the PFL Here is the sample source code, based on the example provided by the PFL installation package. This section, along with configuration of working environment, introduces initialization and required variable / function declaration within every program that uses the PFL.

```
#include <stdio.h>                  /* standard input output library */
#include <stdlib.h>                 /* standard library */
#include <string.h>                 /* string operation library */
#include <conio.h>                  /* console I/O */

#include <stdpfl.h>                 /* definition for PFL init and close functions */
#include <pflmeasurement.h>         /* definition for PFL loss measurement functions */
#include <pfltransformation.h>      /* definition for PFL loss calculation functions */
#include <pflanalysis.h>            /* definition for PFL analysis functions */

ViSession IHandle;                  /* global variable definition for 816xA handle */
ViSession IpolHandle;               /* global variable definition for 8169A pol controller */

static void CheckError (ViStatus errStatus);   /* check measurement status */
static void OpenPFLSession ();      /* routine to establish communication with instruments */
static void ClosePFLSession ();     /*  routine to close communication with instruments */

#define NUM_POL_STATES 4            /* number of polarization states used by pdl measurement */


int main ()
{
  return 0;
}
```

Step 5: The definitions used for all measurement routines introduced in the PFL to establish and free the communications link with the instruments and perform specific measurements are programmed in **main ()**.

"**OpenPFLSession**" and "**ClosePFLSession**" are called at the beginning and the end of the program to initiate and free the instruments communication link. The instrument handles are stored in the variables, "**IHandle**", for 816x and "**IpolHandle**" for 8169A, which are defined as global variables.

This guide introduces three static functions, "**CheckError**", "**OpenPFLSession**", and "**ClosePFLSession**" to ease the readability of C++ source code and to demonstrate repeatable usage in different measurement program routines.

```
Void CheckError (ViStatus errStatus)
{
 ViInt32 inst_err,
                errSource;
 ViChar  err_message[256],
                inst_err_message[256];

 if (errStatus < VI_SUCCESS)
 {
        PflErrorInfo (errStatus, err_message, &errSource, &inst_err, inst_err_message);
        printf ("      Error : %lx, %s\n", errStatus, err_message);
        printf("       Source: %lx\n", errSource, err_message);
        printf(„       Inst  : %lx, %s\n", inst_err, inst_err_message);

        ClosePFLSession();

        printf("Press <ENTER> to continue");
        getchar();
        exit(errStatus);
 }
 return;
}
```

Step 6: The "**CheckError**" function works on an error status generated in the PFL function that could result from either hardware failure such as GPIB connection problems or software failure such as an out of range parameter setting. It is especially useful to debug the problem by seeing the error message transcribed by a PFL specific error number and its description.
An error message is generated on the **Console Screen** which awaits keyboard input for confirmation before halting the program.

```
void OpenPFLSession ()
{
 ViRsrc          defBus816xadress = "GPIB0::20::INSTR"     /* instrument address for 816x*/
 ViRsrc          defBus8169adress = "GPIB0::24::INSTR"     /* instrument address for 8169*/

 errStatus = PflInit816x (defBus816xadress, VI_FALSE, VI_TRUE, &IHandle);
 CheckError (errStatus);

 errStatus = PflInit8169 (defBus8169adress,VI_FALSE, VI_TRUE, &IpolHandle);
 CheckError (errStatus);

 errStatus = PflInit(IHandle, IpolHandle);
 CheckError (errStatus);

 return;
}
```

Step 7: The **Initialization** routine that opens the session involving the instruments and the PFL takes place in three steps:
"PflInit816x"
"PflInit8169"
"PflInit".
This routine assumes that the 816x Lightwave Measurement System and 8169A Polarization Controller are in the system setup.
Note: Although some measurements like insertion loss and real time don't require the 8169A polarization controller in the test setup, it is good practice to use this routine throughout all measurement algorithms. Where the PDL measurement is not required or the 8169A is NOT available in the system, the programmer MUST omit the line that initializes the 8169A. Otherwise, the "**CheckError**" function after the "**PflInit8169**" function generates an error message that fails to find or describe the instrument that causes the program to halt.
Note: Instrument addresses are arbitrarily set to address 20 for the 816x Lightwave Measurement System and address 24 for the 8169A Polarization Controller. These addresses must match with the hardware settings found in the instruments "**config menu**".

```
void ClosePFLSession ()
{
  PflSetTLSState (VI_FALSE);
  PflUnregisterMainframe (IHandle);
  PflClose816x (IHandle);
  PflClose8169 (IpolHandle);
  PflClose ();

  return;
}
```

Step 8: The **Close** routine closes any session involving the instruments and the PFL. It also includes the functionality to switch the tunable laser source's power output to "off" and unregisters the mainframe. This routine assumes that the 816x Lightwave Measurement System and 8169A Polarization Controller are in the system setup.

Note: As previously described, you should omit the "**PflClose8169**" routine if no polarization controller is used in the test system.

Module Summary

In this module you have learned how to configure and initialize the PFL C++ API in Microsoft Visual C++. (ANSI C, API library)

Insertion Loss Measurement:

Based on the stimulus response system, the insertion loss of the DUT is determined by a two step approach. However, the program can be written in such a way that the same functions and the same parameters can be used for both measurements.

The following diagram features line by line instructions on programming techniques using the PFL. All source code should be written in the main function, "**int main**()".

```
ViStatus errStatus;                     /* measurement status */
ViReal64 power,
                startWavelength,        /* start wavelength for lambda scan */
                stopWavelength,         /* stop wavelength of lambda swan */
                stepSize,               /* step size of lambda scan */
                dOffset;                /* wavelength offset if calibrated with gas cell */
ViInt32   opticalOutput,                /* optical output used for measurement */
                numberofScans,          /* number of scan depend on required device dynamic */
                PWMChannel;             /* power meter channel */
ViUInt32 numberofDatapoints,            /* number of data to be read from instrument */
                numberofValueArrays;    /*  number of channel detected in the system setup*/
ViUInt32 i, j;                          /* index used for reading & writing data */

char            filename[256] = "ilout.csv"; /* output file name to store loss data */
FILE            fp;                     / file pointer used to write output file */

ViReal64        wavelengthArray,        / wavelength data */
                powerArray,             / measured absolute power data */
                lossArray;              / calculated loss data */
```

Step 1: Several important variables are declared according to their data type. The data type must be specifically defined according to the input / output parameters of the PFL functions (for example, it is crucial to differentiate between signed or unsigned integers). The programmer can use whatever descriptive name he chooses to name variables. For convenience, all further descriptions will use the above variable names.

```
power = -5.0;                           /* TLS power is set to –5 dBm */
startWavelength = 1520.0e-9;            /* start wavelength for lambda scan is set to 1520nm */
stopWavelength = 1570.0e-9;             /* stop wavelength for lambda scan is set to 1570nm */
stepSize = 10e-12;                      /* step size for lambda scan is set to 10pm */
opticalOutput = pfl_LOWSEE;             /* low SSE optical output is used */
numberofScans = pfl_NO_OF_SCANS_1;      /* number of scan is set to 1 for low dynamic */
PWMChannel = 0;                         /* 1 power meter channel is used (0 index) */
dOffset = 0.0;                          /* wavelength calibrated offset value */

OpenPFLSession();                       /* call function to open PFL session */
```

Step 2: The initialization part of the program always involves parameter settings and session opening with the instruments. The parameters must be valid according to the option of Tunable Laser Source in the test system. For example, the wavelength range of 81640A, C-L band tunable laser source with low SEE output, is from 1520nm to 1640nm. Wavelength range outside of this designated wavelength will result in error.

