NETTEST

# OMETS cPCI/PCI OTDR Reference Manual

PN: 39407B
October 2005

©2005 NetTest North America Inc., a Member of the Anritsu Group
All rights reserved.

This document and the product to which it relates are protected by copyright law from unauthorized reproduction.

**Notice to U.S. Government End Users**
The Software and Documentation are “Commercial Items,” as that term is defined at 48 C.F.R. 2.101, consisting of “Commercial Computer Software” and “Commercial Computer Software Documentation,” as such terms are used in 48 C.F.R. 12.212 or 48 C.F.R. 227.7202, as applicable. Consistent with 48 C.F.R. 12.212 or 48 C.F.R. 227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to the U.S. Government end users (a) only as Commercial Items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished rights reserved under the copyright laws of the United States.

NetTest North America Inc. SHALL NOT BE LIABLE FOR DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING, BUT NOT LIMITED TO, LOSS OF USE, REVENUE, OR PROSPECTIVE PROFITS RESULTING FROM THE USE OF THIS DOCUMENT OR THE PRODUCT TO WHICH IT RELATES. ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE HEREBY DISCLAIMED.

The information in this manual may be subject to change without notice.

NetTest North America Inc.
Center Green, Building 4
6 Rhoads Drive
Utica, New York 13502

## Document Revision History

| Manual Revision | Date | Software Release | Changes in this Release |
|---|---|---|---|
| A | April 2004 | 20040415 | Initial release |
| B | October 2005 | 20050901 | Manual updated to include new software features and WEEE information. |

## Compliance Information

Units bearing the CE Marking have been tested to show compliance to the EMC Directive 89/336/EEC and the Low Voltage Directive 73/23/EEC. Copies of compliance documentation are available from either NetTest Customer Service or Anritsu A/S, Kirkebjerg Alle' 90, 2605 Brondby, Denmark.

Units bearing the C-Tick mark have been tested to show compliance to Australia's Framework for EMC. Copies of compliance documentation are available from either NetTest Customer Service or CommsForce Pty Ltd., 1020 Doncaster Road, East Doncaster, VIC 3109, Australia.

## Laser Safety

The OMETS cPCI/PCI OTDR test ports are fully compliant with both the CDRH(FDA) Federal Register 21CFR parts 1000 and 1040 and EN60825-1 Class I Laser Emissions levels. The Class I level is considered to be eye and radiation exposure safe. This compliance is met when the product is used as intended.

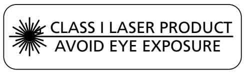
To avoid any possibility of eye injury, adhere to the following warnings:

DO NOT stare into these ports or into the end of a fiber connected to these ports when the instrument power is on. They emit invisible infrared laser radiation

- DO NOT inspect the optical ports or fiber end with a lens or 'fiber microscope' while power is on. It is possible that a lens, viewer or scope could concentrate the laser power to injurious levels.
- DO NOT clean the optical port while the power is on.

- DO keep these ports and the ends of connected fibers covered when the instrument is in use.

The following is a facsimile of the label located on the front panel that identifies the Class I laser ports.


CLASS I LASER PRODUCT
AVOID EYE EXPOSURE

**Caution:** Use of controls or adjustments or performance of procedures other than those specified herein may result in hazardous radiation exposure.

## Electrical Safety

To avoid personal injury, do not operate the product unless the case is properly assembled.

**Warning:** There is a potential shock hazard if equipment is used incorrectly.

**Warning:** Always power off your computer before installing or removing the OMETS cPCI/PCI OTDR components.

## Disposal according to WEEE

### Recycling (for EEC countries only)

**After this product has served its purpose, it should be recycled according to local regulations.**

In the European Union, the WEEE (Waste Electronic and Electrical Equipment) Directive 2002/96/EC specifies that electronic waste be returned to a recycling center for dismantling and re-use of materials. Please contact NetTest Technical Support (see page vi) or your local NetTest distributor for directions as to disposal of NetTest products in your area.

**Note**

Do not dispose of the device, including accessories, in the household waste!

All returns require a Return Materials Authorization (RMA) number. If possible, return the unit in its original shipping materials. If the original shipping materials are not available, please contact NetTest for instructions (see “Technical Support Center” below).

## Technical Support, Service, and Repairs

Our Technical Support Center is at your service to answer technical questions and provide return authorization for service, repairs, or other returns.

### Technical Support Center

+1 800-443-6154 (USA, Canada, and Mexico)
+1 315-266-5000
Fax: +1 315-797-3010
Technical Support e-mail: utica.support@nettest.com
Customer Repair Center e-mail: utica.service@nettest.com
Web: www.nettest.com

All returns require a Return Materials Authorization (RMA) number. If possible, return the unit in its original shipping materials. If the original shipping materials are not available, please contact the Technical Support Center for instructions.

Whether or not the warranty period of your NetTest product has expired, the unit can be returned to NetTest for repairs. Out-of-warranty repairs are billed for time and materials. Call our Technical Support Center for further information.

NetTest offers a performance verification service that extensively tests this product and documents its performance. All test equipment used in the verification process is traceable to NIST or NPL standards, and is calibrated annually. NetTest recommends that you have your unit calibrated annually at our factory.

# Table of Contents

# Chapter 1: Getting Started

## Overview

The Optical Module Embedded Test System (OMETS) cPCI/PCI OTDR is designed to be integrated into Remote Fiber Test Systems (RFTS) or embedded in transmission systems for accurate fault location and preventative maintenance of optical communication networks. The OMETS cPCI/PCI OTDR is available in two formats:

- Compact PCI (cPCI) for rack mount applications
- PCI interface card for use with desktop PCs.

## System Requirements

The following are the minimum system requirements for the OMETS cPCI and PCI OTDRs.

### cPCI System Requirements

| | |
|---|---|
| Bus: | 32bit / 33Mhz cPCI, 3.3VDC or 5VDC |
| Chassis: | 6U cPCI card compatible |
| Slots: | Uses 2 adjacent cPCI card slots (total width = 8HP), but has only one bus presence |
| OS: | Windows 2000/XP |
| CPU: | Pentium II or Celeron 300 MHz, or better |
| Memory: | 256MB RAM, minimum |
| CD-ROM Drive: | 2x or faster |
| Disk space: | 5MB free space, application plus manual. (User will need to define extra space for the test data storage) |
| Power: | 20 Watts available |
| Mounting: | All optics and electronics are mounted on the 8HP, two slot module. |

## PCI System Requirements

| | |
|---|---|
| Chassis: | Standard desktop or tower type 'PC' |
| Slots: | 1 free PCI card slot, (can use either a 3.3V, a 3.3V/5V, or a 5V PCI slot) |
| OS: | Windows 2000/XP |
| CPU: | Pentium II or Celeron 300 MHz, or better |
| Memory: | 256MB RAM, minimum |
| CD-ROM Drive: | 2x or faster |
| Disk space | 5MB free space, application plus manual. (User will need to define extra space for the test data storage) |
| Power: | 20 Watts available |
| Test Optics: | Internal Version only requires one (1) open 3.5" compatible drive bay |

## Unpacking and Inspection

**The OMETS cPCI OTDR includes the following:**

- cPCI OTDR module
- Software CD
- Wrist strap

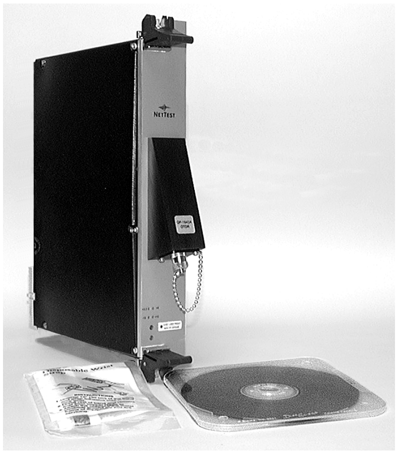


*Figure 1-1: OMETS cPCI OTDR*

**The external OMETS PCI OTDR includes the following:**

- PCI acquisition card
- Ribbon Cable
- OTDR module
- Software CD
- Wrist strap

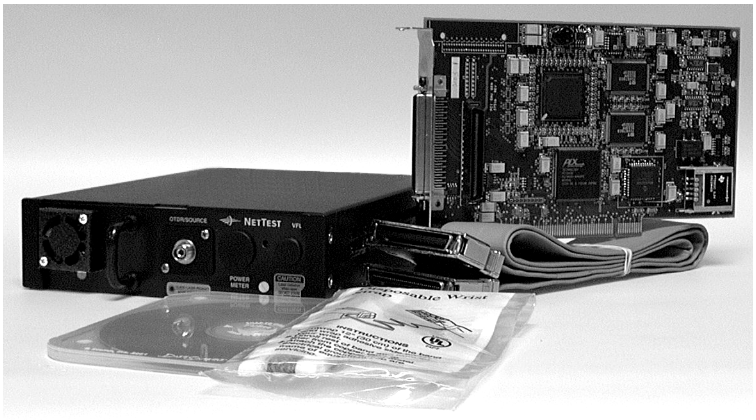


*Figure 1-2: OMETS PCI OTDR*

You should also receive a packing list. Inspect the packing list to make sure you have received the equipment as ordered.

## Theory of Operation

### Optical Time Domain Reflectometer (OTDR)

The OMETS cPCI/PCI OTDR operates by transmitting pulses of light from a laser into a fiber and measuring the reflected light with a detector. Reflections are measured at specific points along the fiber. The OTDR correlates these measurements in the correct sequence and constructs a “fiber trace.” Distance and loss measurements are taken from the fiber trace.

The fiber trace is constructed from two types of basic reflections: Fresnel reflections and backscatter reflections. Fresnel reflections occur where there is not a continuous glass core within the fiber cable, due to mechanical splices, breaks in the fiber, an so on. Backscatter reflections are generated uniformly along the fiber.

A sample fiber trace is shown in Figure 1-3. This figure illustrates many trace features encountered during measurements.

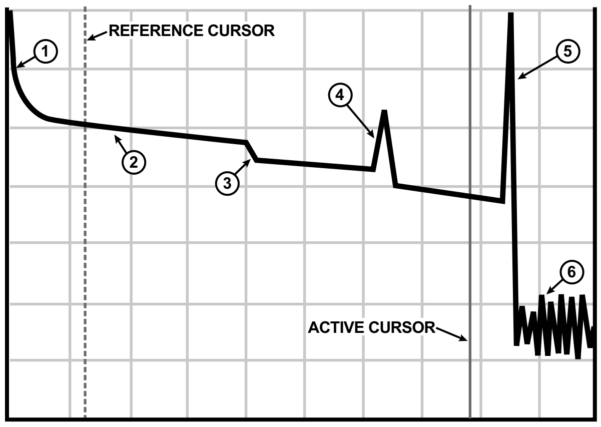


*Figure 1-3: Fiber Trace Features*

The downward slope of the fiber trace is a result of the gradual attenuation (weakening) of the light signal. A splice causes an abrupt loss of power in the fiber, which results in characteristic steps in the fiber trace. The attenuation of the fiber can be calculated by measuring the slope of the backscatter signal from any section of the fiber in the fiber trace. Splice loss measurements can be calculated by measuring the amount of power drop across a splice.

**Note**

A range setting less than the actual length of the fiber can distort the fiber trace. Be sure the range setting is greater than the total length of the fiber under test.

| No. | Description |
|---|---|
| 1 | Initial Pulse<br><br>This is the initial reflection due to the connection of the OTDR to the fiber under test. Useful measurement information begins after this pulse. |
| 2 | Backscatter Signal<br><br>This section of the trace shows the continuous backscatter from the fiber. The downward slope indicates attenuation in the fiber. |
| 3 | Nonreflective Event<br><br>The abrupt drop in the backscatter signal is caused by the loss in a nonreflective event. Fusion splices are the most common cause of nonreflective events. A similar drop in the backscatter signal can occur at the location of a cable fault. |
| 4 | Reflective Event<br><br>Any vertical pulse that appears in the fiber trace and past the initial pulse is called a reflective event. In addition to the initial pulse, there are three kinds of reflective events: reflective splices, end reflections, and ghost reflections. A reflective splice shows up in the fiber trace as a vertical pulse and a step change in the backscatter signal. Reflective event are generally caused by mechanical splices and are easier to locate on the fiber trace than the nonreflective event caused by a fusion splice. See No. 5 for detail on end reflections.<br><br>A ghost reflection has no step change in the backscatter. A ghost reflection is not due to a feature at the display location but to a strong reflection that has occurred elsewhere in the fiber. |

| No. | Description |
|---|---|
| 5 | End Reflection<br><br>A reflective event that appears at the end of a fiber is called an end reflection. An end reflection is identified by the fact that there is no further backscatter signal after this pulse.<br><br>**Note**<br>A fiber without a clean, flat end (e.g., when a break has occurred) may not show an end reflection. |
| 6 | Noise Floor<br><br>The noise floor is the sensitivity limit of the OTDR. The noise floor is lowered by the averaging process. |

# Chapter 2: Installation

## Software Installation

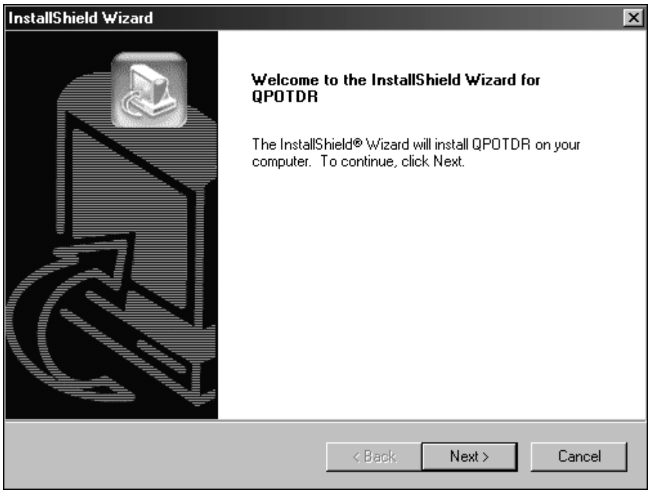
Use the following procedure for PC installation of the QPOTDR software.

1. With the PC operating, place the OMETS OTDR Support CD into the PC's CD drive.
2. Select the Start button on the Windows desktop.
3. Select **Run...**, the Run dialog appears with "X:\Setup.exe" in the Open field (where "X" is the letter of the CD drive). Select **OK** to continue.

**Note**

If something other than "X:\Setup.exe" is displayed in the **Open** field (where "X" is the letter of the CD drive), select **Browse** and then use the **Look in** field to select the CD drive. Once the CD drive is selected, highlight **Setup.exe** in the field directly below the **Look in** field and then select **Open**.

4. The **QPOTDR Setup** window appears. Follow the instructions displayed in the InstallSheild® Wizard (see Figure 2-1). Select **Next** to continue the installation.