The consideration of measurement time over accuracy can be easily analyzed by adjusting two parameters: number of scan and step size. For multiple channels like MUX / DeMUX, adjust a variable, PWMChannel, to availability of power meter channels, which are

simultaneously synchronized with a unique hardware triggering system of tunable laser source and mainframe via BNC cable. The instrument handle for an additional mainframe has also to be declared and initialized if used in the test system. Optimizing parameters by analyzing test results is simple programming when compared to lower level development where it is easy to get lost in a sea of details.

```
errStatus = PflPrepareMfLambdaScan   (power, opticalOutput, numberofScans, PWMChannel +1,
                                      startWavelength, stopWavelength, stepSize, pfl_SPEED_AUTO,
                                      &numberofDatapoints, &numberofValueArrays);
CheckError (errStatus);
```

Step 3: Three steps (step 3,5,6) are central to insertion loss measurement. These steps measure the absolute power level over wavelength.
"**PflPrepareMfLambdaScan**" sets parameters for the instrument that in return generates the number of data to be measured in one channel, and the number of channels available in the instrument.

```
wavelengthArray = (ViReal64 *)malloc(numberofDatapoints * sizeof (ViReal64));
powerArray = (ViReal64 *)malloc(numberofDatapoints * sizeof (ViReal64));
lossArray = (ViReal64 *)malloc(numberofDatapoints * sizeof (ViReal64));
```

Step 4: Based on the number of data points for one channel estimated by "**PflPrepareMfLambdaScan**", the memory size for wavelength array, power reading array and loss calculated array must be properly allocated to specific data types. Data read from the instruments will be stored in these parameters according to the name description.
The description of each line is:
ALLOCATE memory space of NUMBER OF DATA, each data has a SIZE OF 64 BITS REAL

```
errStatus = PflExecuteMfLambdaScan (wavelengthArray, dOffset);
CheckError (errStatus);
```

Step 5: "**PflExecuteMfLambdaScan**" takes properly allocated memory of wavelength data and wavelength calibrated offset as input. An offset is a wavelength offset of tunable laser source, usually less than one half of picometer, which can be determined with the "**PflMeasureWavelengthOffset**" function provided in the PFL using standard gas cells, such as Acetylene or Cyanide gas cells with NIST specification. The tunable laser source in conjunction with the power meter synchronized reading starts off the swept wavelength measurement.

```
errStatus = PflGetMfLambdaScanResult (PWMChannel, powerArray, wavelengthArray);
CheckError (errStatus);
```

Step 6: The memory buffer in each power meter stores absolute power readings over a measured wavelength's range until measurement is complete. To retrieve data from the power meter, the "**PflGetMfLambdaScanResult**" function is used.
Note: To measure multiple channels at the same time, "**PWMChannel**" has to be a variable along with the algorithm to handle a loop where it provides the counter to repeatedly execute this function until data from all channels is read. See the PFL sample programs, provided in the installation package, for the programming algorithm. This guide features instructions for first-time users to the PFL, therefore the number of channels used for the measurement is 1, as set in parameter initialization step 2.

```
errStatus = PflCalcLossFromPwr(numberofDatapoints, power, powerArray, lossArray);
CheckError (errStatus);
```

Step 7: Based on the absolute power reading (W), the loss data (dB) measurement is calculated by comparing the reference of a power level used by a tunable laser source (dBm).
Note: It is recommended to take a separate reference measurement by repeating the steps 3, 5, 6. Variables of two absolute power readings, reference and DUT, are passed to the "**PflCalcLossFromRef**" function to calculate the loss property of the device. The function also calculates connectivity loss such as fiber connector and/or spliced section.

```
fp = fopen (filename, "wt";
if (fp == NULL)
        printf ("unable to open file %s\n", filename);
else
{
```

```
        for (i = 0; i < numberofDatapoints; i++)
        {
                fprintf (fp, "%1.13lf,%1.12lf\n", wavelengthArray[i], lossArray[i]);
        }
}
fclose(fp);
```

Step 8: A data file, the name is a variable of filename, is created to save loss data. This is a standard C routine where it directs data output to the file with the "**fprintf**" command after setting file pointer, "**fp**", using the "**fopen"** command.

```
free (wavelengthArray);
free (powerArray);
free (lossArray);
```

Step 9: Free any array variable created in the program at the end of program.

```
ClosePFLSession();
```

Step 10: Closing part of the program involves switching off the laser power, unregistering instruments, and then closing the session. The routine is described in the PFL C++ API Configuration & initialization module of this guide.

Module Summary

You have completed the programming instruction for insertion loss measurement. In this module you have learned how to apply the functionality of the PFL to:
measure insertion loss using swept wavelength measurements for a DUT
calculate the loss property of the device
For enhanced usage of the function, please review the sample programs provided in the PFL installation package.

```
Int main ()
{
 ViStatus        errStatus;                  /* measurement status */
 ViReal64        power,
                 startWavelength,            /* start wavelength for lambda scan */
                 stopWavelength,             /* stop wavelength of lambda swan */
                 stepSize,                   /* step size of lambda scan */
                 dOffset;                    /* wavelength offset if calibrated with gas cell */
 ViInt32 opticalOutput,                      /* optical output used for measurement */
                 numberofScans,              /* number of scan depend on required device dynamic */
                 PWMChannel;                 /* power meter channel */
 ViUInt32        numberofDatapoints,         /* number of data to be read from instrument */
                 numberofValueArrays;        /*  number of channel detected in the system setup*/
 ViUInt32        i, j;                       /* index used for reading & writing data */

 char    filename[256] = "ilout.csv";        /* output file name to store loss data */
 FILE    fp;                                 / file pointer used to write output file */

 ViReal64 wavelengthArray,                   / wavelength data */
                 powerArray,                 / measured absolute power data */
                 lossArray;                  / calculated loss data */


 power = -5.0;                               /* TLS power is set to –5 dBm */
 startWavelength = 1520.0e-9;                /* start wavelength for lambda scan is set to 1520nm */
 stopWavelength = 1570.0e-9;                 /* stop wavelength for lambda scan is set to 1570nm */
 stepSize = 10e-12;                          /* step size for lambda scan is set to 10pm */
 opticalOutput = pfl_LOWSEE;                 /* low SSE optical output is used */
 numberofScans = pfl_NO_OF_SCANS_1;          /* number of scan is set to 1 for low dynamic */
 PWMChannel = 0;                             /* 1 power meter channel is used (0 index) */
 dOffset = 0.0;                              /* wavelength calibrated offset value */

 OpenPFLSession();                           /* call function to open PFL session */


 errStatus = PflPrepareMfLambdaScan (power, opticalOutput, numberofScans, PWMChannel +1,
                                     startWavelength, stopWavelength, stepSize, pfl_SPEED_AUTO,
                                     &numberofDatapoints, &numberofValueArrays);
```

```
  CheckError (errStatus);

  wavelengthArray = (ViReal64 *)malloc(numberofDatapoints * sizeof (ViReal64));
  powerArray = (ViReal64 *)malloc(numberofDatapoints * sizeof (ViReal64));
  lossArray = (ViReal64 *)malloc(numberofDatapoints * sizeof (ViReal64));

  errStatus = PflExecuteMfLambdaScan (wavelengthArray, dOffset);
  CheckError (errStatus);

  errStatus = PflGetMfLambdaScanResult (PWMChannel, powerArray, wavelengthArray);
  CheckError (errStatus);

  fp = fopen (filename, "wt";
  If (fp == NULL)
          Printf ("unable to open file %s\n", filename);
  Else
  {
          for (i = 0; i < numberofDatapoints; I++)
          {
                  fprintf (fp, "%1.13lf,%1.12lf\n", wavelengthArray[i], lossArray[i]);
          }
  }
  fclose(fp);

  free (wavelengthArray);
  free (powerArray);
  free (lossArray);

  ClosePFLSession();

  return 0;
}
```

(ANSI C, API library)

Polarization Dependent Loss Measurement:

The programming side of PDL measurement involves 5 steps:
initialization and variable declarations
polarizer angle adjustment
reference measurement
DUT measurement
close and deallocation of variables.

However, the complexity of measurement techniques and instrument control is, once again, hidden under the PFL to facilitate programmers and thereby reduce development time.

```
ViStatus errStatus;                     /* measurement status */
ViReal64 power,
         startWavelength,               /* start wavelength for lambda scan */
         stopWavelength,                /* stop wavelength of lambda swan */
         stepSize,                      /* step size of lambda scan */
         dOffset;                       /* wavelength offset if calibrated with gas cell */
ViInt32  opticalOutput,                 /* optical output used for measurement */
         numberofScans,                 /* number of scan depend on required device dynamic */
         PWMChannel;                    /* power meter channel */
ViUInt32 numberofDatapoints,            /* number of data to be read from instrument */
         numberofValueArrays;           /*  number of channel detected in the system setup*/
ViUInt32 i, j;                          /* index used for reading & writing data */

char     filename[256] = "pdlout.csv";  /* output file name to store loss data */
FILE     fp;                            / file pointer used to write output file */

ViReal64 wavelengthArray,               / wavelength data */
         powerArray,                    / measured absolute power data */
         lossArray;                     / calculated loss data */

ViPReal64 refPower[NUM_POL_STATES]={0,0,0,0}, /* measured ref data at different pol states*/
          dutPower[NUM_POL_STATES]={0,0,0,0}, /* measured DUT data at different pol states*/
          wavelengthArray = 0,          /* wavelength data */
          minLoss = 0,                  /* minimum loss occurrence within 4 SOP spectrums  */
          maxLoss = 0,                  /* maximum loss occurrence within 4 SOP spectrums */
          avgLoss = 0,                  /* average loss occurrence  within 4 SOP spectrums */
          pdl = 0;                      /* calculated PDL data after the measurement */

ViReal64 polAngle;                      /* maximum polarizer angle in pol controller */
ViUInt32 polState;                      /* counter to measure 4 pol states */
ViInt32  polStates[NUM_POL_STATES] = {pfl_PS_LH0, pfl_PS_LDP45, pfl_PS_LDN45, pfl_PS_RHC};
```

Step 1: The description with the variable declaration above lists the parameters required for PDL program. Unlike insertion loss, PDL is the loss measurements at four different polarization states that is required to define a 2 dimensional array for reference and DUT power readings, named as "**refPower**" and "**dutPower**". The "**refPower[0**]" variable stores reference data as linear horizontal. The order of polarization states is defined in the "**polStates**" array in order to rotate the polarized light to linear horizontal, linear diagonal plus 45, linear diagonal negative 45, and right hand circle in sequence.
The "**polAngle**" variable is the result of determining the polarizer position of the 8169A polarization controller at the maximum absolute power reading.

```
power = -10.0;                          /* TLS power is set to –10 dBm */
startWavelength = 1520.0e-9;            /* start wavelength for lambda scan is set to 1520nm */
stopWavelength = 1560.0e-9;             /* stop wavelength for lambda scan is set to 1560nm */
stepSize = 10e-12;                      /* step size for lambda scan is set to 10pm */
opticalOutput = pfl_LOWSSE;             /* low SSE optical output is used */
numberofScans = pfl_NO_OF_SCANS_1;      /* number of scan is set to 1 for low dynamic */
PWMChannels = 0;                        /* 1 power meter channel is used (0 index) */
dOffset = 0.0;                          /* wavelength calibrated offset value */

OpenPFLSession();                       /* call function to open PFL session */
```

Step 2: Variable initialization depends on the instrument setup and test requirement. For details on opening a PFL session, which initializes the instruments, see C++ API Configuration & Initialization section of this paper.

```
printf("Prepare the optical connections for the reference measurement!\n");
printf("Press <ENTER> to continue");
fflush();
getchar();

errStatus = PflFindMaxPolPosition (0, opticalOutput, power,
                                   (startWavelength+stopWavelength)*.5, &polAngle);
CheckError (errStatus);
```

Step 3: A prompt informs users of the need for a reference connection, by first temporarily stopping the execution flow and then waiting for keyboard input. This is needed when the program takes a top-down approach to PDL measurement where both

reference and DUT are measured consecutively. It is possible to implement the software to save the reference data in a file where it can be retrieved for repeated usage.
Optimum algorithm to find the polarizer angle of the polarization controller at maximum transmission power is implemented in the "**PflFindMaxPolPosition**" function. A center wavelength of configured start and stop wavelengths for maximum transmission is sufficient to measure PDL of wide wavelength range. This is because the PFL takes into account the post data error correction of PDL uncertainty at any wavelength outside of the polarization controller's designated wavelength range.

```
for(polState = 0; polState < NUM_POL_STATES; ++polState)
  {
        errStatus = PflPrepareMfLambdaScan (power, opticalOutput, numberofScans,
                                                PWMChannels +1, startWavelength, stopWavelength,
                                                stepSize, sweepSpeed, &numberofDatapoints,
                                                &numberofValueArrays);
        CheckError (errStatus);
```

Step 4: As described in the definition of the Mueller method, four absolute powers are measured at four different polarization states. A **For Loop** is appropriate to index to polarization states.
"**PflPrepareMfLambdaScan**" sets parameters of the instrument which in return generates number of data to be measured in one channel and number of channel available in the instrument

```
.       if(0 == polState)
                wavelengthArray = (ViReal64 *)malloc(numberofDatapoints *sizeof(ViReal64));

        refPower[polState] = (ViReal64 *)malloc(numberofDatapoints *sizeof(ViReal64));
```

Step 5: If the loop is being executed for the first time, then you must allocate memory for wavelength data with the size of number of data points, where each data point consists of 64 bits real. For all reference power data indexed at the polarization state, you must also allocate enough memory for reference data with the size of a number of data points, each consisting of 64 bits real.

```
        errStatus = PflExecuteMfLambdaPolScan(wavelengthArray, polStates[polState],
                                                polAngle, dOffset);
        CheckError (errStatus);
```

Step 6: The functionality of "**PflEexecuteMfLambdaPolScan**" is similar to that of "**PflExecuteMfLambdaScan**" (described in IL step 5 in C++ API) except that it has a polarization state and a polarizer angle for its additional parameters. A polarization state parameter is an **integer** used to define the polarization states and a polarizer angle parameter is the angle found in "**PflFindMaxPolPosition**". Adjust the wavelength offset if you need to do wavelength calibration for more accurate measurement.

```
        errStatus = PflGetMfLambdaScanResult(PWMChannels, refPower[polState], wavelengthArray);
        CheckError (errStatus);
  }
```

Step 7: The memory buffer in each power meter stores absolute power readings over a measured wavelength's range until measurement is complete. To retrieve data from the power meter, the "**PflGetMfLambdaScanResult**" function is used.
Note: To measure multiple channels at the same time, "**PWMChannel**" has to be a variable along with the algorithm to handle a loop where it provides the counter to repeatedly execute this function until data from all channels is read. See the PFL sample programs, provided in the installation package, for programming algorithm. This guide features instructions for first-time users to the PFL, therefore the number of channels used for the measurement is 1, as set in parameter initialization step 2.

```
printf("Prepare the optical connections for the DUT measurement!\n");
printf("Press <ENTER> to continue");
fflush();
getchar();
```

Step 8: The prompt lets users know the connection requirements of the DUT by temporarily stopping the execution flow and waiting for keyboard input.

```
for(polState = 0; polState < NUM_POL_STATES; ++polState)
  {
        errStatus = PflPrepareMfLambdaScan(power, opticalOutput, numberofScans,
                                                                PWMChannels + 1, startWavelength, stopWavelength,
                                                                stepSize, sweepSpeed, &numberofDatapoints,
                                                                &numberofValueArrays);
        CheckError (errStatus);

        dutPower[polState] = (ViReal64 *)malloc(numberofDatapoints *sizeof(ViReal64));
        if(!dutPower[polState]) goto Exit;

        errStatus = PflExecuteMfLambdaPolScan(wavelengthArray, polStates[polState],
                                                                        polAngle, dOffset);
        CheckError (errStatus);

        errStatus = PflGetMfLambdaScanResult(PWMChannels, dutPower[polState], wavelengthArray);
        CheckError (errStatus);
  }
```

Step 9: To measure the PDL of the DUT you can use the same routine that was used to take reference measurements as described in Steps 4 to 7, except that the variable to store data changes to "**dutPower**". Therefore, you can implement the source code with this part functionalized for ease of programming.

```
minLoss = (ViReal64 *)malloc(numberofDatapoints *sizeof(ViReal64));
maxLoss= (ViReal64 *)malloc(numberofDatapoints *sizeof(ViReal64));
avgLoss = (ViReal64 *)malloc(numberofDatapoints *sizeof(ViReal64));
pdl             = (ViReal64 *)malloc(numberofDatapoints *sizeof(ViReal64));
```

Step 10: Allocate four arrays to store:
minimum loss
maximum loss
average loss
polarized dependent loss

The technique to allocate memory is the same as before, and involves using the "**malloc**" function of C++.

```
errStatus = PflCalcPDLMueller8169A( numberofDatapoints, startWavelength, stepSize,
                                        refPower[0], refPower[1], refPower[2], refPower[3],
                                        dutPower[0], dutPower[1], dutPower[2], dutPower[3],
                                        minLoss, maxLoss, avgLoss, pdl);
CheckError (errStatus);
```