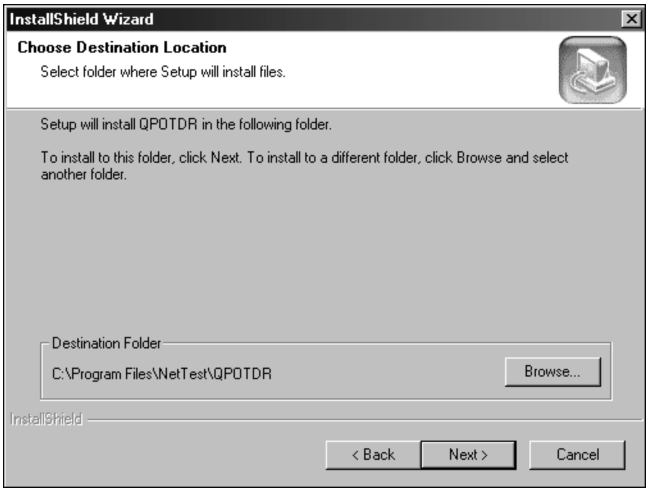

InstallShield Wizard

**Welcome to the InstallShield Wizard for QPOTDR**

The InstallShield® Wizard will install QPOTDR on your computer. To continue, click Next.

*Figure 2-1: Welcome to the InstallSheild Wizard dialog*

5. The **Choose Destination Location** dialog appears:
   - Select **Next** to use the default location "C:\ProgramFiles".
     or
   - Select **Browse** in the Destination Folder field to select a different destination location and then select **Next** to continue.



InstallShield Wizard

Choose Destination Location

Select folder where Setup will install files.

Setup will install QPOTDR in the following folder.

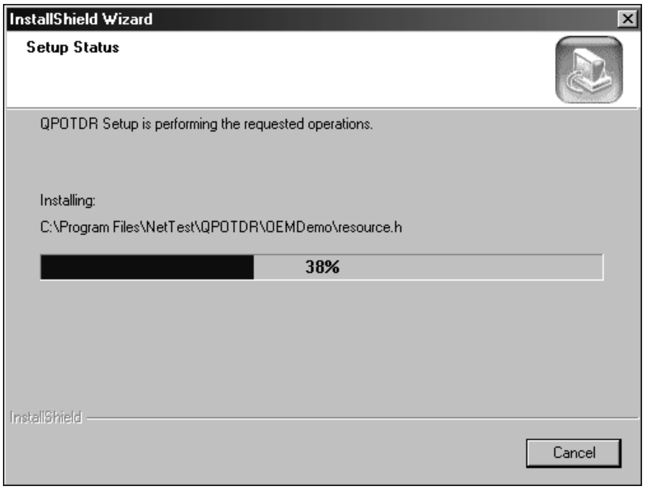
To install to this folder, click Next. To install to a different folder, click Browse and select another folder.

Destination Folder

C:\Program Files\NetTest\QPOTDR

*Figure 2-2: Destination Location dialog*

6. The **Setup Status** dialog appears. The status bar fills in as the software installation progresses.



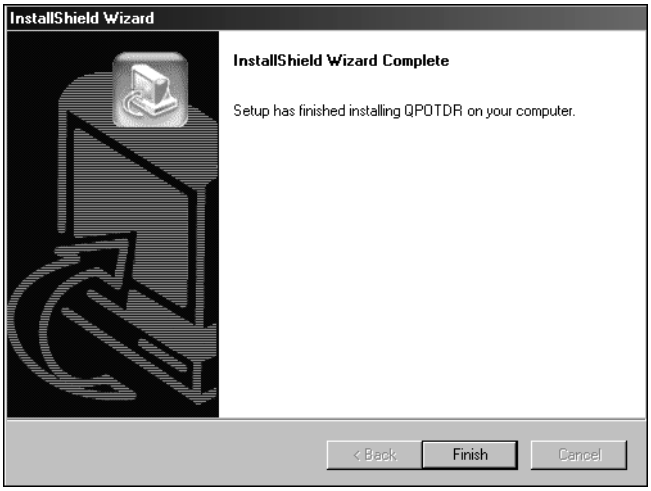
QPOTDR Setup is performing the requested operations.

Installing:
C:\Program Files\NetTest\QPOTDR\OEMDemo\resource.h

38%

*Figure 2-3: Setup Status dialog*

7. The **InstallSheild Wizard Complete** dialog appears (see Figure 2-4). Select **Finish** to complete the installation and exit the wizard.



InstallShield Wizard Complete

Setup has finished installing QPOTDR on your computer.

*Figure 2-4: InstallSheild Wizard Complete dialog*

8. Remove the software CD from the CD drive.

## Hardware Installation

### External PCI Installation

**Note**

You must use the wrist strap provided with the OMETS PCI OTDR to protect the PCI card from electrostatic discharge during installation. Failure to use the wrist strap may permanently damage the card.

#### Installing the PCI card

1. Power OFF your computer. Unplug and/or remove the computer's line cord.

**Warning:** Always power off your computer before installing or removing the OMETS cPCI/PCI OTDR components.

2. Remove the cover or panel on your PC which allows access to the unit's expansion card slots.
3. Use the static wrist strap included in the packing box. Slip the flexible strap over your wrist. Attach the clip to the bare metal part where the card slot covers are in your computer.
4. Remove a card slot cover from your computer and save the screw for later use.
5. Remove the PCI card (see Figure 2-5) from the packing box.

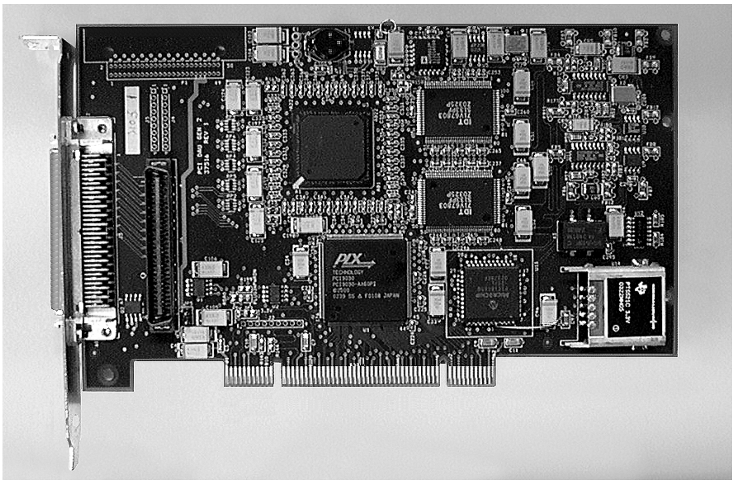


*Figure 2-5: PCI Card*

6. Insert the PCI card into the slot where you removed the slot plate.

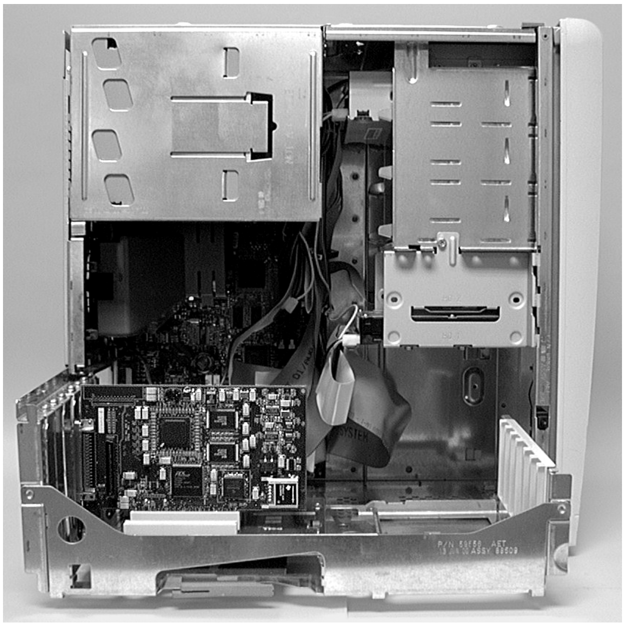


*Figure 2-6: Example of the PCI OTDR installed in a PC*

7. Secure the PCI card bracket with the screw reserved in step 4.
8. Attach one end of the ribbon cable to the connector on the PCI card and attach the other end to the connector on the back of the OTDR module.

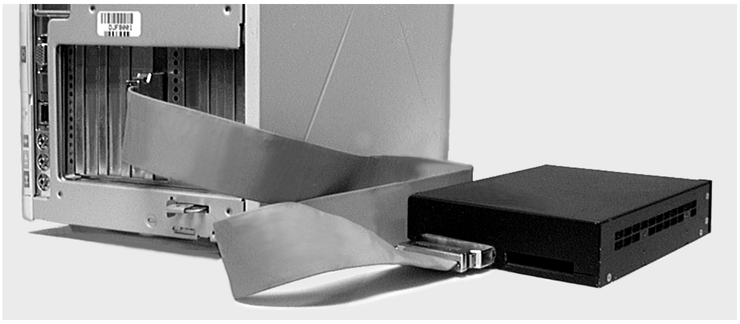


*Figure 2-7: Ribbon Cable connection from PC to OTDR Module (rear view)*

9. Power up the PC. If this is a first time installation, the Found New Hardware Wizard appears at the Windows desktop. See “Installing the Hardware Device Driver” on page 2-9 for details.

## cPCI Installation

To install, remove, or replace an OMETS cPCI OTDR module, you need the following:

- A small flat blade screwdriver
- A small Phillips head screwdriver
- An ESD wrist strap
  (A disposable ESD wrist strap is included for first time installation use.)

**Warning:** Electro Static Discharge (ESD) can damage electronic components. To avoid ESD when handling an OMETS cPCI OTDR module, use a working ESD wrist strap prior to installing, removing or replacing the OMETS cPCI OTDR module,

### Installing/Replacing an OMETS cPCI OTDR module

1. Remove the OMETS cPCI OTDR module from its ESD bag. Take care to handle the OMETS cPCI OTDR module by its faceplate or the ejector handles, avoiding touching any electronic elements.
2. Disable the power to the cPCI chassis.
3. **If installing a new OMETS cPCI OTDR module:**
   If necessary, remove the EMI/RFI panel that corresponds to the slot position where the OMETS cPCI OTDR card will be installed. Save the panel and screws for future use.
   Continue at step 5.
4. **If replacing an OMETS cPCI OTDR card:**
   - Use a Phillips head or flat blade screwdriver (as appropriate) to loosen the screws at both the upper and lower ejector handles on the card being removed.

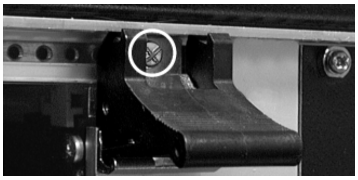


*Figure 2-8: Ejector Handle Detail - Screw Location*

- Remove the module by activating the ejection handles. Simultaneously press upward on the upper handle while pressing downward on the lower handle.
- Carefully slide the module out of the slot.

5. Align the module vertically with the open slot. Make sure that the printing on the front panel is not upside down and that the module board is engaged in the guides at the top and bottom.

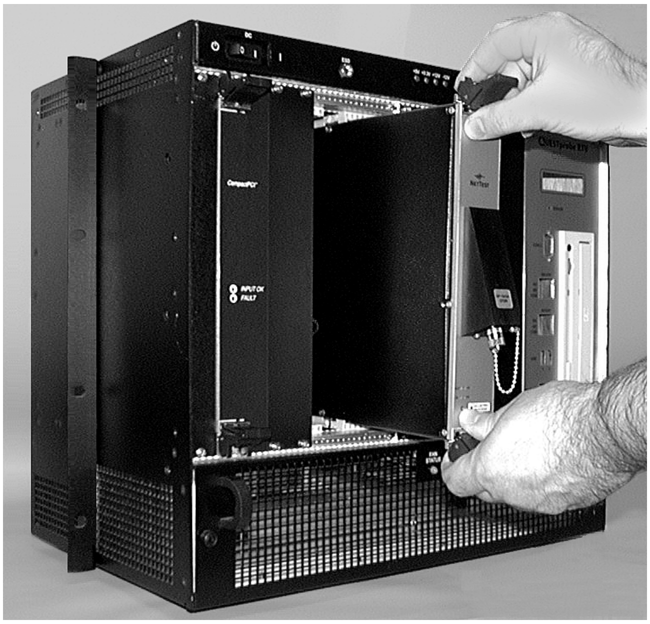


*Figure 2-9: cPCI Module Installation*

6. When the module is placed into the guides, carefully slide the module straight into the chassis, making sure that components on the module do not interfere with adjacent panels.

**Caution:** Correct seating of the module's connector requires precise alignment, as a bent pin on the backplane could yield the system inoperable.

7. As the module begins to engage with the bus connector, you will feel some additional resistance. This is normal. The latching hooks on the ejector handles will begin to line up with the holes on the cPCI chassis mounting rails.

- Press the ejector handles towards each other. The hooks on the handles should insert into the holes in the mounting rail to provide mechanical leverage which helps to ensure that the module will be fully seated.
- While the module is being pulled into the chassis by the handles, make sure that the EMI/RFI gasket strips on the side of the module's front panel, and on the side of the panel next to it, are properly aligned and do not bend or deform.
- Firmly press the handles towards each other to latch the module into the chassis. When fully seated the front plate of the module is flush with the panels of the adjacent slots.

8. Use a Phillips head or flat blade screwdriver (as appropriate) to tighten the screws of the ejector handles (see Figure 2-8: Ejector Handle Detail - Screw Location) to provide an additional measure of mechanical security.

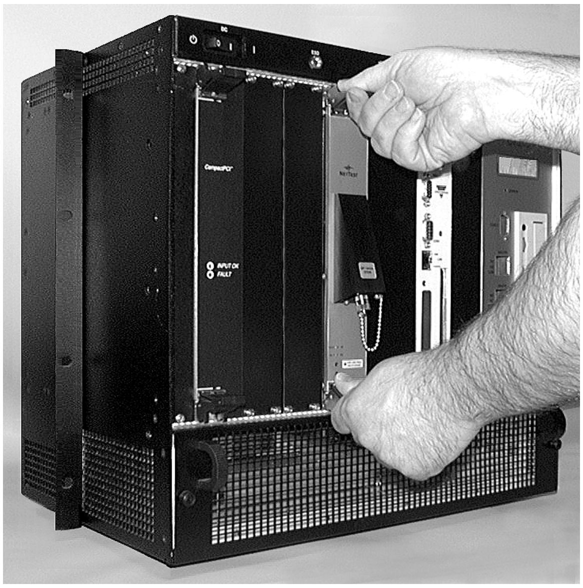


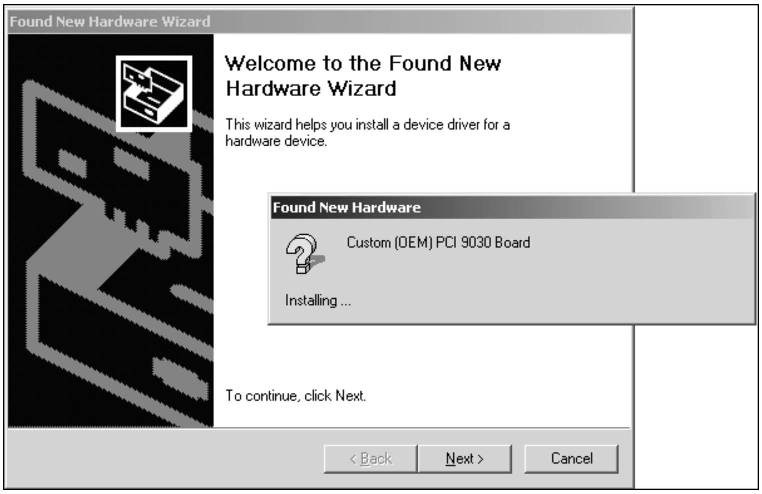
*Figure 2-10: Engaging the Ejector Handles*

9. Power up the cPCI chassis. If this is a first time installation, the Found New Hardware Wizard appears at the Windows desktop. See "Installing the Hardware Device Driver" on page 2-9 for details.