Step 11: The complex calculus and optical theory behind the Mueller method is hidden in the "**PflCalcPDLMueller8169A**" function. In addition to passing data for reference and DUT measurement to calculate PDL, this function also passes some measurement parameters to apply post data collection.

```
fp = fopen(filename,"wt");
if( fp == NULL)
        printf("unable to open file %s",filename);
else
  {
        fprintf(fp, "wvl[m],min[dB],max[dB],avg[dB],pdl[dB]\n");
        for (i = 0 ; i <  numberofDatapoints; i++)
          {
                fprintf(fp,"%1.13lf,%1.12lf,%1.12lf,%1.12lf,%1.12lf\n",
                     wavelengthArray[i], minLoss[i], maxLoss[i], avgLoss[i], pdl[i]);
          }
  }
fclose(fp);
```

Step 12: A data file (the name is a variable of filename) is created to save PDL data. This is a standard C routine where it directs data output to the file using the "**fprintf**" command after initially setting the file pointer, "**fp**", using the "**fopen"** command.

```
free (wavelengthArray);
free (minLoss);
free (maxLoss);
free (avgLoss);
free (pdl);

for(polState = 0; polState < NUM_POL_STATES; ++polState)
  {
        free(refPower[polState]);
        free(dutPower[polState]);
  }
```

Step 13: You must free any array variable created in the program at the end of program.

```
ClosePFLSession();
```

Step 14: Closing part of the program involves switching off the laser power, unregistering instruments, and closing the session. The routine is described in the Configuration & Initialization of the PFL C++ API section of this guide.

Module Summary

You have completed the programming instruction for PDL measurement. In this module you have learned how to apply the functionality to:

Measure PDL

Calculate the PDL value derived from reference and DUT data.

```
For enhanced usage of the function, please review the sample programs provided in the PFL installation package.Int main ()
{
 ViStatus        errStatus;                      /* measurement status */
 ViReal64        power,
                 startWavelength,                /* start wavelength for lambda scan */
                 stopWavelength,                 /* stop wavelength of lambda swan */
                 stepSize,                       /* step size of lambda scan */
                 dOffset;                        /* wavelength offset if calibrated with gas cell */
 ViInt32 opticalOutput,                          /* optical output used for measurement */
                 numberofScans,                  /* number of scan depend on required device dynamic */
                 PWMChannel;                     /* power meter channel */
 ViUInt32        numberofDatapoints,             /* number of data to be read from instrument */
                 numberofValueArrays;            /*  number of channel detected in the system setup*/
 ViUInt32        i, j;                           /* index used for reading & writing data */

 char    filename[256] = "pdlout.csv";           /* output file name to store loss data */
 FILE    fp;                                     / file pointer used to write output file */

 ViReal64 wavelengthArray,                       / wavelength data */
                 powerArray,                     / measured absolute power data */
                 lossArray;                      / calculated loss data */

 ViPReal64 refPower[NUM_POL_STATES]={0,0,0,0}, /* ref data at different pol states*/
                 dutPower[NUM_POL_STATES] = {0,0,0,0}, /* DUT data at different pol states*/
                 wavelengthArray = 0,            /* wavelength data */
                 minLoss = 0,                    /* minimum loss occurrence within 4 SOP spectrums  */
                 maxLoss = 0,                    /* maximum loss occurrence within 4 SOP spectrums */
                 avgLoss = 0,                    /* average loss occurrence  within 4 SOP spectrums */
                 pdl = 0;                  /* calculated PDL data after the measurement */

 ViReal64        polAngle;                       /* maximum polarizer angle in pol controller */
 ViUInt32        polState;                       /* counter to measure 4 pol states */
 ViInt32 polStates[NUM_POL_STATES] = {pfl_PS_LH0, pfl_PS_LDP45, pfl_PS_LDN45, pfl_PS_RHC};

 power = -10.0;                                  /* TLS power is set to –10 dBm */
 startWavelength = 1520.0e-9;                    /* start wavelength for lambda scan is set to 1520nm */
 stopWavelength = 1560.0e-9;                     /* stop wavelength for lambda scan is set to 1560nm */
 stepSize = 10e-12;                              /* step size for lambda scan is set to 10pm */
 opticalOutput = pfl_LOWSSE;                     /* low SSE optical output is used */
 numberofScans = pfl_NO_OF_SCANS_1;              /* number of scan is set to 1 for low dynamic */
 PWMChannels = 0;                                /* 1 power meter channel is used (0 index) */
 dOffset = 0.0;                                  /* wavelength calibrated offset value */
```

```
OpenPFLSession();                                   /* call function to open PFL session */

printf("Prepare the optical connections for the reference measurement!\n");
printf("Press <ENTER> to continue");
getchar();

errStatus = PflFindMaxPolPosition (0, opticalOutput, power,
                                                                    (startWavelength+stopWavelength)*.5, &polAngle);
CheckError (errStatus);

for(polState = 0; polState < NUM_POL_STATES; ++polState)
 {
        errStatus = PflPrepareMfLambdaScan (power, opticalOutput, numberofScans,
                                                                    PWMChannels +1, StartWavelength, stopWavelength,
                                                                    stepSize, sweepSpeed, &numberofDatapoints,
                                                                    &numberofValueArrays);
        CheckError (errStatus);

        if(0 == polState)
                wavelengthArray = (ViReal64 *)malloc(numberofDatapoints *sizeof(ViReal64));
        refPower[polState] = (ViReal64 *)malloc(numberofDatapoints *sizeof(ViReal64));

        errStatus = PflExecuteMfLambdaPolScan(wavelengthArray, polStates[polState],
                                                                    polAngle, dOffset);
        CheckError (errStatus);

        errStatus = PflGetMfLambdaScanResult(PWMChannels, refPower[polState], wavelengthArray);
        CheckError (errStatus);
 }

printf("Prepare the optical connections for the DUT measurement!\n");
printf("Press <ENTER> to continue");
getchar();

for(polState = 0; polState < NUM_POL_STATES; ++polState)
 {
        errStatus = PflPrepareMfLambdaScan(power, opticalOutput, numberofScans,
                                                                    PWMChannels + 1, startWavelength, stopWavelength,
                                                                    stepSize, sweepSpeed, &numberofDatapoints,
                                                                    &numberofValueArrays);
        CheckError (errStatus);

        dutPower[polState] = (ViReal64 *)malloc(numberofDatapoints *sizeof(ViReal64));
        if(!dutPower[polState]) goto Exit;

        errStatus = PflExecuteMfLambdaPolScan(wavelengthArray, polStates[polState],
                                                                    polAngle, dOffset);
        CheckError (errStatus);

        errStatus = PflGetMfLambdaScanResult(PWMChannels, dutPower[polState], wavelengthArray);
        CheckError (errStatus);
 }
minLoss         = (ViReal64 *)malloc(numberofDatapoints *sizeof(ViReal64));
maxLoss         = (ViReal64 *)malloc(numberofDatapoints *sizeof(ViReal64));
avgLoss         = (ViReal64 *)malloc(numberofDatapoints *sizeof(ViReal64));
pdl             = (ViReal64 *)malloc(numberofDatapoints *sizeof(ViReal64));

errStatus = PflCalcPDLMueller8169A( numberofDatapoints, startWavelength, stepSize,
                                                    refPower[0], refPower[1], refPower[2], refPower[3],
                                                    dutPower[0], dutPower[1], dutPower[2], dutPower[3],
                                                    minLoss, maxLoss, avgLoss, pdl);
CheckError (errStatus);

fp = fopen(filename,"wt");
if( fp == NULL)
        printf("unable to open file %s",filename);
else
 {
        fprintf(fp, "wvl[m],min[dB],max[dB],avg[dB],pdl[dB]\n");
```

```
            for (i = 0 ; i <  numberofDatapoints; i++)
              {
                    fprintf(fp,"%1.13lf,%1.12lf,%1.12lf,%1.12lf,%1.12lf\n",
                          wavelengthArray[i], minLoss[i], maxLoss[i], avgLoss[i], pdl[i]);
              }
   }
  fclose(fp);

  free (wavelengthArray);
  free (minLoss);
  free (maxLoss);
  free (avgLoss);
  free (pdl);

  for(polState = 0; polState < NUM_POL_STATES; ++polState)
   {
            free(refPower[polState]);
            free(dutPower[polState]);
   }

  ClosePFLSession();

  return 0;
 }
```

(ANSI C, API library)

Real Time Measurement:

For real time measurement, the PFL introduces what is probably the easiest program for one of the library's most powerful tools. The same parameters as for insertion loss measurement are used for continuously monitoring absolute power readings. Triggering and data retrieving timing are improved in the software to maximize the hardware capability of the test system.

```
ViStatus errStatus;                     /* measurement status */
ViReal64 power,
                startWavelength,        /* start wavelength for lambda scan */
                stopWavelength,         /* stop wavelength of lambda swan */
                stepSize,               /* step size of lambda scan */
                pmRange;                /* power meter range set by factor of 10 (ie. 0, -10,...) */
ViInt32         opticalOutput,          /* optical output used for measurement */
                PWMChannel;             /* power meter channel */
ViUInt32 numberofDatapoints,            /* number of data to be read from instrument */
                numberofValueArrays;    /* number of channel detected in the system setup*/
ViUInt32 i, j;                          /* index used for writing data */

ViReal64        wavelengthArray,        / wavelength data */
                powerArray;             / measured absolute power data */
```

Step 1: Parameters used for real time measurement are declared. Again, the data type of each variable must be specific to the definition of the PFL described in the header file.

```
power = -10.0;                          /* TLS power is set to –10 dBm */
startWavelength = 1540.0e-9;            /* start wavelength for lambda scan is set to 1520nm */
stopWavelength = 1560.0e-9;             /* stop wavelength for lambda scan is set to 1560nm */
stepSize = 10e-12;                      /* step size for lambda scan is set to 10pm */
opticalOutput = pfl_LOWSSE;             /* low SSE optical output is used */
PWMChannels = 0;                        /* 1 power meter channel is used (0 index) */
pmRange = -10;                          /* power meter range to –10dB

OpenPFLSession();                       /* call function to open PFL session */
```

Step 2: The power meter range, "**pmRange**" is selectable with the **Real Time** function. The auto ranging capability of the power meter is switched to a manual setting in order to improve the speed of measurement. The power meter range is set based on the device loss properties. Note: The power meter can read power values of –40dBm to +3dBm from the selected range in a single sweep. For example, setting the power meter (with an exception of fast power meter, 81636B & 81637B) range to –10dBm enables optical power readings between –50dBm and –7dBm.

```
errStatus = PflPrepareFastSweep (power, optical_output, PWMChannels,
                                 start_wavelength, stop_wavelength, step_size,
                                 pmRange, & numberofDatapoints, & numberofValueArrays);
CheckError (errStatus);
```

Step 3: Defined parameters are applied to the instrument by calling "**PflPrepareFastSweep**", which returns the number of data to be measured in one channel and the number of channel available in the instrument.

```
wavelengthArray  = (ViReal64 *)malloc(datapoints *sizeof(ViReal64));
powerArray       = (ViReal64 *)malloc(datapoints *sizeof(ViReal64));
```

Step 4: The wavelength and transmission power data are stored in arrays, while memories are allocated in this C standard routine using the **malloc** function.

```
for (i=0; i<10;i++)
{
        errStatus = PflDoFastSweep();
        for (j=0; j< numberofValueArrays; j++)
        {
                errStatus = PflGetFastSweepResult(j, powerArray, wavelengthArray);
        }
}
```

Step 5: This is the simplest form of repeated routine for executing real time sweep over wavelength. The outer loop specifies the number of repeated measurements to be executed. Although the above instruction is limited to

10 loops, the real time function allows infinite repetition if required. The inner loop is used to retrieve data from multiple channels. A variable, "**numberofValueArray**", is estimated by the previous function, "**PflPrepareFastSweep**", to count all enabled channels. If only one channel is required, the programmer can even omit the inner FOR loop that indexes the channel number.

Note: Stopping the program within the execution of real time measurement might cause the system to halt due to a precise timing mechanism that enables maximum measurement speed. It is highly recommended that the programmer implements the source code in such a way that it always executes the next two steps of the instruction.

```
PflCloseFastSweep();
```

Step 6: Real time measurement, also called fast sweep, has a special close routine to properly close the communication link with the instruments because the functionality of fast sweep is only possible by providing minimum allowable triggering timing by directly talking and listening the instrument.

```
free (wavelengthArray);
free (powerArray);

ClosePFLSession();
```

Step 7: At the end of the program, this function frees any array variables created in the program. Closing part of the program involves switching off laser power, unregistering instruments, and closing the session. The routine is described in the Configuration & Initialization of the PFL C++ API section of this guide.

Module Summary

You have completed the programming instruction for real time sweep measurement. In this module you have learned how to:

Perform a real time measurement using the PFL fast sweep functionality

Create a loop object that repeatedly executes the PFL function for real time measurement

For enhanced usage of the function, please review the sample programs provided in the PFL installation package.

```
int main ()
{
 ViStatus        errStatus;                      /* measurement status */
 ViReal64        power,
                 startWavelength,                /* start wavelength for lambda scan */
                 stopWavelength,                 /* stop wavelength of lambda swan */
                 stepSize,                       /* step size of lambda scan */
                 pmRange;                        /* power meter range set by factor of 10 (ie. 0, -10,...) */
 ViInt32 opticalOutput,                          /* optical output used for measurement */
                 PWMChannel;                     /* power meter channel */
 ViUInt32        numberofDatapoints,             /* number of data to be read from instrument */
                 numberofValueArrays;            /* number of channel detected in the system setup*/
 ViUInt32        i, j;                           /* index used for writing data */

 ViReal64        wavelengthArray,                / wavelength data */
                 powerArray;                     / measured absolute power data */

 power = -10.0;                                  /* TLS power is set to –10 dBm */
 startWavelength = 1540.0e-9;                    /* start wavelength for lambda scan is set to 1540nm */
 stopWavelength = 1560.0e-9;                     /* stop wavelength for lambda scan is set to 1560nm */
 stepSize = 10e-12;                              /* step size for lambda scan is set to 10pm */
 opticalOutput = pfl_LOWSSE;                     /* low SSE optical output is used */
 PWMChannels = 0;                                /* 1 power meter channel is used (0 index) */
 pmRange = -10;                                  /* power meter range to –10dB

 OpenPFLSession();                               /* call function to open PFL session */

 errStatus = PflPrepareFastSweep (power, optical_output, PWMChannels,
                                                  start_wavelength, stop_wavelength, step_size,
                                                  pmRange, & numberofDatapoints, & numberofValueArrays);
 CheckError (errStatus);

 wavelengthArray         = (ViReal64 *)malloc(datapoints *sizeof(ViReal64));
 powerArray              = (ViReal64 *)malloc(datapoints *sizeof(ViReal64));

 for (i=0; i<10;i++)
 {
        errStatus = PflDoFastSweep();
        for (j=0; j< numberofValueArrays; j++)
        {
                errStatus = PflGetFastSweepResult(j, powerArray, wavelengthArray);
        }
}

 PflCloseFastSweep();

 free (wavelengthArray);
 free (powerArray);

 ClosePFLSession();

 return 0;
}
```

(ANSI C, API library)

Trace Analysis:

Trace analysis functions are among the most powerful precompiled tools in the PFL. These functions enable programmers to implement software not only to measure the device but also to analyze the device characteristics based on its insertion loss data. Such device characterizations include among others:

loss at ITU / peak wavelength
n dB center wavelength
bandwidth
ripple
crosstalk and more

Based on the structure data type (which will be described in the next section), the individual analysis library functions included in the PFL enable programmers to justify measured data using just one function call. This has been achieved through the development of software that leverages the complex concepts behind the test and measurement of passive optical components. The programmer can study the diagrams provided in each description step in this module in order to understand the input and output parameters of the analysis function. This makes the subsequent programming task much easier. Some functions take input parameters that are the output of other functions. This parameter conjunction differentiates the device characterization based on its given input parameters. See "**PflNdBPeakAnalysis**" as an example.

```
Loss  = (double*)malloc(numSamples * sizeof(double));          /* allocate memory for loss */
trace = (double*)malloc(numSamples * sizeof(double)); /* allocate memory for trace */

ViStatus PFL_API PflCalcLossFromPwr (ViInt32  numSamples,
                                                    ViReal64  refPowerDbm,
                                                    ViReal64  dutPowerW[],
                                                    ViReal64  loss[]);

trace = loss;                                          /* store calculated loss data as a trace data for analysis */
```

The following section describes analysis functions, definition found in header file, to clarify individual parameters and functionality. Unlike other measurement routines, such as the IL and PDL algorithm, individual functions can be called separately and do not have to be in the order of this paper, but some output parameters can be passed from one function to another to observe different device characteristics. For example, reference wavelength for "**PflNdBPeakAnalysis**" can be either ITU or peak wavelength depending on the device manufacture test specification.
Before we begin with the description of trance analysis functions, it has to be noted that the programmer must be careful with the data used for each parameter. Trace data, the insertion loss calculated from the result of reference and device measurements using transformation function (**PflCalcLossFromPwr** or **PflCalcLossFromRef**), needs to be stored in a variable to be analyzed.