## Installing the Hardware Device Driver

The Found New Hardware Wizard appears when Windows is launched after the first time installation of either the OMETS PCI or cPCI OTDR.

1. Install the QPOTDR software, if not already installed. See “Software Installation” on page 2-1 for details.
2. Install the OMETS PCI (see,“External PCI Installation” on page 2-4) or OMETS cPCI OTDR (see “cPCI Installation” on page 2-6).
3. Power up the PC or cPCI chassis. Once Windows initializes, the Found New Hardware Wizard appears. Select **Next** to continue.


Found New Hardware Wizard

Welcome to the Found New Hardware Wizard

This wizard helps you install a device driver for a hardware device.

Found New Hardware

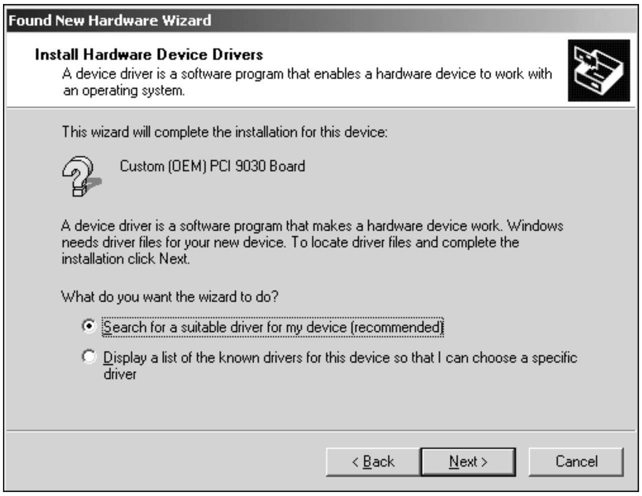
Custom (OEM) PCI 9030 Board

Installing ...

To continue, click Next.

*Figure 2-11: Found New Hardware Wizard*

4. The Install Hardware Device Drivers dialog appears. Select **Search for a suitable driver for my device (recommended)** (see Figure 2-12 on page 2-10) and then select **Next** to continue.



Found New Hardware Wizard

**Install Hardware Device Drivers**
A device driver is a software program that enables a hardware device to work with an operating system.

This wizard will complete the installation for this device:

Custom (OEM) PCI 9030 Board

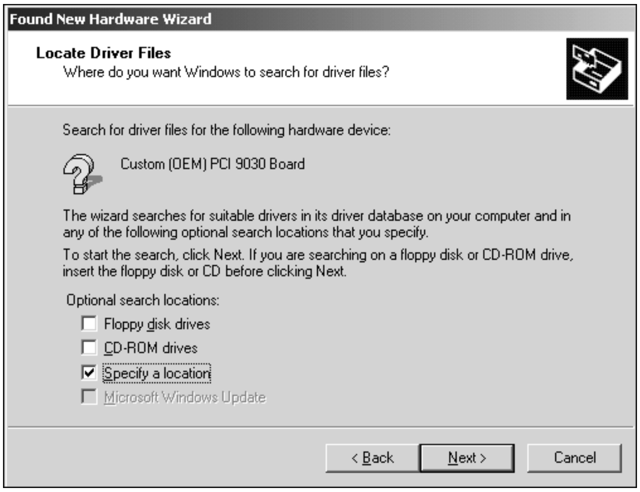
A device driver is a software program that makes a hardware device work. Windows needs driver files for your new device. To locate driver files and complete the installation click Next.

What do you want the wizard to do?

- ● Search for a suitable driver for my device (recommended)
- ○ Display a list of the known drivers for this device so that I can choose a specific driver

*Figure 2-12: Install Hardware Device Drivers dialog*

5. The Locate Driver Files dialog appears. Select **Specify a location** and then select **Next** to continue.



Found New Hardware Wizard

**Locate Driver Files**
Where do you want Windows to search for driver files?

Search for driver files for the following hardware device:

Custom (OEM) PCI 9030 Board

The wizard searches for suitable drivers in its driver database on your computer and in any of the following optional search locations that you specify.

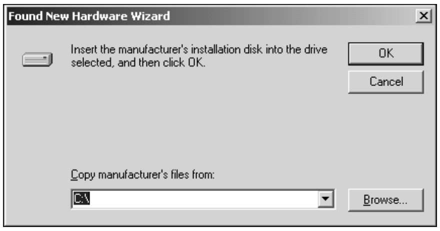
To start the search, click Next. If you are searching on a floppy disk or CD-ROM drive, insert the floppy disk or CD before clicking Next.

Optional search locations:

- ☐ Floppy disk drives
- ☐ CD-ROM drives
- ☑ Specify a location
- ☐ Microsoft Windows Update

*Figure 2-13: Locate Driver Files dialog*

6. The Found New Hardware Wizard -Browse dialog appears. Select **Browse** to continue.



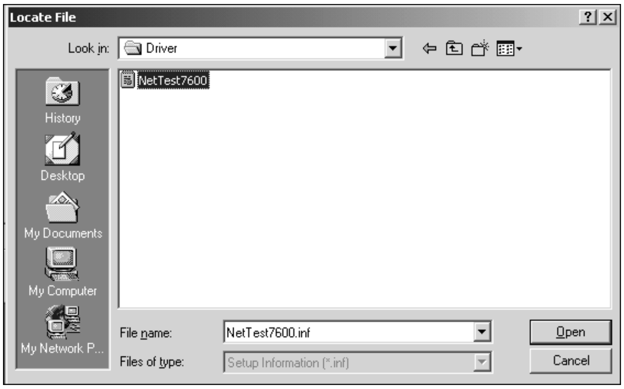
Insert the manufacturer's installation disk into the drive selected, and then click OK.

Copy manufacturer's files from:

C:\

*Figure 2-14: Found New Hardware Wizard - Browse dialog*

7. The Locate File dialog appears.



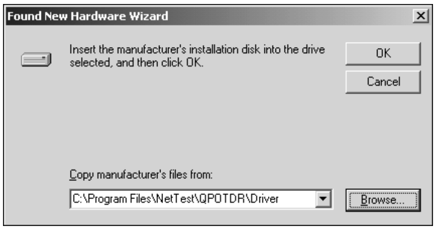
File name: NetTest7600.inf

Files of type: Setup Information (*.inf)

*Figure 2-15: Locate File dialog*

- Navigate to the Driver folder (Windows (C:)\Program Files\NetTest\QPOTDR\Driver) and select **NetTest7600.inf.**
- Select **Open** to continue.

8. The Found New Hardware Wizard - Browse dialog reappears with the path shown in the Browse field as shown in the following figure, select **OK** to continue.

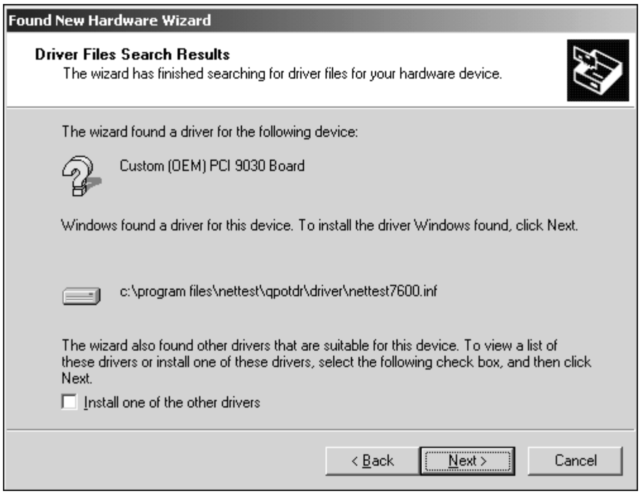


Insert the manufacturer's installation disk into the drive selected, and then click OK.

Copy manufacturer's files from:

C:\Program Files\NetTest\QPOTDR\Driver

9. The Driver Files Search Results dialog appears, displaying the path for the driver (c:\program files\nettest\qpotdr\driver\nettest7600.inf), select **Next** to continue.



Found New Hardware Wizard

Driver Files Search Results

The wizard has finished searching for driver files for your hardware device.

The wizard found a driver for the following device:

Custom (OEM) PCI 9030 Board

Windows found a driver for this device. To install the driver Windows found, click Next.

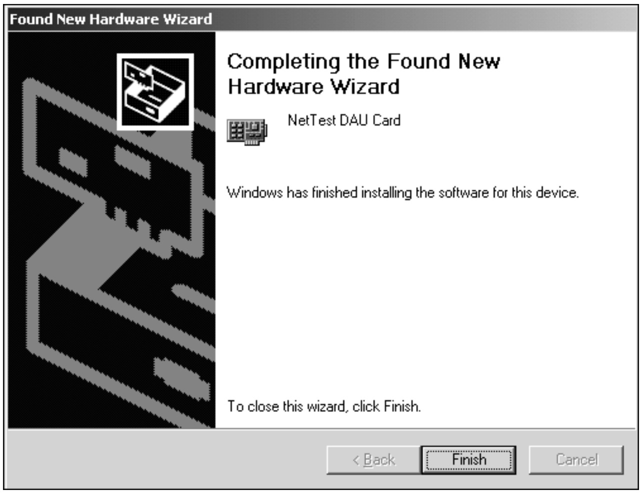
c:\program files\nettest\qpotdr\driver\nettest7600.inf

The wizard also found other drivers that are suitable for this device. To view a list of these drivers or install one of these drivers, select the following check box, and then click Next.

☐ Install one of the other drivers

*Figure 2-16: Driver Files Search Results dialog*

10. The Completing the Found New Hardware Wizard dialog appears. Select **Finish** to complete the process and close the wizard.



Found New Hardware Wizard

Completing the Found New Hardware Wizard

NetTest DAU Card

Windows has finished installing the software for this device.

To close this wizard, click Finish.

# Chapter 3: Using the OMETS OTDR

## User Controlled Functions

- Module Initialization
- Querying/Setting Acquisition parameters
- Starting and Stopping Data Acquisition cycles
- Retrieval of Data and Acquisition settings
- Retrieval of Event Analysis Table
- Setting of Event Analysis Thresholds
- File Save
- File Load
- Getting and Setting Trace Parameters

## Operating System Requirements

The OMETS OTDR product is comprised of a cPCI or PCI bus compatible data acquisition unit (DAU) card. The optics are integral to the cPCI DAU assembly, but are separate from the PCI DAU assembly. Also included are the Windows compatible DLLs (Dynamic Link Libraries) and ancillary files (drivers, etc.) that allow the user to assert complete control in the Windows 2000/XP Operating System environment.

## Module Initialization

Use the "**QPOTDRInitialize**" function to initialize the OTDR module. This function performs several tasks, such as loading the module's setup EEPROM data, initializing several DAU (Data Acquisition Unit) and Optics hardware registers, and performing data interleave calibration.

The function returns a status code indicating either a successful completion, or a failure. Failure codes are documented in the section titled "Initialization and Setup API Functions" on page 5-1.

## Retrieval of Module Parameters

Use the "**QPOTDRGetAvail...()**" functions to retrieve the parameters from all modules.Use the data returned by these functions to fill the allocated user-arrays.

Information such as available test wavelengths, available pulsewidths, available ranges and data point spacing are returned by the appropriate functions. Use this information to create arrays containing the available parameters. You must maintain these arrays and use them to select the desired settings for data acquisition.

Each of the QPOTDRGetAvail...() functions have associated a Max Count #define so you can allocate the arrays adequately and maintain legal setting within the arrays. The legal setting for Point spacing is determined by the currently selected Range and Wavelength.

The parameter units for the Range, Point Spacing, Pulsewidth and Wavelength are:

Range in Kilometers

Point Spacing in Meters

Pulsewidth in nanoseconds

Wavelength in nanometers

## Selecting Acquisition Settings

To achieve optimum OTDR performance, it is critical that certain rules are followed in the selection of acquisition parameters.