Step 1: This is a transformation function defined in the "**pfltransform.h**" header file to calculate insertion loss from two absolute powers, reference "**refPowerDbm**" and device "**dutPowerW**". A function is, again, introduced here in the analysis section of the paper for one reason; loss array in the parameter will be used by most analysis function calls.
A variable "**trace[]**" in the rest of this paper is loss array data, and a variable "**loss[]**" is calculated in "**PflCalcLossFromPwr**" if TLS output power level (dBm) is used as a reference, or in "**PFLCalcLossFromRef**" if reference measurement is also done prior to DUT measurement. An input variable, "**numSamples**", is size of loss trace that can be either calculated from the size of data or saved from an output variable of "**PflPrepareMfLambdaScan**".

```
ViReal64 PFL_API PflITUChannelWvl (EChSpacing spacing,
                                                    ViInt32    chNumber);
```

PflITUChannelWvl:
"**PflITUChannelWvl**" returns ITU wavelength according to input channel spacing "**spacing**" and target channel number "**chNumber**" of which programmers want to find the wavelength. Channel spacing used in this function is an enumerate type defined as below:

SPC_25_GHZ → 25 GHz SPC_400_GHZ → 400 GHz
SPC_50_GHZ → 50 GHz SPC_500_GHZ → 500 GHz
SPC_100_GHZ → 100 GHz SPC_600_GHZ → 600 GHz
SPC_200_GHZ → 200 GHz SPC_1000_GHZ → 1000 GHz

```
EStatus PFL_API PflPeakAnalysisITU (ViReal64    trace[],
                                                    TTraceInfo*  trcInfo,
                                                    TPeakParameterITU* peakParameter,
                                                    TPeakResult*  result);
```

PflPeakAnalysisITU:

A tested device characteristics can be confirmed with information based on ITU specification. These characteristics include loss at ITU wavelength, at nominal wavelength, and at spectrum peak wavelength. See the description of "**TPeakResult**" structure data type introduced at the beginning of this section.

User defined input parameters such as wavelength range and offset level, included in the member of "**trcInfo**" and "**peakParameter**", set test specification over traced data, "**trace[]**".

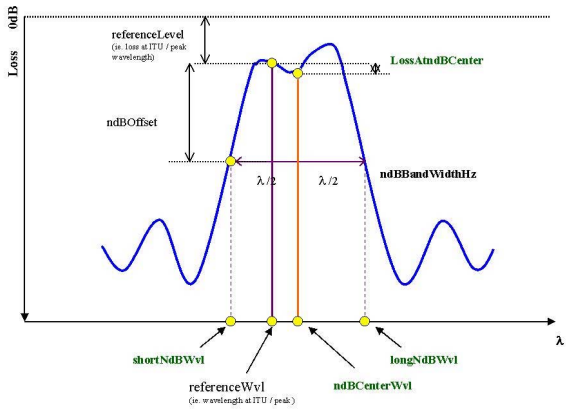
Note, status flag returned from "**PflPeakAnalysisITU**" is enumerate type that the trace analysis function, which generates spectrum characteristics, returns with more detailed information to be handled by implemented software to clarify for both users and programmers the right input parameter usage.

ST_NotFound
ST_OK
ST_InvalidPara
ST_NoTrace
ST_InvalidTraceInfo
ST_NoOutputPara
ST_InternalError
ST_LicenseNotAvailable
ST_LicenseNotGranted
ST_PflNotInitialized
ST_InvalidSession


```
EStatus PFL_API PflPeakAnalysis (ViReal64   trace[],
                                 TTraceInfo* trcInfo,
                                 TpeakParameter* peakParameter,
                                 TPeakResult* result);
```

PflPeakAnalysis: "**PflPeakAnalysis**" is similar to "**PflPeakAnalysisITU**" except it takes a data structure "**TPeakParameter**" instead of "**TPeakParameterITU**" to analyze loss spectrum, "**trace[]**", with user defined specification instead of ITU.

```
EStatus PFL_API PflNdBPeakAnalysis (ViReal64    trace[],
                                    TTraceInfo* trcInfo,
                                    ViReal64    referenceWvl,
                                    ViReal64    referenceLevel,
                                    ViReal64    ndBOffset,
                                    TNdBPeakResult* result);
```



PflNdBPeakAnalysis:

"**PflNdBPeakAnalysis**" takes offset levels, reference and offset, as shown in the diagram and returns spectrum information of center wavelength, "**ndBCenterWvl**", and losses, "**lossAtndBCenter**" (member of structure data type "**TndBPeakResult**") at any offset level depending on the device test requirement.

Center wavelength at n dB offset is where spectrum starts at offset, "**shortNdBWvl**", and ends at offset, "**longNdBWvl**", are divided exactly in the middle. In conjunction to such amplitude and wavelength information, spectrum bandwidth at the offset level, "**ndBBandWidthHz**", is calculated.

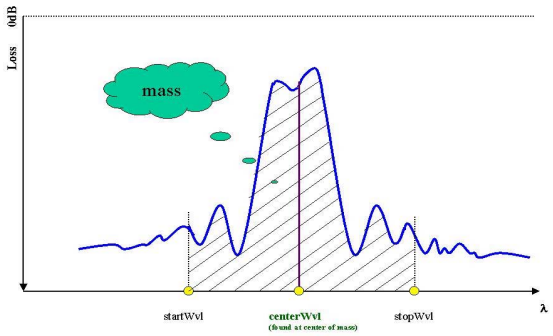
```
EStatus PFL_API PflNdBPointShort (ViReal64  trace[],
                                  TTraceInfo* trcInfo,
                                  TPoint*     refPoint,
                                  ViReal64    offset,
                                  TPoint*     ndBPoint);
```

PflNdBPointShort: To simplify the previous analysis function, "**PflNdBPointShort**" returns only the coordinate at spectrum where offset level **starts**. The coordinate is expressed in a struct, "**TPoint**" based on x (wavelength) and y (loss) in the graph.

```
EStatus PFL_API PflNdBPointLong (ViReal64  trace[],
                                 TTraceInfo* trcInfo,
                                 TPoint*     refPoint,
                                 ViReal64    offset,
                                 TPoint*     ndBPoint);
```

PflNdBPointLong: To simplify the previous analysis function, "**PflNdBPointLong**" returns only the coordinate at the spectrum where offset level **ends**. The coordinate is expressed in a struct, "**TPoint**" based on x (wavelength) and y (loss) in the graph.

```
EStatus PFL_API PflCenterOfMassWvlPeak (ViReal64  trace[],
                                        TTraceInfo*  trcInfo,
                                        ViReal64     startWvl,
                                        ViReal64     stopWvl,
                                        ViReal64*    centerWvl);
```



PflCenterOfMassWvlPeak:

"**PflCenterOfMassWvlPeak**" takes another approach to find the center wavelength, "**centerWvl**", by calculating the mass within the defined spectrum. Again, important parameters, such as "**startWvl**" and "**stopWvl**" narrow down the mass to calculate more precisely the location. An image, for example, of the center of a circle helps to understand the concept of this function.

```
EStatus PFL_API PflMinMax (ViReal64  trace[],
                           TTraceInfo* trcInfo,
                           ViReal64    startWvl,
                           ViReal64    stopWvl,
                           TPoint*     minPoint,
                           TPoint*     maxPoint);
```

PflMinMax:

The coordinates at minimum loss and maximum loss occurrences are searched by calling "**PflMinMax**" within the predefined start wavelength, "**startWvl**", and stop wavelength, "**stopWvl**". The coordinate is stored in structure format, "**TPoint**" where wavelength is expressed in **m** (meter) and loss in **dB**.

```
EStatus PFL_API PflRipple (ViReal64  trace[],
                           TTraceInfo*  trcInfo,
                           ViReal64     startWvl,
                           ViReal64     stopWvl,
                           ViReal64*    ripple);
```

PflRipple:

The definition of ripple is the difference between minimum and maximum losses within the predefined wavelength. "**PflRipple**" can be used to find a "**ripple**" of the spectrum by selecting the wavelength range, "**startWvl**" and "**stopWvl**".

```
EStatus PFL_API PflLossAtWvl (ViReal64  trace[],
                              TtraceInfo*  trcInfo,
                              ViReal64     wvl,
                              ViReal64*    loss);
```

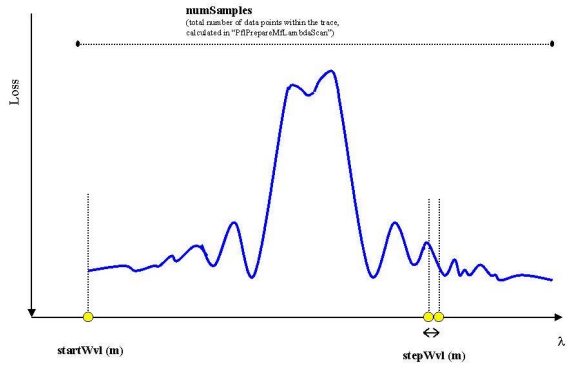
PflLossAtWvl:

"**PflLossAtWvl**" looks for loss, "**loss**", at specific wavelength, "**wvl**", within the spectrum. The linear interpolation calculation is used within the function at the wavelength outside of step size.
(ANSI C, API library)

Structure Design for Trace Analysis:

Using the library in C makes it easier for programmers to understand a function's input and output parameter by using **structure data type**. The diagrams shown in this section describe the meaning of each **member** in the **structure** (some languages refer to a structure as a ***record*** with ***fields*** instead of members).

TTraceInfo



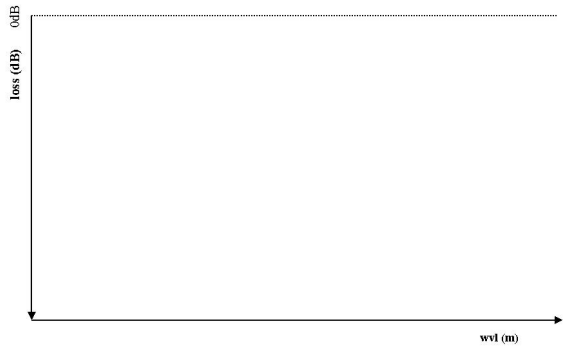
TTraceInfo

typedef struct

```
{
 ViInt32    numSamples;
 ViReal64   startWvl;
 ViReal64   stepWvl;
} TTraceInfo;
```

A structure "**TTraceInfo**" defines measured loss spectrum information. "**numSamples**" is a total number of data points within the trace generated by calling "**PflPrepareMfLambdaScan**" in the insertion loss routine. "**startWvl**" is a start wavelength of the trace and "**stepWvl**" is a step size used for measurement. Both take real numbers in units of "m" with a scientific notation (ie. for 1530nm start wavelength, use 1530.0e-9).

TPoint



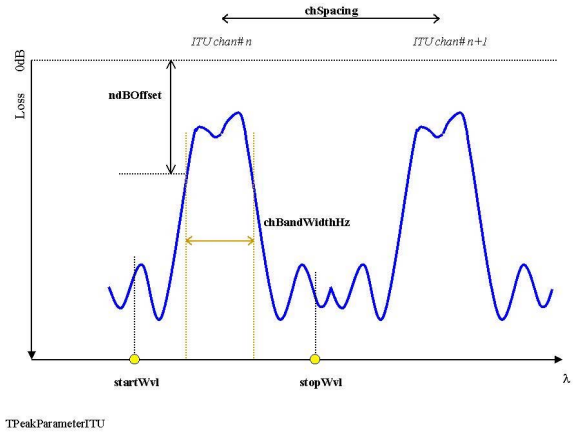
TPoint

typedef struct

{
 ViReal64 wvl;
 ViReal64 loss;
} *TPoint;*

A structure "**TPoint**" is used to define the coordinate information in the function. A member, "**wvl**", is wavelength on x-axis with unit of "**m**" (meter) and "**loss**" is loss data on y-axis with unit of "**dB**". Such trace point is useful in the parameter of "**PflMinMax**", for example, where minimum / maximum loss over wavelength can be passed as (wavelength, loss) format.

TPeakParameterITU



TPeakParameterITU

typedef struct

{
 ViReal64 startWvl;
 ViReal64 stopWvl;
 ViReal64 chBandWidthHz;
 ViReal64 ndBOffset;
EchSpacing chSpacing;
} TPeakParameterITU;

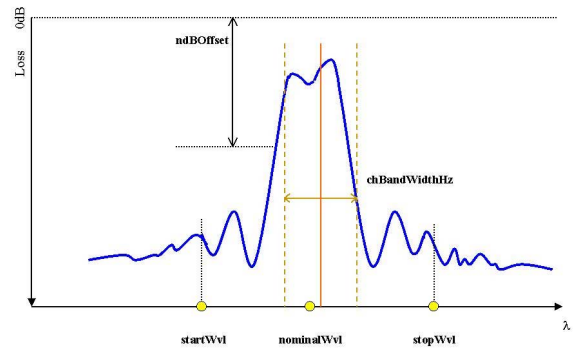
A structure "**TPeakParameterITU**" is a parameter for the trace analysis function "**pfl_peakAnalysisITU**" that requires trace information according to its ITU specification. Such wavelengths as "**startWvl**" and "**stopWvl**" differ from the member defined in the "**TTraceInfo**" structure, by the way that they define only at the target wavelength range of the peak spectrum to which the function applies for its calculation (see "**TPeakParameterITU**" function for detailed description). "**startWvl**" and "**stopWvl**" specify the wavelength range approximated around ITU wavelength. For convenience, these parameters can be 0 if the whole trace range contains just one device information, one peak. Yet, it is highly recommended to specify passband of target range to avoid including noise data from uninteresting wavelength ranges in this calculation.

A member "**chBandWidthHz**", specifies the peak spectrum bandwidth of the device under test with the unit of "**Hz**". For 50GHz bandwidth device, for example, use 50e9 to set a value.

Channel spacing is defined as the index in the structure as follows:

0 – 25 GHz	4 – 400 GHz
1 – 50 GHz	5 – 500 GHz
2 – 100 GHz	6 – 600 GHz
3 – 200 GHz	7 – 1000 GHz

TPeakParameter



TPeakParameter

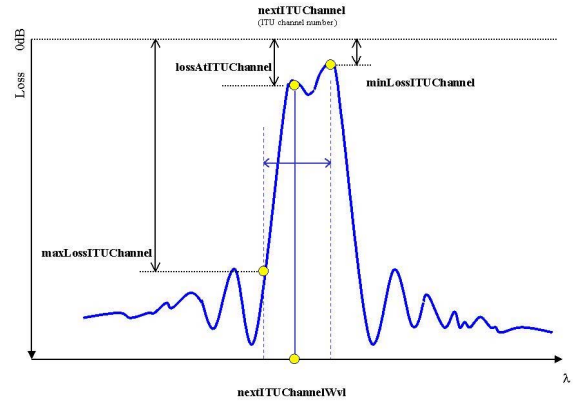
typedef struct

{
 ViReal64 startWvl;
 ViReal64 stopWvl;
 ViReal64 chBandWidthHz;
 ViReal64 ndBOffset;
 ViReal64 nominalWvl;
} *TPeakParameter;*

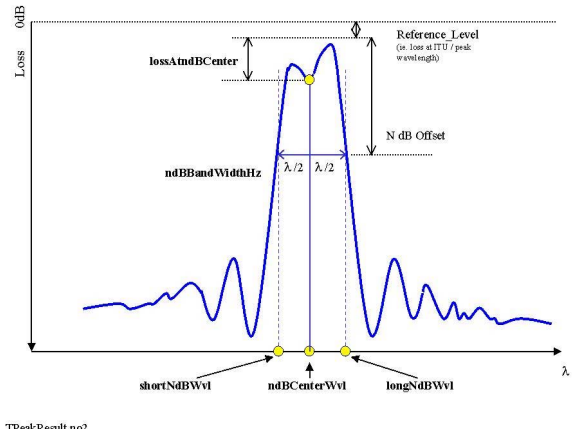
Instead of specifying ITU specification information as input, "**TPeakParameter**" can be used to find peak loss spectrum information. Nominal wavelength, "**nominalWvl**", can be either peak or ITU wavelength according to test requirements.