This section provides useful information for selecting settings for data acquisition and includes general guidelines based on the length of the fiber under test

| Parameter | Description |
|---|---|
| Wavelength | The wavelength of the laser that the OTDR will use to test the fiber, expressed in nanometers (nm). The wavelength(s) available will depend on the specific OTDR module currently selected. |
| Range | The optical distance over which the OTDR acquires data for the purpose of locating and identifying features, expressed in kilometers (km). Available Range settings are dependent on the currently selected OTDR module, but typically offer the following settings: 5km, 20km, 50km, 75km, 125km, 250km, and 300km. The Range setting must be longer than the length of the fiber under test to prevent potential distortion of the trace signature and possible occlusion of events. Selecting unnecessarily long range setting will reduce the number of averages that can be performed in a given amount of time. Additionally, the selection of too long a Range could potentially restrict data point spacing to a coarser (longer) setting, sacrificing event and measurement resolution. Select a range setting that is approximately 125% (or more) of the length of the fiber under test. |

| Parameter | Description |
|---|---|
| Pulsewidth | The duration of a laser light pulse launched into the fiber under test by the OTDR. The pulsewidth is specified in nanoseconds (ns) and is accompanied by a specifier denoting whether the pulsewidth is High Resolution (HR) or Long Haul (LH). Available pulsewidth settings for the OTDR module are typically 5ns, 10ns, 20ns, 50ns, 100ns, 200ns, 500ns, 1000ns, 2000ns, 5000ns, 20000ns, and 30000ns. Some modules may not offer the 30000ns pulsewidth, and some modules may offer both High Resolution and Long Haul versions of some of the pulsewidths (1000ns, 2000ns, 5000ns). High Resolution (HR) pulsewidths generally offer better event and attenuation deadzones, but less dynamic range, while Long Haul (LH) pulsewidths sacrifice deadzone specifications for increased dynamic range. |
| Pointspacing | The distance between discrete sampled data points in a data acquisition, expressed in meters. Available Pointspacings for the OTDR are: 0.125m, 0.25m, 0.5m, 1m, 2m, 4m, 8m, and 16m. These pointspacing values are nominal since they are slightly affected by the current IOR (Index of Refraction) value in use. Shorter pointspacings will improve distance accuracy and deadzone/Loss measurements, but will reduce the number of averages that can be accomplished in a given amount of time. |
| Averages | The number of data sweeps (single OTDR data collection samples) that may be accumulated to provide an averaged result. Available average settings for the OTDR are 1 to 32786. Each count represents a number of 128 hardware accumulations. 128 hardware accumulations is called a FastScan. |
| Average Timer | OTDR averaging may also be specified as a total number of seconds of averaging as opposed to a discrete average count. Acceptable ranges are from 5 seconds to 600 seconds. |

## General Guidelines for Selecting Test Parameters

Refer to the following table to select the optimum test parameters based on the length of the fiber under test.

| Fiber length | Range | Pulsewidth | Pointspacing | Filter | FastScan |
|---|---|---|---|---|---|
| 1km – 4km | 5km | 5ns – 50ns | .125m – .5m | OFF | 16 – 64 |
| 4km – 15km | 20km | 100ns – 500ns HR | .250m – 1m | OFF | 96 |
| 15km – 40km | 50km | 500ns – 1000ns HR | .50m – 2m | OFF | 192 |
| 40km – 100km | 125km | 2000ns – 10000ns | 1m – 8m | ON | 384 |
| 100km – 200km | 250km | 10000ns – 20000ns | 4m – 16m | ON | 768 |
| 200km – 250km | 300km | 20000ns – 30000ns | 8m – 16m | ON | 1536 |

## Data Acquisition Procedure

The following steps must be performed before initiating a Data Acquisition cycle:

1. Select the wavelength for the test. The available wavelengths for a module are obtained from the “**OPOTDRGetAvailWaves()**” function and the user is responsible for initializing and accessing the resultant array for available wavelengths.
2. Select the Range. The range must be greater than the actual length of the fiber under test. Refer to the table shown in “General Guidelines for Selecting Test Parameters” above for general guidelines in selection of a Range.
3. Select the test pulsewidth. Again, refer to the table shown in “General Guidelines for Selecting Test Parameters” above for general guidelines in selecting a test pulsewidth. The best results for distance and measurement accuracy are obtained by using the shortest pulsewidth that provides sufficient dynamic range. Insure that the pulsewidth selection and the number of averages yields at least 5dB to 6dB between the measured power level at the end of the fiber under test and the noise floor. This will

assure accurate fiber measurements and fiber-end detection in the Events Analysis algorithms. If the level of optical power is insufficient, increase the test pulsewidth to increase the dynamic range. Doubling the pulsewidth should yield a 1.5dB increase in dynamic range, given the same amount of averaging.

4. Select the Pointspacing. For optimum results, select a Pointspacing that provides at least 4 data points per pulsewidth. Use the following formula to determining the points per pulsewidth.

$$\text{Points per Pulsewidth} = \frac{\text{Pulsewidth in nanoseconds / 10}}{\text{Pointspacing in meters}}$$

Example: Pulsewidth of 50ns, with a Pointspacing of 0.50 meters yields 10 points per pulsewidth:

$$\frac{50 / 10}{0.50} = 5 / 0.50 = 10$$

5. Select number of averages or averaging time (in seconds).

## Initiating a Data Acquisition Cycle

### Setup Acquisition Parameters

The first function called is "**QPOTDRAcqSetParams()**". Make sure the desired data acquisition parameters are set.

### Initiate Data Acquisition

The next function called is "**QPOTDRAcqStart()**". This function begins the data acquisition process.

## Continuing a Data Acquistion

After initiating a Data acquisition, the function "WaitForMultipleObjects()" must be called in a loop to detect the progress of the Data Acquisition. This is a Win32 API that handles event detection. Refer to the sample source code in the OEMDemo folder.

## Termination of a Data Acquisition Cycle

Use the "**QPOTDRAcqStop()**" function to terminate a data acquisition process.

# Chapter 4: Data Structure & API Prototype Information

## Data Collection/Parameter Setup

```
// ONT.UT.QPOTDR.H Defines and exports

#if !defined(_ONT_UT_QPOTDR_H_INCLUDED_)
#define _ONT_UT_QPOTDR_H_INCLUDED_

#ifdef __cplusplus
extern "C" {
#endif

// Defines
#if !defined QP_API_
#define QP_API_(type) __declspec(dllimport) type WINAPI
#endif

#if !defined QP_API
#define QP_API __declspec(dllimport) HRESULT WINAPI
#endif

#define MAX_CARDS           8
#define MAX_WAVES           4
#define MAX_RANGES          16
#define MAX_PULSES          16
#define MAX_AVERAGES        3
#define MAX_SPACINGS        13
#define MAX_DIST_RANGES     10
#define MAX_WFM_POINTS      0x40000
#define REAL_TIME_SCAN      1
#define LIVE_FIBER_DETECT   0x00010000
#define GAIN_SPLICE_ON      2

#define EV_TYPE_JUNK       -1
#define EV_TYPE_LAUNCH      0   // 1st event, index 0.
#define EV_TYPE_REFL        1
#define EV_TYPE_NONREFL     2
#define EV_TYPE_GROUPED     3
#define EV_TYPE_END         4
#define EV_TYPE_QT_END      5

#define EV_SATURATED         0X00000001  // Saturated.
```

```
#define EV_END_OF_FIBER      0X00000002  // End of fiber event.
#define EV_LOSS_EXCEEDED     0X00000004  // Loss exceeds Loss Threshold.
#define EV_REFL_EXCEEDED     0X00000008  // Refl exceeds Refl Threshold.
#define EV_END_OF_DATA      0X00000100  // For launch event : no backscatter
or multiple adjacent reflections.
#define EV_OUT_OF_RANGE      0X00000200  // Dynamic range exceeded.
#define EV_OUT_OF_DIST       0X00000400  // Distance range exceeded.
#define EV_RISING_EDGE       0X00002000  // Event ended on a rising edge.
#define EV_BACKSCATTER       0X00004000  // Launch or refl event ended on
backscatter.
#define EV_REFL_CLAMPED      0X00040000  // Reflection is clamped.
#define EV_PW_RESO_ERR      0X00080000  // PW/Reso combination causes poorly
resolved reflections.
#define EV_END_NTH_REFL      0X01000000  // End set to Nth Refl with
Refl>=EndNthReflThreshold.

// Waveform data format
typedef unsigned short  QPOTDRWaveformData;

// Waveform signFlags format
typedef unsigned long   QPOTDRWaveformSignFlags;

// Waveform header, contains information about waveform
typedef struct
{
   unsigned short        updateData;       // Update data flag (TRUE=buffer
is updated)
    DWORD                   FPOffset;           // Front panel distance(m)
from first data point.
    double                  BaseLine;// Baseline
    double                  Reference;// Reference
    unsigned long           Averages;           // # of acquisitions
    double                  Noise;              // dB Rms noise * 1000
    int                     NumPts;             // # of data points
    QPOTDRWaveformSignFlags   *SignFlags;         // array of sign flags
    size_t                  SignFlagSize;       // size of sign flag array
    BYTE                    gainSpliceFailed;   // Gain splice failed flag
} QPOTDRWaveformHeader;

// Plugin Info, information about plugin
typedef struct _PluginInfo
{
    char ModelNumber[32];// Model number
    char SerialNumber[32];// Serial Number
    char szLastCalibrationDate[16];// Last Cal Date
    char szPartNumber[16];// Part number / Revision
    char szModel[16];// Model name
    char szManufacturer[20];// Manufacturer name
    char opt_file_version[10];// OPT file version
    char dll_version[22];// QPOTDR Version
} QPPluginInfo, *PQPPluginInfo;

// Plugin / DAU specific info
typedef struct
{
    DWORD dwMaxDataPoints;// Maximum data points supported
    float fBaseReso;// Base reso
    WORD wRevision;// Revision ID
    QPPluginInfo pluginInfo;// Information about plugin
    DWORD dwMaxFastAvg;// Maximum number of hardware averages
    DWORD dwFastScanCount;// Number of averages in hardware per scan
} QPPluginData;

// Fiber Analysis Event info
typedef struct _QPOTDREVENT
{
    int iType;// Event type
    DWORD dwStart;// Event start data point
    DWORD dwEnd;// Event end data point
    double dLoss;// Event loss
    double dRefl;// Event reflectance
    DWORD dwFlags;// Special flags
    double dSpanLoss; // Span Loss or QPOTDR_CALC_ERROR
    double dSpanLossdBkm; // Span Loss dB/km or QPOTDR_CALC_ERROR
} QPOTDREVENT, *PQPOTDREVENT;

///////////////////////////////////////////////////////////////////////////
// QPOTDRGetMaxWaves
// card - card index
// Returns maximum number of waves for card
QP_API_(int) QPOTDRGetMaxWaves( int card );

///////////////////////////////////////////////////////////////////////////
// QPOTDRGetMaxPointSpacings
// card - card index
// Returns maximum number of point spacings for card
QP_API_(int) QPOTDRGetMaxPointSpacings( int card );

///////////////////////////////////////////////////////////////////////////
// QPOTDRGetMaxPulses
// card - card index
// wave - wavelength
// Returns maximum number of pulses for wave
QP_API_(int) QPOTDRGetMaxPulses( int card, float wave );

//////////////////////////////////////////////////////////////////////////
// QPOTDRGetMaxRanges
// card - card index
// wave - wavelength
// Returns maximum number of ranges for wave
QP_API_(int) QPOTDRGetMaxRanges( int card, float wave );

//////////////////////////////////////////////////////////////////////////
// QPOTDRGetMaxAverages
// card - card index
// wave - wavelength
// Returns maximum number of averages for wave
QP_API_(int) QPOTDRGetMaxAverages( int card, float wave );

//////////////////////////////////////////////////////////////////////////
// QPOTDRGetAvailWaves
// Gets available wavelengths
// card - dau card index
// wavelengths - array of floats, element count = MAX_WAVES
QP_API_(int) QPOTDRGetAvailWaves( int card, float *wavelengths );

//////////////////////////////////////////////////////////////////////////
// QPOTDRGetAvailRanges
// Gets available ranges
// card - dau card index
// wave - wavelength
// ranges - array of floats, element count = MAX_RANGES
QP_API_(int) QPOTDRGetAvailRanges( int card, float wave, float *ranges);

//////////////////////////////////////////////////////////////////////////
// QPOTDRGetAvailPulses
// Gets available pulses
// card - dau card index
// wave - wavelength
// pulsewidths - array of DWORDs, element count = MAX_PULSES
QP_API_(int) QPOTDRGetAvailPulses( int card, float wave, DWORD *pulsewidths
);

//////////////////////////////////////////////////////////////////////////
// QPOTDRGetDynamicRange
// Gets dynamic range for selected pulse
// card - dau card index
// wave - wavelength
// pulse - pulse index
// pulsewidths - array of DWORDs, element count = MAX_PULSES
QP_API_(int) QPOTDRGetDynamicRange( int card, float wave, int pulse, WORD
*wDynamicRange );

////////////////////////////////////////////////////////////////////////////
// QPOTDRGetAvailAverages
// Gets available averages counts
// card - dau card index
// wave - wavelength index
// averages - array of DWORDs, element count = MAX_AVERAGES
QP_API_(int) QPOTDRGetAvailAverages( int card, float wave, DWORD *averages
);

////////////////////////////////////////////////////////////////////////////
// QPOTDRGetAvailSpacings
// Gets available point spasings
// card - dau card index
// point_spacings - array of floats, element count = MAX_SPACINGS
QP_API_(int) QPOTDRGetAvailSpacings( int card, float *point_spacings );

////////////////////////////////////////////////////////////////////////////
// QPOTDRGetDefaultIOR
// wCard - card index
// wave - wavelength
// returns DefaultIOR
QP_API_(float) QPOTDRGetDefaultIOR(WORD wCard, float wave);

////////////////////////////////////////////////////////////////////////////
// QPOTDRGetDefaultBSC
// wCard - card index
// wave - wavelength
// returns DefaultBSC
QP_API_(float) QPOTDRGetDefaultBSC(WORD wCard, float wave);

////////////////////////////////////////////////////////////////////////////
// QPOTDRGetFiberType
// wCard - card index
// wave - wavelength
// returns Fiber type [0 = SM or 1 = MM]
QP_API_(BYTE) QPOTDRGetFiberType(WORD wCard, float wave);

////////////////////////////////////////////////////////////////////////////
// QPOTDRGetCards
// wCardIndexes - card indexes array of type WORD element count = MAX_CARDS
// returns card count
```

```
QP_API_(WORD) QPOTDRGetCards(WORD *wCardIndexes);

//////////////////////////////////////////////////////////////////////////
// QPOTDRInitialize
// Used to initialize the hardware.  In addition, the plugin is read,
// and memory is cleared.  This function must be called before using
// the card.
// Return error codes
// -1   Bad CardId
// -2   Unable to get card status
// -4   Unable to get Xilinx bitstream
// -5   LSR_PWR Initialize Failed
// -6   APD_VOLT Initialize Failed
// -7   PREAMP Initialize Failed
// -8   LSR_LTS Initialize Failed
// -9   LSR Initialize Failed
// -10  LTS_FREQ Initialize Failed
// -11  LTS_PWR Initialize Failed
// -12  PM_VFL Initialize Failed
// -13  LSR_COOL Initialize Failed
// -15  Xilinx General filure
// -16  Xilinx Bad parameter
// -17  Xilinx Bad CardID
// -18  Xilinx Data Block Start
// -19  Xilinx Data Block Middle
// -20  Xilinx Data Block End
// -22  Xilinx Self Test
// -23  EEPROM read error/bad CRC16
// -24  OPT version incorrect
// -25  Unable to determine interleave
// -26  Memory allocation error
// -27  Memory allocation error
// -28  Memory allocation error
// -29  Memory allocation error
// -30  Memory allocation error
// -31  Memory allocation error
// -32  Memory allocation error
// -33  CD OPT structure bad CRC16
// 0    Success
QP_API_(int) QPOTDRInitialize( int cardid );

//////////////////////////////////////////////////////////////////////////
// QPOTDRGetOTDRLaserCount
// wCard - card index
// Returns number of lasers in DAU
QP_API_(BYTE) QPOTDRGetOTDRLaserCount( WORD wCard );
```

```
////////////////////////////////////////////////////////////////////////////
// QPOTDRDataCollectInfo
// card - card index
// pPluginData - pointer to  QPOTDRPluginData structure
QP_API_(void) QPOTDRDataCollectInfo(WORD card, QPPluginData *pPluginData);

////////////////////////////////////////////////////////////////////////////
// QPOTDRAcqSetParams
// wCard - card index
// wAverages - number of averages (number of fast scan counts)
// wWave - wave index
// dwRange - range (number of data points)
// wPointSpacing - point spacing index
// wPulse - pulse index
// wFilter - filter on/off flag
// wUpdateCount - update count (multiple of fast scan count)
// Returns:
// 0  - Success
// -1 - bad card index
// -2 - bad number of averages
// -3 - bad wave index
// -4 - bad pulse index
// -5 - bad range
// -6 - bad point spacing index
QP_API_(int) QPOTDRAcqSetParams(  WORD    wCard,
                                    WORD    wAverages,
                                    WORD    wWave,
                                    DWORD   dwRange,
                                    WORD    wPointSpacing,
                                    WORD    wPulse,
                                    WORD    wUpdateCount);

////////////////////////////////////////////////////////////////////////////
// QPOTDRAcqStart
// Starts DAU
// card - card index
// avgPts - average points (array of 20 elements) [has to be passed, but not
currently used]
// flags - flags []
QP_API_(HANDLE *) QPOTDRAcqStart( int card, DWORD flags);

////////////////////////////////////////////////////////////////////////////
// QPOTDRRetrieveWaveformSync
// card - card index
// h - Waveform header
// w - Waveform data
// lastTrace - boolean for last trace notification
QP_API_(int) QPOTDRRetrieveWaveformSync( int card, QPOTDRWaveformHeader *h,
QPOTDRWaveformData *w, BOOL lastTrace );

////////////////////////////////////////////////////////////////////////////
// QPOTDRAcqStop
// Stops DAU
// card - card index
QP_API_(int) QPOTDRAcqStop( int card );

////////////////////////////////////////////////////////////////////////////
// QPOTDRFilterLastWaveform
// Filters last waveform using boxcar filter
// card - card index
// w - Waveform data
QP_API_(int) QPOTDRFilterLastWaveform( WORD card, QPOTDRWaveformData *w );
```

## Event Processing

```
////////////////////////////////////////////////////////////////////////////
// QPOTDRSetFASParms
// wCard - Card index
// dLossThreshDB - Loss threshold
// dReflThreshDB - Reflectance threshold
// dBreakThreshDB - Fiber Break Threshold
// nNumBreaks - End set to Nth event >= Fiber Break Threshold.
// nEndAtNthRefl - <= 0 then no change of end >0   then set end to Nth Refl
event
QP_API_(void) QPOTDRSetFASParms(WORD wCard,
                                double dLossThreshDB,
                                double dReflThreshDB,
                                double dBreakThreshDB,
                                LONG   nNumBreaks,
                                LONG   nEndAtNthRefl,
                                double dNthReflThresh);

////////////////////////////////////////////////////////////////////////////
// QPOTDRFAS
// Runs Fiber Analysis Software : finds features and events
// return 0 if success
QP_API_(int) QPOTDRFAS(WORD wCard);

////////////////////////////////////////////////////////////////////////////
// QPOTDRGetEvents
// Fills in event info
// wCard - card index
// events - array of events (call QPOTDRGetEventCount for size)
// element_count - count of elements in events array
// return number of events
QP_API_(int) QPOTDRGetEvents(WORD wCard, DWORD element_count, PQPOTDREVENT
events);

//////////////////////////////////////////////////////////////////////////
// QPOTDRGetEventCount
// wCard - card index
// returns number of events
QP_API_(int) QPOTDRGetEventCount(WORD wCard);

#ifdef __cplusplus
}
#endif

#endif // _ONT_UT_QPOTDR_H_INCLUDED_
```

## Save Trace

```
//////////////////////////////////////////////////////////////////////////
// QPOTDRSaveTrace
// wCard - card index
// fileName - name of the file
// type trace type as defined above
QP_API_(LONG) QPOTDRSaveTrace(WORD wCard, const char *fileName, int type);

#define IOR_MIN 1.300000f
#define IOR_MAX 1.700000f
```

## Load Trace

```
//////////////////////////////////////////////////////////////////////////
// QPOTDRLoadTrace
// Load trace from file
QP_API_(LONG) QPOTDRLoadTrace(const char *filename);
```

## Loaded (Recalled) Trace Operations

```
New API's are used by OEMDemo app.

New API's:
#define QPOTDR_ERROR_FILE_DATA       1      // File data corrupted.
#define QPOTDR_ERROR_MEMORY_ALLOC    2      // Memory allocation error.
```

```
#define QPOTDR_ERROR_FILE_TOO_LARGE  4      // File too large.
#define QPOTDR_ERROR_INVALID_FORMAT  5      // Invalid trace format.
#define QPOTDR_ERROR_CRC             6      // CRC error.
#define QPOTDR_ERROR_DECOMPRESS      8      // Decompression error.
#define QPOTDR_ERROR_CANT_OPEN       9      // Can't open file.

//////////////////////////////////////////////////////////////////////////
// QPOTDRCalculateORLLT
// dAkm  - start position in km
// dBkm - end position in km
// fullTrace - calculate ORL full trace, ignoring dAkm and dBkm
// relToOrigin - relative to origin, otherwise to dAkm
// dwFlags - flags (QPOTDR_SATURATED/QPOTDR_LESS_THAN)
// returns ORL value or QPOTDR_CALC_ERROR
QP_API_(double) QPOTDRCalculateORLLT(double dAkm, double dBkm, BOOL
fullTrace, BOOL relToOrigin, DWORD *dwFlags);

//////////////////////////////////////////////////////////////////////////
// QPOTDRGetEnd2EndLossLT
// returns End-End loss or QPOTDR_CALC_ERROR
QP_API_(double) QPOTDRGetEnd2EndLossLT(void);

//////////////////////////////////////////////////////////////////////////
// QPOTDRGetBSCLT
// returns BSC
QP_API_(float) QPOTDRGetBSCLT(void);

//////////////////////////////////////////////////////////////////////////
// QPOTDRSetBSCLT
// BSC - BSC value to use
// returns BSC
QP_API_(float) QPOTDRSetBSCLT(float bsc);

//////////////////////////////////////////////////////////////////////////
// QPOTDRGetIORLT
// returns IOR
QP_API_(float) QPOTDRGetIORLT(void);

//////////////////////////////////////////////////////////////////////////
// QPOTDRSetIORLT
// IOR - IOR value to use
// returns IOR
QP_API_(float) QPOTDRSetIORLT(float ior);

//////////////////////////////////////////////////////////////////////////
// QPOTDRSaveTraceLT
// returns 0 on success
QP_API_(LONG) QPOTDRSaveTraceLT(const char *fileName, int idType);

//////////////////////////////////////////////////////////////////////////
// QPOTDRGetEventCountLT
// returns number of events
QP_API_(int) QPOTDRGetEventCountLT(void);

//////////////////////////////////////////////////////////////////////////
// QPOTDRGetEventsLT
// Fills in event info
// events - array of events (call QPOTDRGetEventCount for size)
// element_count - count of elements in events array
// return number of events
QP_API_(int) QPOTDRGetEventsLT(DWORD element_count, PQPOTDREVENT events);

//////////////////////////////////////////////////////////////////////////
// QPOTDRFASLT
// Runs Fiber Analysis Software : finds features and events
// return 0 if success
QP_API_(int) QPOTDRFASLT(void);

//////////////////////////////////////////////////////////////////////////
// QPOTDRSetFASParmsLT
// dLossThreshDB - Loss threshold
// dReflThreshDB - Reflectance threshold
// dBreakThreshDB - Fiber Break Threshold
// nNumBreaks - End set to Nth event >= Fiber Break Threshold.
// nEndAtNthRefl - <= 0 then no change of end >0   then set end to Nth Refl
event
QP_API_(void) QPOTDRSetFASParmsLT(double dLossThreshDB,
                                  double dReflThreshDB,
                                  double dBreakThreshDB,
                                  LONG   nNumBreaks,
                                  LONG   nEndAtNthRefl,
                                  double dNthReflThresh);

//////////////////////////////////////////////////////////////////////////
// QPOTDRGetFASParmsLT
// dLossThreshDB - Loss threshold
// dReflThreshDB - Reflectance threshold
// dBreakThreshDB - Fiber Break Threshold
// nNumBreaks - End set to Nth event >= Fiber Break Threshold.
// nEndAtNthRefl - <= 0 then no change of end >0   then set end to Nth Refl
event
QP_API_(void) QPOTDRGetFASParmsLT(double *dLossThreshDB,
                                  double *dReflThreshDB,
                                  double *dBreakThreshDB,
                                  LONG   *nNumBreaks,
                                  LONG   *nEndAtNthRefl,
                                  double *dNthReflThresh);

///////////////////////////////////////////////////////////////////////////
// QPOTDRFilterWaveformLT
// Filters last waveform using boxcar filter
// w - Waveform data
// size - soze of w array
QP_API_(int) QPOTDRFilterWaveformLT(QPOTDRWaveformData *w, DWORD size );

///////////////////////////////////////////////////////////////////////////
// QPOTDRWaveformLT
// Get waveform
// w - Waveform data
// size - size of w array
QP_API_(int) QPOTDRWaveformLT(QPOTDRWaveformData *w, DWORD size );

typedef struct
{
    double dDistanceKM;
    double dWaveLength;
    double dPointSpacing;
    double dDynamicRange;
    double dIOR;
    double dBSC;
    LONG lFiberMode;
    LONG lPulseWidth;
    LONG lPreampType;
    LONG lTime;
    LONG lNumberOfPoints;
    LONG lZKM;
    LONG lAverages;
    BOOL bFiltered;
} QPOTDRWaveformInfo;

///////////////////////////////////////////////////////////////////////////
// QPOTDRWaveformLT
// Get waveform info (wavelength, pulsewidth, ...)
// info - Waveform info pointer
QP_API_(BOOL) QPOTDRGetWaveformInfoLT(QPOTDRWaveformInfo *info);
```

## Live Trace Operations

```
//////////////////////////////////////////////////////////////////////////
// QPOTDRSetIOR
// wCard - card index
// wWaveIDx - wavelength index
// IOR - IOR value to use [IOR_MIN:IOR_MAX]
QP_API_(float) QPOTDRSetIOR(WORD wCard, WORD wWaveIDx, float ior);

//////////////////////////////////////////////////////////////////////////
// QPOTDRGetIOR
// wCard - card index
// wWaveIDx - wavelength index
// returns IOR
QP_API_(float) QPOTDRGetIOR(WORD wCard, WORD wWaveIDx);

#define BSC_MIN -90.0f
#define BSC_MAX -40.0f

//////////////////////////////////////////////////////////////////////////
// QPOTDRSetBSC
// wCard - card index
// wWaveIDx - wavelength index
// BSC - BSC value to use [BSC_MIN:BSC_MAX]
QP_API_(float) QPOTDRSetBSC(WORD wCard, WORD wWaveIDx, float bsc);

//////////////////////////////////////////////////////////////////////////
// QPOTDRGetBSC
// wCard - card index
// wWaveIDx - wavelength index
// returns BSC
QP_API_(float) QPOTDRGetBSC(WORD wCard, WORD wWaveIDx);

//////////////////////////////////////////////////////////////////////////
// QPOTDRGetEnd2EndLoss
// wCard - card index
// returns End-End loss or QPOTDR_CALC_ERROR
QP_API_(double) QPOTDRGetEnd2EndLoss(WORD wCard);

#define QPOTDR_SATURATED 0x00000800
#define QPOTDR_LESS_THAN 0x00000200

//////////////////////////////////////////////////////////////////////////
// QPOTDRCalculateORL
// wCard - card index
// dAkm  - start position in km
// dBkm - end position in km
// fullTrace - calculate ORL full trace, ignoring dAkm and dBkm
// relToOrigin - relative to origin, otherwise to dAkm
// dwFlags - flags (QPOTDR_SATURATED/QPOTDR_LESS_THAN)
// returns ORL value or QPOTDR_CALC_ERROR
QP_API_(double) QPOTDRCalculateORL(WORD wCard, double dAkm, double dBkm,
BOOL fullTrace, BOOL relToOrigin, DWORD *dwFlags);
```

# Chapter 5: API Definitions & Descriptions

## Initialization and Setup API Functions

### Useful Definitions

```
#define MAX_CARDS          8
#define MAX_WAVES          4
#define MAX_LASERS         MAX_WAVES
#define MAX_RANGES         16
#define MAX_PULSES         16
#define MAX_AVERAGES        3
#define MAX_SPACINGS       13
#define MAX_DIST_RANGES    16
```

### QPOTDRInitialize

#### Syntax

OEM_API_(int) QPOTDRInitialize( int cardid, void (*call_back)(char* s) );

#### Parameters

int cardid   Card Number
void (*call_back)(char* s) – Null Pointer (Not Used)

#### Description

Used to initialize the hardware. In addition, the plugin is read, and memory is cleared. This function must be called before using the card.

#### Return Values

| | |
|---|---|
| -1 | Bad Card ID |
| -2 | Unable to get card status |
| -4 | Unable to get Xilinx bitstream |
| -5 | LSR_PWR Initialize Failed |
| -6 | APD_VOLT Initialize Failed |
| -7 | PREAMP Initialize Failed |
| -8 | LSR_LTS Initialize Failed |
| -9 | LSR Initialize Failed |
| -10 | LTS_FREQ Initialize Failed |

| | |
|---|---|
| -11 | LTS_PWR Initialize Failed |
| -12 | PM_VFL Initialize Failed |
| -13 | LSR_COOL Initialize Failed |
| -15 | Xilinx General failure |
| -16 | Xilinx Bad parameter |
| -17 | Xilinx Bad CardID |
| -18 | Xilinx Data Block Start |
| -19 | Xilinx Data Block Middle |
| -20 | Xilinx Data Block End |
| -22 | Xilinx Self Test |
| -23 | EEPROM read error/bad CRC16 |
| -24 | OPT version incorrect |
| -25 | Unable to determine interleave |
| -26 | Memory allocation error |
| -27 | Memory allocation error |
| -28 | Memory allocation error |
| -29 | Memory allocation error |
| -30 | Memory allocation error |
| -31 | Memory allocation error |
| -32 | Memory allocation error |
| -33 | CD OPT structure bad CRC16 |
| 0 | Success |

## QPOTDRGetCards

### Syntax

OEM_API_(WORD) QPOTDRGetCards(WORD *wCardIndexes);

### Parameters

wCardIndexes - card indexes array of type WORD element count = MAX_CARDS

### Description

Returns the card count. Cards refer to the module “Address” in the chassis. If only one module is installed, the Card value will always be “0”. If multiple modules are installed in a cPCI/PCI chassis, then the USER must perform an identification process to determine the card address for all modules. This may be accomplished by calling the QPOTDRInitialize function() for each card address and noting which module’s fans and LaserOn LEDs are active during the function call.