"**ndBOffset**" is an offset from reference level to find wavelength / loss information of the spectrum at specified level.
"**startWvl**" and "**stopWvl**" specify the wavelength range approximated around ITU wavelength. For convenience, these parameters can be 0 if whole trace range contains only one device information, one peak. Yet, it is highly recommended to specify passband of target range to avoid noise in its calculation.

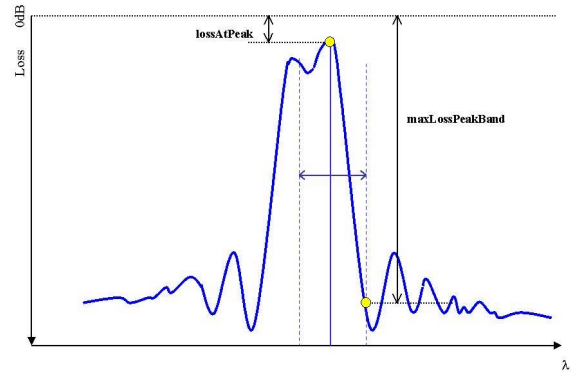
TPeakResult



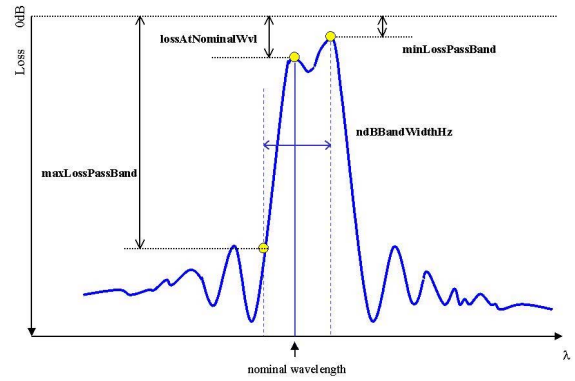
TPeakResult no1



TPeakResult no2



TPeakResult no3



TPeakResult no4

typedef struct

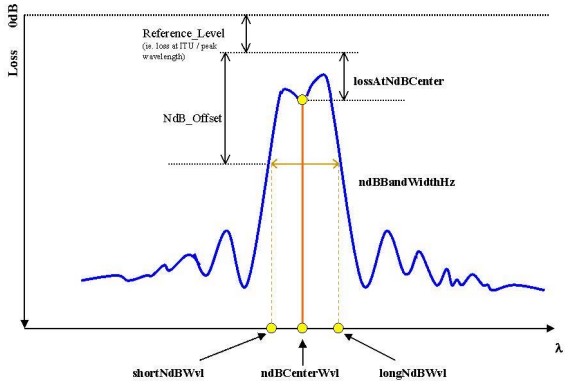
{
 ViInt32 nextITUChannel;
 ViReal64 nextITUChannelWvl;
 ViReal64 lossAtITUChannel;
 ViReal64 minLossITUChannel;
 ViReal64 maxLossITUChannel;
 ViReal64 ndBCenterWvl;
 ViReal64 lossAtndBCenter;
 ViReal64 shortNdBWvl;
 ViReal64 longNdBWvl;
 ViReal64 ndBBandWidthHz;
 ViReal64 peakWavelength;
 ViReal64 lossAtPeak;
 ViReal64 maxLossPeakBand;
 ViReal64 lossAtNominalWvl;
 ViReal64 minLossPassBand;
 ViReal64 maxLossPassBand;
} *TPeakResult;*

The results of trace analysis function for peak loss spectrum are stored completely as one comprehensive structure, "**TPeakResult**" for both ITU specific data and peak spectrum specific data.
"**nextITUChannel**" is an ITU channel number found in the closest peak within defined wavelength range, specific ITU channel spacing and channel bandwidth.

Wavelength information at ITU channel is returned along with loss information at ITU channel, "**lossAtITUChannel**", minimum, "**minLossITUChannel**", and maximum, "**maxLossITUChannel**", losses within predefined spectrum bandwidth centered at ITU wavelength. (see "**TPeakResult** no1" diagram above) Analysis information at n dB offset from reference level are described in "**TPeakResult** no2" of the diagram above. "**ndBCenterWvl**" is a center wavelength of the spectrum bandwidth at n dB offset and a loss at that wavelength is "**lossAtndBCenter**". Spectrum intersection at offset level, both at short and long wavelength, are parameters of "**shortNdBWvl**" and "**longNdBWvl**" and such bandwidth is described as "**ndBBandWidthHz**".

Spectrum peak wavelength information, "**peakWavelength**", and maximum loss within the bandwidth centered at peak wavelength, "**maxLossPeakBand**", are shown in diagram "**TpeakResult** no3". Parameters referenced at nominal wavelength are shown in "**TPeakResult** no4".

TNdBPeakResult



TNdBPeakResult

typedef struct

{
 ViReal64 ndBCenterWvl;
 ViReal64 lossAtndBCenter;
 ViReal64 shortNdBWvl;
 ViReal64 longNdBWvl;
 ViReal64 ndBBandWidthHz;
} *TNdBPeakResult;*

See "**TPeakResult**" for the description of each member.

This page left intentionally blank

Agilent Technologies' Test and Measurement Support, Services, and Assistance
Agilent Technologies aims to maximize the value you receive, while minimizing your risk and problems. We strive to ensure that you get the test and measurement capabilities you paid for and obtain the support you need. Our extensive support resources and services can help you choose the right Agilent products for your applications and apply them successfully. Every instrument and system we sell has a global warranty. Support is available for at least five years beyond the production life of the product. Two concepts underlie Agilent's overall support policy: "Our Promise" and "Your Advantage."

Our Promise

Our Promise means your Agilent test and measurement equipment will meet its advertised performance and functionality. When you are choosing new equipment, we will help you with product information, including realistic performance specifications and practical recommendations from experienced test engineers. When you use Agilent equipment, we can verify that it works properly, help with product operation, and provide basic measurement assistance for the use of specified capabilities, at no extra cost upon request. Many self-help tools are available.

Your Advantage

Your Advantage means that Agilent offers a wide range of additional expert test and measurement services, which you can purchase according to your unique technical and business needs. Solve problems efficiently and gain a competitive edge by contracting with us for calibration, extra-cost upgrades, out-of-warranty repairs, and on-site education and training, as well as design, system integration, project management, and other professional engineering services. Experienced Agilent engineers and technicians worldwide can help you maximize your productivity, optimize the return on investment of your Agilent instruments and systems, and obtain dependable measurement accuracy for the life of those products.

By Internet, phone, or fax, get assistance with all your test & measurement needs

Online assistance:
www.agilent.com/comms/lightwave

Phone or Fax

United States:
(tel) 1 800 829 4444
(fax) 1 800 829 4433

Canada:
(tel) 1 877 894 4414
(fax) 1 800 746 4866

Europe:
(tel) +31 20 547 2111
(fax) +31 20 547 2190

Japan:
(tel) 120 421 345
(fax) 120 421 678

Latin America:
(tel) +55 11 4197 3600
(fax) +55 11 4197 3800

Australia:
(tel) 800 629 485
(fax) 800 142 134

Asia Pacific:
(tel) +852 800 930 871
(fax) +852 800 908 476

Related Agilent Literature
http://www.agilent.com/comms/octaccessories

[1] Photonic Foundation Library : Enhancing Swept Loss Measurement, Application Notes
p/n 5988-3622EN

[2] Agilent N4150A, N4151A Photonic Foundation Library Rev 1.0 Technical Specification, Max 2001
p/n 5980-1453E

[3] Optical Communication Measurement Division How to use VXI Plug and Play Driver with Agilent VEE, C/C++, Visual Basic, LabVIEW, and LabWindows/CVI
p/n 5980-2790N

[4] PDL Measurement Using The Agilent 8169A Polarization Controller, Product Note
p/n 5964-9937E

[5] State of the Art characterization of optical components for DWDM application, Application Brief
p/n 5980-1454E

August 8, 2005
5988-4100EN

Agilent Technologies