### Return Value

card Count

## QPOTDRGetAvailWaves

### Syntax

OEM_API_(int) QPOTDRGetAvailWaves(int card, float *wavelengths);

### Parameters

Int card - Card Number
float *wavelengths – array of floats, element count = MAX_WAVES

### Description

This function fills the wavelengths array with the available wavelengths for the module addressed by “card”. The array maximum number of elements is defined as MAX_WAVES, but the module may contain less than this maximum. The “QPOTDRGetMaxWaves()” function returns the actual number of wavelengths (as a count) which may be used to prevent illegal array access.

### Return Value

Array of floating point Wavelength values

## QPOTDRGetMaxWaves

### Syntax

OEM_API_(int) QPOTDRGetMaxWaves(int card);

### Parameters

Int card - Card Number

### Description

This function returns the actual number of wavelengths (as a count), for the addressed module (card), which may be used to prevent illegal array access. The array of waves returned by the above function (QPOTDRGetAvailWaves()) contains a number of entries determined by the return value of this function.

### Return Value

An integer count of the number of waves (lasers) in the addressed module. It represents a zero-based index to the last valid element in the array of waves.

## QPOTDRGetAvailRanges

### Syntax

OEM_API_(int) QPOTDRGetAvailRanges( int card, float wave, float *ranges);

### Parameters

Int card - Card Number
float *wave – Wavelength Index
float *ranges - array of floats, element count = MAX_RANGES

### Description

This function fills the ranges array with the available ranges for the module addressed by "card". The array maximum number of elements is defined as MAX_RANGES, but the module may contain less than this maximum. The "QPOTDRGetMaxRanges()" function returns the actual number of Ranges (as a count) which may be used to prevent illegal array access.

### Return Value

Array of floating point Range values in kilometers (km)

## QPOTDRGetMaxRanges

### Syntax

OEM_API_(int) QPOTDRGetMaxRanges( int card, float wave);

### Parameters

Int card - Card Number
float wave – The wavelength for which the number of actual available Ranges is queried.

### Description

This function returns the number of ranges supported by the wavelength specified in the "wave" parameter. This count represents the actual number of ranges stored in the array initialized by the above function (QPOTDRGetAvailRanges()) for the specified wavelength, and may be used to prevent illegal array access.

### Return Value

An integer count of the number of ranges for the specified wavelength for the addressed module. It represents a zero-based index to the last valid element in the array of Ranges.

## QPOTDRGetAvailPulses

### Syntax

OEM_API_(int) QPOTDRGetAvailPulses( int card, float wave, DWORD *pulsewidths);

### Parameters

Int card - Card Number
float *wave – Wavelength Index
DWORD *pulsewidths - array of DWORDS, element count = MAX_PULSES

### Description

This function fills the pulsewidths array with the available pulsewidths for the specified wavelength in the module addressed by "card". The array maximum number of elements is defined as MAX_PULSES, but the module may contain less than this maximum. The "QPOTDRGetMaxPulses()" function returns the actual number of pulsewidths (as a count) which may be used to prevent illegal array access.

### Return Value

Array of DWORD Pulsewidth values in nanoseconds (ns).

## QPOTDRGetMaxPulses

### Syntax

OEM_API_(int) QPOTDRGetMaxPulses( int card, float wave);

### Parameters

Int card - Card Number
float wave – The wavelength for which the number of actual available pulsewidths is queried.

### Description

This function returns the actual number of pulsewidths (as a count), for the specified wavelength (wave) and module (card), which may be used to prevent illegal array access. This count represents the actual number of pulsewidths stored in the array initialized by the above function (QPOTDRGetAvailPulses()) for the specified wavelength, and may be used to prevent illegal array access.

### Return Value

An integer count of the number of pulsewidths for the specified wavelength for the addressed module. It represents a zero-based index to the last valid element in the array of pulsewidths

## QPOTDRGetAvailAverages

### Syntax

OEM_API_(int) QPOTDRGetAvailAverages( int card, float wave, DWORD *averages);

### Parameters

Int card - Card Number
float wave – The wavelength for which the number of available averaging types is queried.
DWORD *averages – Array of Averaging Types. Element count = MAX_AVERAGES.

- Averaging types are defined FAST, MEDIUM and SLOW and typically have values of:

| | |
|---|---|
| • FAST – | 4096 Hardware averages |
| • MEDIUM – | 32768 Hardware averages |
| • SLOW – | 262144 Hardware averages |

The DEMO Application offers these three averaging types. The Demo application divides the above counts by 128 since the **QPOTDRAckSetParams()** API requires the number of averages to be expressed as a number of Fastscans (128 HW averages = 1 Fastscan).

The user may utilize these averaging types (but must divide the array value by 128 to obtain the number of Fastscans), or may pass a discrete number of averages to the **QPOTDRAcqSetParams()** function.

### Description

This function fills the averages array with the available averaging types for the specified wavelength and addressed card. The array maximum number of elements is defined as MAX_AVERAGES. The “QPOTDRGetMaxAverages()” function returns the actual number of averaging types (as a count) which may be used to prevent illegal array access.

### Return Value

Array of DWORDS representing the available averaging types.

## QPOTDRGetMaxAverages

### Syntax

OEM_API_(int) QPOTDRGetMaxAverages( int card, float wave);

### Parameters

Int card  -  Card Number
float wave – The wavelength for which the maximum number of averaging types is queried.

### Description

This function returns the actual number of averaging types as a count, for the specified wavelength (wave) and module (card), which may be used to prevent illegal array access. This count represents the actual number of averaging types stored in the array initialized by the above function (QPOTDRGetAvailAverages()) for the specified wavelength, and may be used to prevent illegal array access.

### Return Value

An integer count of the number of averaging types for the specified wavelength for the addressed module. It represents a zero-based index to the last valid element in the array of averaging types.

## QPOTDRGetAvailSpacings

### Syntax

OEM_API_(int) QPOTDRGetAvailSpacings( int card, float* point_spacings);

### Parameters

Int card  -  Card Number
Float* point_spacings – Array of floats containing the available point spacings in meters.  Element count = MAX_SPACINGS.
For typical modules, this array will contain the following point spacings:
{ 16.0, 8.0, 4.0, 2.0, 1.0, 0.50, 0.25, 0.125 }

### Description

This function fills the spacings array with the available point spacings for the specified wavelength (wave) and module (card). The array maximum number of elements is defined as MAX_SPACINGS, but the module may contain less than this maximum. The "QPOTDRGetMaxPointSpacings()" function returns the actual number of point spacings (as a count), which may be used to prevent illegal array access.

### Return Value

Array of floats of point_spacings (in meters) for the addressed module.

## QPOTDRGetMaxPointSpacings

### Syntax

OEM_API_(int) QPOTDRGetMaxPointSpacings( int card);

### Parameters

Int card - Card Number

### Description

This function returns the actual number of point spacings as a count, for the specified module (card), which may be used to prevent illegal array access. This count represents the actual number of point spacings stored in the array initialized by the above function (QPOTDRGetAvailSpacings()) for the specified module, and may be used to prevent illegal array access.

### Return Value

An integer count of the number of point spacings for the addressed module. It represents a zero-based index to the last valid element in the array of point_spacings.

## QPOTDRGetDefaultIOR

### Syntax

OEM_API_(float) QPOTDRGetDefaultIOR( WORD wCard, float wave);

### Parameters

WORD wCard - Card Number
float wave – The wavelength for which the Default IOR (Index of Refraction) is being queried.

### Description

This function returns the floating point value for the default IOR for the specified wavelength.

### Return Value

float DefaultIOR.

## QPOTDRGetDefaultBSC

### Syntax

OEM_API_(float) QPOTDRGetDefaultBSC( WORD wCard, float wave);

### Parameters

WORD wCard  -  Card Number
float wave – The wavelength for which the default BSC (Backscatter Coefficient) is being queried.

### Description

This function returns the floating point value for the default BSC (Backscatter Coefficient) for the specified wavelength.

### Return Value

float DefaultBSC.

## QPOTDRGetFiberType

### Syntax

OEM_API_(BYTE) QPOTDRGetFiberType( WORD wCard, float wave);

### Parameters

WORD wCard  -  Card Number
float wave – The wavelength for which the FiberType is being queried.

### Description

This function returns a byte representing the fiber type (SM or MM) for the specified wavelength.

### Return Value

BYTE FiberType ( 1 = SM, 0 = MM )

## QPOTDRDataCollectInfo

### Syntax

OEM_API_(VOID) QPOTDRDataCollectInfo( WORD card, QPOTDRPluginData *pPluginData);

### Parameters

WORD card - Card Number
pPluginData – Pointer to QPOTDRPluginData structure.

### Description

This function fills the data structure "QPOTDRPluginData" with useful data for the addressed card (module).

### Return Value

void

## Plugin Data Structures

The following data structures contain useful information for the applicable module. This information is retrieved from the module's read-only-memory during the QPOTDRInitialize function call. The "PluginInfo" structure contains manufacturing information and the "PluginData" structure contains definitions of the DAU (Data Acquisition Unit) hardware capabilities.

```
typedef struct _PluginInfo
{
        char ModelNumber[32];
        char SerialNumber[32];
        char szLastCalibrationDate[16];
        char szPartNumber[16];
        char szModel[16];
        char szManufacturer[20];
        char opt_file_version[10];
        char dll_version[22];

} PluginInfo, *PPluginInfo;

// Plugin - DAU specific info
typedef struct
{

        DWORD dwMaxDataPoints;
        float fBaseReso;
        WORD wRevision;
        PluginInfo pluginInfo;
        DWORD dwMaxFastAvg;
        DWORD dwFastScanCount;

} CMA5000PluginData;
```

## Data Acquisition Control Functions

A data acquisition cycle is initiated by first setting up the acquisition parameters using the QPOTDRAcqSetParams() function. The acquisition parameters are retrieved from the USER arrays using the various "GetAvail…()" and GetMax…() functions defined in section 5.1.

## QPOTDRAcqSetParams

### Syntax

```
OEM_API_(int) QPOTDRAckSetParams( WORD      wCard,
                                  WORD      wAverages,
                                  WORD      wWave,
                                  DWORD     dwRange,
                                  WORD      wPointSpacing,
                                  WORD      wPulse,
                                  WORD      wFilter,
                                  WORD      wUpdateCount );
```

### Parameters

WORD wCard  -  Card Number

WORD wAverages – Number of Fastscans (1 Fastscan = 128 averages)

WORD wWave – Wavelength for Test, as an index from USER table of waves

DWORD dwRange – Range for Test. Expressed as a number of data points, with "Pointspacing" defining the distance between consecutive data points. For example, to request a 125km Range, at 1.0m Pointspacing, the USER would pass a value of 125,000 to this function.

WORD wPointSpacing – The distance between consecutive data points, as an index from USER array of PointSpacings.
The Pointspacing array should look like:
{16.0, 8.0, 4.0, 2.0, 1.0, 0.50, 0.250, 0.125 };

WORD wPulse – The pulsewidth as an index from USER array of pulses.

WORD wUpdateCount – This parameter defines when the data will be presented in a format suitable for display (log converted). The USER assigned data buffer will be populated with the log-converted data at the specified Fastscan count contained in wUpdateCount. The user may request that the data be available for display updates in multiples of "Fastscan" (128 averages). This allows a reduction of system overhead, and acquisition speed, by limiting the number of conversions and plotting functions in a given test cycle.

### Description
This function sets up all of the Data acquisition parameters for an ensuing Data acquisition cycle.

### Return Value

| | |
|---|---|
| 0 | - Success |
| -1 | - bad card index |
| -2 | - bad number of averages |
| -3 | - bad wave index |
| -4 | - bad pulse index |
| -5 | - bad range |
| -6 | - bad point spacing index |

## QPOTDRAcqStart

### Syntax
OEM_API_(HANDLE *) QPOTDRAcqStart( int card, DWORD flags);

### Parameters
Int card - Card Number
DWORD flags – Flags used in the Data Acquisition. Three flags used to specify the following:

REAL_TIME_SCAN – This flag is set TRUE if the USER wants to acquire data in a non-averaged fashion. The data is refreshed, based on USER parameter selection, in a non-accummulated manner. This feature allows Real Time monitoring of the fiber connection with rapid updates of acquired data.

GAIN_SPLICE_ON – This flag is set TRUE if the USER desires maximum dynamic range. If the current pulsewidth and current Range ( > 30 km) support the gain splice feature, then Gain splicing woll run.

LIVE_FIBER_DETECT – This flag is set TRUE if the USER wants the Acquisition function to check for live traffic on the Fiber Under Test, before emitting Laser energy into the fiber. If live traffic is detected on the fiber under test, the USER requested acquisition is aborted.

### Description
This function initiates a DAU Data Acquisition cycle using the parameters set by the call to QPOTDRAckSetParams().

### Return Value

This function returns a pointer to an array of HANDLEs. HANDLEs to notification Events are as follows

Average Complete
New waveform
Life fiber / critical error

Check return code of WaitForMultipleObjects to identify event as following

| | |
|---|---|
| WAIT_OBJECT_0 | = Average Complete |
| WAIT_OBJECT_0 + 1 | = New waveform |
| WAIT_OBJECT_0 + 2 | = Life fiber / critical error |

## QPOTDRRetrieveWaveformSync

### Syntax

QP_API_(VOID) QPOTDRRetrieveWaveformSync( int card, QPOTDRWaveformHeader *h, QPOTDRWaveformData *w, BOOL lastTrace );

### Parameters

int card - Card Number
QPOTDRWaveformHeader *h - Pointer to OTDRWaveformHeader Data structure.
QPOTDRWaveformData *w – Pointer to Data point buffer.
BOOL lastTrace – TRUE if buffer contains LAST (averaging-complete) Data set.

### Description

This function fills the WaveformHeader and WaveformData structures. The WaveformHeader structure contains settings and other useful information reqarding the acquisition data, and the WaveformData array contains all the acquired data in logarithmic format.

### Return Value

void

## QPOTDRRetrieveWaveformSyncEx

### Syntax

QP_API_(VOID) QPOTDRRetrieveWaveformSyncEx( int card, QPOTDRWaveformHeader *h, QPOTDRWaveformData *w, double start_meters, double end_meters,  BOOL lastTrace);

### Parameters

int card  -  Card Number
QPOTDRWaveformHeader *h - Pointer to OTDRWaveformHeader Data structure.
QPOTDRWaveformData *w – Pointer to Data point buffer.
double start_meters – start in meters
double end_meters – end in meters
BOOL lastTrace – TRUE if buffer contains LAST (averaging-complete) Data set.

### Description

This function fills the WaveformHeader and WaveformData structures.  The WaveformHeader structure contains settings and other useful information reqarding the acquisition data, and the WaveformData array contains all the requested data in logarithmic format.  The USER specifies a subset of the entire data acquisition results, by requesting data from start_meters to end_meters.

### Return Value

Void

## QPOTDRAcqStop

### Syntax

QP_API_(int) QPOTDRAcqStop( int card);

### Parameters

int card  -  Card Number

### Description

This function halts the current acquisition cycle when the current hardware accumulation cycle has completed. It may take up to approximately 2 seconds for a hardware accumulation cycle to complete.

### Return Value

FALSE(0)=SUCCESS
-1=FAILURE (INVALID CARD)

## QPOTDRFilterLastWaveform

### Syntax

QP_API_(int) QPOTDRFilterLastWaveform( WORD card, QPOTDRWaveformData *w);

### Parameters

WORD card - Card Number
QPOTDRWaveformData – Pointer to filtered data buffer

### Description

This function executes a filter algorithm on the last acquired data set and returns the resultant filtered data into the USER buffer.

### Return Value

FALSE(0)

## Trace Save/Load APIs (GR-106 or SR-4731

### Constants and Defines:

**// Span calculation error**

```
#define QPOTDR_CALC_ERROR -1.23456E31 // Error code that is returned in the event
                                      // the Span Loss can not be computed

#define TRACETYPE_SR4731 1          // Specifies Trace Save type as SR-4731
                                    // (Bellcordia)

#define TRACETYPE_SR196  2          // Specifies Trace Save type as GR-196
                                    // (Bellcore)

#define IOR_MIN 1.300000f           // Minimum allowable IOR value

#define IOR_MAX 1.700000f           // Maximum allowable IOR value

#define BSC_MIN -90.0f              // Minimum allowable Backscatter Coefficient
                                    // value

#define BSC_MAX -40.0f              // Maximum allowable Backscatter Coefficient
                                    // value
```

**// Reflectance Flags:**

```
#define QPOTDR_SATURATED 0x00000800    // Reflectance Saturated flag

#define QPOTDR_LESS_THAN 0x00000200    // Reflectance "<" flag
```

**// Note:** The Reflectance Flags indicate inaccuracies in the calculated Reflectance value due to pulsewidth clamping, reflectance saturation, or insufficient number of data points per pulsewidth.

### Trace File SAVE APIs

**Error Codes –**

```
#define QPOTDR_ERROR_MEMORY_ALLOC     2  // Memory allocation error.

#define QPOTDR_ERROR_INVALID_FORMAT   5  // Invalid trace format.

#define QPOTDR_ERROR_COMPRESS         7  // Compression error.

#define QPOTDR_ERROR_CANT_OPEN        9  // Can't open file.

#define QPOTDR_ERROR_WRITING          10 // Write error.
```

## QPOTDRSaveTrace

### Syntax:

QP_API_(LONG) QPOTDRSaveTrace(WORD wCard, const char *fileName, int type);

### Parameters

| | |
|---|---|
| WORD wCard - | Card Number - Card or Slot number |
| CONST CHAR * | filename – Pathname specifier (full path) |
| int type | Type of Trace (TRACETYPE_SR4731 or or TRACETYPE_SR196) |

### Description

This API saves the internal Trace Array and parameters into a file of the type specified by the user in the “type” parameter, to a filename specified in the “filename” parameter. The “filename” parameter must contain a full path specifier (eg: Drive\folder\filename).

### Return Value:

ZERO for success. Non-Zero value for error (see “**Error Codes –**” on page 5-17).

## QPOTDRSaveTraceLT

API to save/re-save a loaded trace

### Syntax:

QP_API_(LONG) QPOTDRSaveTraceLT(const char *fileName, int idType);

### Parameters:

| | |
|---|---|
| CONST CHAR * | filename – Pathname specifier (full path) |
| int idType | Type of Trace (TRACETYPE_SR4731 or TRACETYPE_SR196) |

### Description

This API is used to save a trace file that was previously loaded and operated on.

### Return Value:

ZERO to indicate success. Non Zero to indicate failure (see defines)

## Trace File LOAD APIs

The following APIs operate on Loaded (retrieved from media) trace files that have been saved to media using the QPOTDRSaveTrace() API. These API names all contain an “LT” at the end of the API function name to indicate that they operate on a “Loaded Trace”.

The following series of API definitions utilize the following Defines to indicate errors returned by the functions:

**Error Codes –**

```
#define QPOTDR_ERROR_FILE_DATA          1  // File data corrupted.
#define QPOTDR_ERROR_MEMORY_ALLOC       2  // Memory allocation error.
#define QPOTDR_ERROR_FILE_TOO_LARGE     4  // File too large.
#define QPOTDR_ERROR_INVALID_FORMAT     5  // Invalid trace format.
#define QPOTDR_ERROR_CRC                6  // CRC error.
#deifne QPOTDR_ERROR_DECOMPRESS         8  // Decomprssion error.
#define QPOTDR_ERROR_CANT_OPEN          9  // Can’t open file.
```

## QPOTDRLoadTrace

### Syntax:

QP_API_(LONG) QPOTDRLoadTrace(const char *filename);

### Parameters

| | |
|---|---|
| const char *filename | The full pathname of the desired Trace file. |

### Description

This API loads into memory the specified trace file.

### Return Value:

ZERO for success. Non-Zero value for error (see “**Error Codes –**” above).

## Trace Opeations:

### QPOTDRSetIOR

#### Syntax:

QP_API_(float) QPOTDRSetIOR(WORD wCard, WORD wWaveIDx, float ior);

#### Parameters

| | |
|---|---|
| WORD wCard - | Card Number - Card or Slot number |
| WORD wWaveIDx | Wavelength index, as an index from USER table of waves |
| FLOAT ior | OR Value, specified to 6 places (eg: 1.467700) |

#### Description

This API allows the user to set the IOR for the currently accessible trace (latest trace collected) for the purposes of calculating the true distance between data points.

#### Return Value:

Float IOR

The following equation may be used to determine true distance between data points:

dDxKM = (ResoNominalM * dT6SpeedOfLight) / (1.0E05 * 2000.0 * dIOR)

where:

| | | |
|---|---|---|
| dDxKM | = | distance in KM between data points. |
| ResoNominalM | = | Nominal Resolution in Meters. |
| dT6SpeedOfLight | = | speed of light in vaccuum = 2.997925 E05 Km per second . |
| dIOR | = | IOR (e.g., 1.460000). |

### QPOTDRSetIORLT

#### Syntax:

QP_API_(float) QPOTDRSetIORLT(float ior);

**Parameters:**

Float IOR (eg: 1.467700)

**Description**

This API allows the user to set the IOR value for the currently loaded trace (recalled from media).

**Return Value:**

FLOAT IOR Value (eg: 1.467700)

## QPOTDRGetIOR

**Syntax:**

QP_API_(float) QPOTDRGetIOR(WORD wCard, WORD wWaveIDx);

**Parameters**

| | |
|---|---|
| WORD wCard - | Card Number - Card or Slot number |
| WORD wWaveIDx | Wavelength index, as an index from USER table of waves |

**Description**

This API allows the user to retrieve the IOR value for the currently accessible trace (latest trace collected).

**Return Value:**

FLOAT IOR Value (eg: 1.467700)

## QPOTDRGetIORLT

**Syntax:**

QP_API_(float) QPOTDRGetIORLT(void);

**Parameters:**

None

**Description**

This API allows the user to retrieve the IOR value for the currently loaded trace (recalled from media).

**Return Value:**

FLOAT IOR Value (eg: 1.467700)

## QPOTDRSetBSC

### Syntax:

QP_API_(float) QPOTDRSetBSC(WORD wCard, WORD wWaveIDx, float bsc);

### Parameters

| | |
|---|---|
| WORD wCard - | Card Number - Card or Slot number |
| WORD wWaveIDx | Wavelength index, as an index from USER table of waves |
| FLOAT bsc | Backscatter Coefficient Value (eg: -75.00) |

### Description

This API allows the user to set the Backscatter Coefficient value for the currently accessible trace (latest trace collected). This value provides information required to fully qualify Reflectance Values in the Event Analysis functions.

### Return Value:

Success : (float) bsc
Failure : BSC_MIN + .000001 (for error condition, e.g.: invaliad Card Number)

## QPOTDRSetBSCLT

### Syntax:

QP_API_(float) QPOTDRSetBSCLT(float bsc);

### Parameters:

| | |
|---|---|
| FLOAT bsc | Backscatter Coefficient Value (eg: -75.00) |

### Description

This API allows the user to set the Backscatter Coefficient value for the currently loaded trace (recalled from media). This value provides information required to fully qualify Reflectance Values in the Event Analysis functions.

### Return Value:

| | |
|---|---|
| FLOAT | Backscatter Coefficient Value that was passed (eg: -75.00) |

## QPOTDRGetBSC

### Syntax:

QP_API_(float) QPOTDRGetBSC(WORD wCard, WORD wWaveIDx);

### Parameters

| | |
|---|---|
| WORD wCard - | Card Number - Card or Slot number |
| WORD wWaveIDx | Wavelength index, as an index from USER table of waves |

### Description

This API allows the user to retrieve the BSC value for the currently accessible trace (latest trace collected).

### Return Value:

Returns the BSC value

## QPOTDRGetBSCLT

### Syntax:

QP_API_(float) QPOTDRGetBSCLT(void);

### Parameters:

None

### Description

This API returns the Backscatter Coefficient factor for the currently loaded trace file (recalled from media).

### Return Value:

Returns the BSC value

## QPOTDRCalculateORL

### Syntax:

```
QP_API_(double) QPOTDRCalculateORL(WORD wCard,
                                   double dAkm,
                                   double dBkm,
                                   BOOL fullTrace,
                                   BOOL relToOrigin,
                                   DWORD *dwFlags);
```

### Parameters

| | | |
|---|---|---|
| WORD | wCard - | Card Number - Card or Slot number |
| Double | dAkm | start distance in km |
| Double | dBkm | end distance in km |
| BOOL | fullTrace | Set to TRUE if ORL value to be calculated from the entire trace |
| BOOL | relToOrigin | Set to TRUE if ORL value is relative to Origin of trace, if FALSE then uses start distance (dAkm) |
| DWORD | *dwFlags | Pointer to returned Flags |

### Description

This API returns the calculated ORL value (in dB) based on the requirements passed in the input parameter list. The dwFlags parameter holds any pertinent information about whether the value that is returned should be considered somewhat inaccurate based on whether any Saturated Events contributed to the result, or whether any event was clamped, or whether the trace was collected at too coarse a point spacing (4 data points per pulsewidth minimum).

### Return Value:

Returns ORL value or QPOTDR_CALC_ERROR

## QPOTDRCalculateORLLT

### Syntax:

```
QP_API_(double) QPOTDRCalculateORLLT(double dAkm,
                                     double dBkm,
                                     BOOL fullTrace,
                                     BOOL relToOrigin,
                                     DWORD *dwFlags);
```

### Parameters

| | |
|---|---|
| Double dAkm | start distance in km |
| Double dBkm | end distance in km |
| BOOL fullTrace | Set to TRUE if ORL value to be calculated from the entire trace |
| BOOL relToOrigin | Set to TRUE if ORL value is relative to Origin of trace, if FALSE then uses start distance (dAkm) |
| DWORD *dwFlags | Pointer to returned Flags |

### Description

This API returns the calculated ORL value (in dB) based on the requirements passed in the input parameter list, of the specified trace file that has been loaded from media. The dwFlags parameter holds any pertinent information about whether the value that is returned should be considered somewhat inaccurate based on whether any Saturated Events contributed to the result, or whether any event was clamped, or whether the trace was collected at too coarse a point spacing (4 data points per pulsewidth minimum).

### Return Value:

Returns ORL value or QPOTDR_CALC_ERROR

## QPOTDRGetEnd2EndLoss

### Syntax:

```
QP_API_(double) QPOTDRGetEnd2EndLoss(WORD wCard);
```

### Parameters

WORD wCard - Card Number - Card or Slot number

### Description

This API allows the user to retrieve the End to End Loss value for the current trace.

### Return Value:

Returns End-End loss or QPOTDR_CALC_ERROR

## QPOTDRGetEnd2EndLossLT

### Syntax:

QP_API_(double) QPOTDRGetEnd2EndLossLT(void);

### Parameters:

None

### Description

This API retrieves the calculated End to End Loss of the currently loaded Trace file (recalled from media)

### Return Value:

Returns End-End loss or QPOTDR_CALC_ERROR

## QPOTDRFilterWaveformLT

### Syntax:

QP_API_(int) QPOTDRFilterWaveformLT(QPOTDRWaveformData *w, DWORD size );

### Parameters:

| | |
|---|---|
| QPOTDRWaveformData *w | - Waveform Data structure |
| DWORD size | - size of array |

### Description

This API applies a boxcar filter to the data point set pointed to by the *w pointer. The parameter "size" passes the number of data points to the function.

### Return Value:

ZERO indicates success.

## QPOTDRWaveformLT

### Syntax:

QP_API_(int) QPOTDRWaveformLT(QPOTDRWaveformData *w, DWORD size );

### Parameters:

| | |
|---|---|
| QPOTDRWaveformData *w | - Waveform Data structure |
| DWORD size | - size of array |

### Description

This API fetches the waveform data from the loaded trace file into the QPOTDRWaveformData array. The size of the arrau is specified by the “size” parameter.

### Return Value:

ZERO indicates success.

## QPOTDRWaveformInfoLT

### Syntax:

QP_API_(int) QPOTDRWaveformInfoLT(QPOTDRWaveformData *info);

### Parameters:

QPOTDRWaveformInfo *info - Waveform Info Data structure

### Description

This API fetches the waveform information data from the loaded trace file into the QPOTDRWaveformInfo array. This info includes the wavelength, pulsewidth etc., but does NOT include the actual data points.

### Return Value:

ZERO indicates success.

## Additional Event Table Functionality:

### QPOTDRSetFASParms

#### Syntax

```
QP_API_(Void) QPOTDRSetFASParms(QP_API_(void)
QPOTDRSetFASParms(WORD wCard,
                              double dLossThreshDB
                              double dReflThreshDB,
                              double dBreakThreshDB,
                              LONG   nNumBreaks,
                              LONG   nEndAtNthRefl,
                              double dNthReflThresh);
```

#### Parameters

WORD wCard - Card Number
dLossThreshDB - Loss threshold – Losses GREATER than this threshold are reported
dReflThreshDB - Reflectance threshold – Reflectances GREATER than this threshold are reported.
dBreakThreshDB - Fiber Break Threshold - First Loss to exceed Fiber Break is classified as last (end) event, unless nNumBreaks is used.
nNumBreaks - End set to Nth event >= Fiber Break Threshold.
nEndAtNthRefl - <= 0 then no change of end >0 then set end to Nth Refl event

#### Description

This API allows the user to set the Event Analysis theshold for the currently accessable trace (latest trace collected).

#### Return Value

Void

### QPOTDRSetFASParmsLT

#### Syntax:

```
QP_API_(void) QPOTDRSetFASParmsLT(double dLossThreshDB,

                              double dReflThreshDB,
                              double dBreakThreshDB,
                              LONG   nNumBreaks,
LONG   nEndAtNthRefl,
double dNthReflThresh);
```

### Parameters:

| | |
|---|---|
| double | dLossThreshDB - Loss threshold – Losses GREATER than this threshold are reported |
| double | dReflThreshDB - Reflectance threshold – Reflectances GREATER than this threshold are reported. |
| double | dBreakThreshDB - Fiber Break Threshold - First Loss to exceed Fiber Break is classified as last (end) event, unless nNumBreaks is used. |
| long | nNumBreaks - End set to Nth event >= Fiber Break Threshold. |
| long | nEndAtNthRefl - <= 0 then no change of end >0 then set end to Nth Refl event |
| double | dNthReflThresh – Reflectance threshold for Nth end event |

### Description

This API allows the user to set the Event Analysis Thresholds for the currently loaded (recalled from media) trace file.

### Return Value:

None

## QPOTDRFAS

### Syntax

QP_API_(int) QPOTDRFAS( WORD wCard);

### Parameters

WORD wCard - Card Number

### Description

This function executes the Fiber Analysis algorithms on the last acquired data set returned by the QPOTDRRetrieveWaveformSync function, and creates a Table of Events which the USER may request.

### Return Value

Int – 0 for Success

## QPOTDRFASLT

### Syntax:

QP_API_(int) QPOTDRFASLT(void);

### Parameters:

None

### Description

This API executes the Fiber Analysis algorithm on the current loaded trace file (recalled from media).

### Return Value:

Int – ZERO for success

## QPOTDRGetEvents

### Syntax

QP_API_(int) QPOTDRGetEvents( WORD wCard, DWORD element_count, PQPOTDREVENT events );

### Parameters

WORD wCard - Card Number
element_count - count of the number of elements in the Event array
events - array of events (call QPOTDRGetEventCount for size)

### Description

This function fills the array "events" with the requested number of events detected by the Fiber Analysis function call.

### Return Value

Int – Number of events

## QPOTDRGetEventsLT

**Syntax:**

QP_API_(int) QPOTDRGetEventsLT(DWORD element_count, PQPOTDREVENT events);

**Parameters:**

| | |
|---|---|
| DWORD | element_count; |
| PQPOTDREVENT | Pointer to Event Data structure. |

**Description**

This API returns the event count, and a pointer to the recalled trace file’s Event Table structure.

**Return Value:**

Int – ZERO for success

## QPOTDRGetEventCount

**Syntax**

QP_API_(int) QPOTDRGetEventCount( WORD wCard );

**Parameters**

WORD wCard  -  Card Number

**Description**

This function returns the number of events for the current Event Table.

**Return Value**

Int – Number of events

**Note**

The 0th event is a “Launch” event and contains no meaningful data for the user.

## QPOTDRGetEventCountLT

**Syntax:**

QP_API_(int) QPOTDRGetEventCountLT(void);

### Parameters:

None

### Description

This API retrieves the number of events in the event table of the currently loaded trace file (recalled from media).

### Return Value:

Int – number of events in the trace file's event table

## QPOTDRGetFASParmsLT

### Syntax:

```
QP_API_(void) QPOTDRGetFASParmsLT(double *dLossThreshDB,
                                  *double dReflThreshDB,
                                  *double dBreakThreshDB,
                                  *LONG   nNumBreaks,
                                  *LONG   nEndAtNthRefl,
                                  *double dNthReflThresh);
```

### Parameters:

| | |
|---|---|
| double | *dLossThreshDB - Loss threshold – Losses GREATER than this threshold are reported |
| double | *dReflThreshDB - Reflectance threshold – Reflectances GREATER than this threshold are reported. |
| double | *dBreakThreshDB - Fiber Break Threshold - First Loss to exceed Fiber Break is classified as last (end) event, unless nNumBreaks is used. |
| long | *nNumBreaks - End set to Nth event >= Fiber Break Threshold. |
| long | *nEndAtNthRefl - <= 0 then no change of end >0 then set end to Nth Refl event |
| double | *dNthReflThresh – Reflectance threshold for Nth end event |

### Note

Parameters are pointers to the Thresholds and are filled by the API.

### Description

This API allows the user to get the Event Analysis Thresholds for the currently loaded (recalled from media) trace file.

### Return Value:

None

# Chapter 6: Demo Application

The OEMDEMO Application is a tool provided to assist the USER in the design of a USER OTDR Application program. The Application exercises all of the published API's and is provided as a Microsoft™ Visual C/C++ project (Version 6.0).

The software is written in the C Language, and source code is provided to promote rapid development by the user. The Demo Application is provided as a Visual C Project, with all source code, header files and utilities and is located in the OEMDemo folder in the installation space on your hard drive. It is recommended that this Demo Application be used to aid in the development of the User Application.

## OEMDEMO Application

1. Click the **Shortcut to OEMDemo** icon on the Windows desktop, the OEMDemo screen appears.

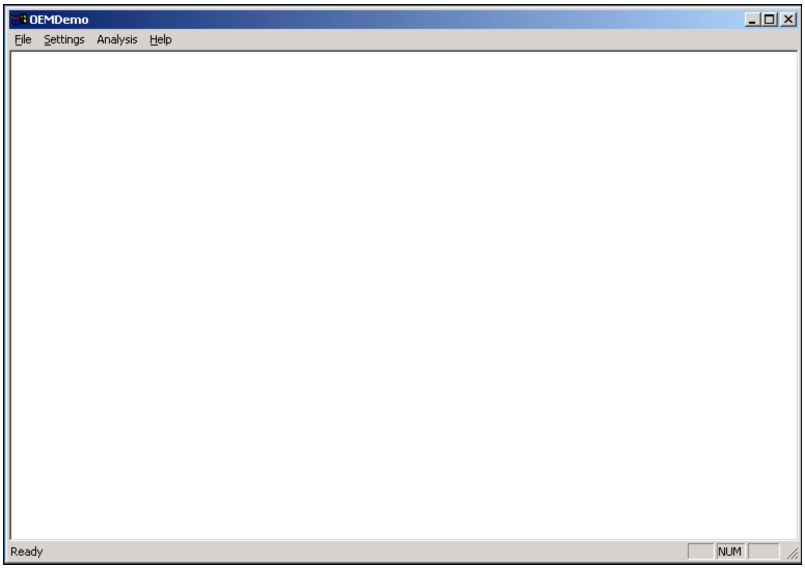


*Figure 6-1: OEMDemo Screen*

1. Select **Initialize** under Settings menu (or press F5)

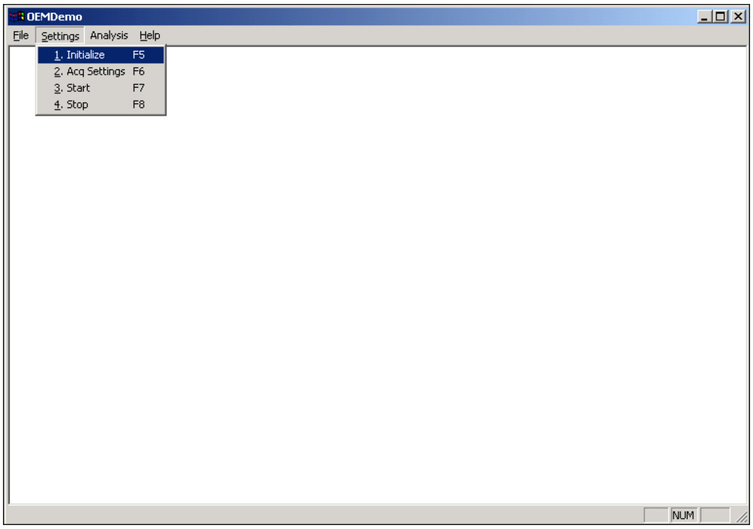
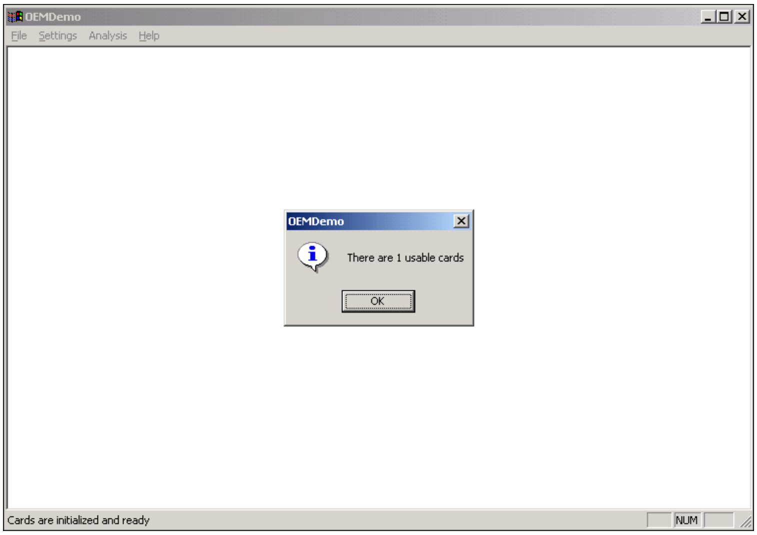


*Figure 6-2: Settings Menu*

2. During initialization observe status messages on the status bar. After initialization, the number of usable cards is reported. Click **OK** to continue.



There are 1 usable cards

Cards are initialized and ready

*Figure 6-3: Number of Usable Cards dialog*

3. Select **Acq Settings** under Settings menu (or press F6)

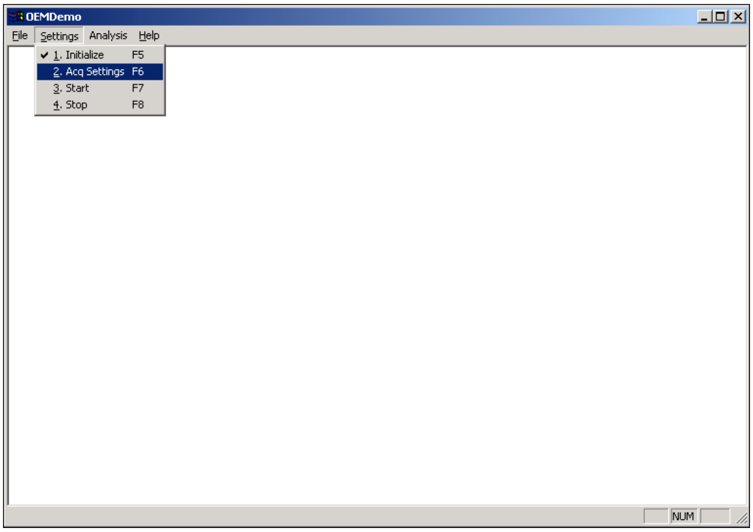
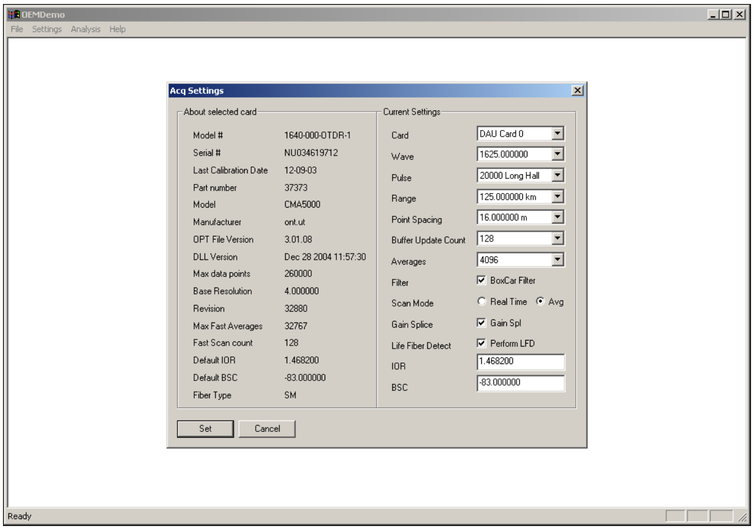


*Figure 6-4: Select "Acq Settings"*

4. Set the desired settings in Acq Settings dialog and then click **Set** to continue.



About selected card

| | |
|---|---|
| Model # | 1640-000-OTDR-1 |
| Serial # | NU034619712 |
| Last Calibration Date | 12-09-03 |
| Part number | 37373 |
| Model | CMA5000 |
| Manufacturer | ont.ut |
| OPT File Version | 3.01.08 |
| DLL Version | Dec 28 2004 11:57:30 |
| Max data points | 260000 |
| Base Resolution | 4.000000 |
| Revision | 32880 |
| Max Fast Averages | 32767 |
| Fast Scan count | 128 |
| Default IOR | 1.468200 |
| Default BSC | -83.000000 |
| Fiber Type | SM |

Current Settings

| | |
|---|---|
| Card | DAU Card 0 |
| Wave | 1625.000000 |
| Pulse | 20000 Long Hall |
| Range | 125.000000 km |
| Point Spacing | 16.000000 m |
| Buffer Update Count | 128 |
| Averages | 4096 |
| Filter | ✓ BoxCar Filter |
| Scan Mode | Real Time [✓ Avg] |
| Gain Splice | ✓ Gain Spl |
| Life Fiber Detect | ✓ Perform LFD |
| IOR | 1.468200 |
| BSC | -83.000000 |

*Figure 6-5: Acq Settings dialog*

5. Select **Start** under Settings menu (or press F7) to start data collect.

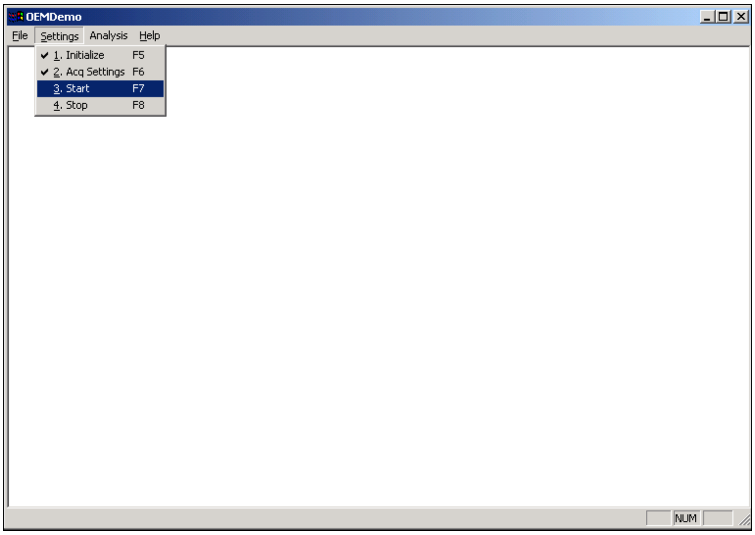


*Figure 6-6: Select "Start"*

6. Select **Stop** under Settings menu (or press F8) to stop data collect.

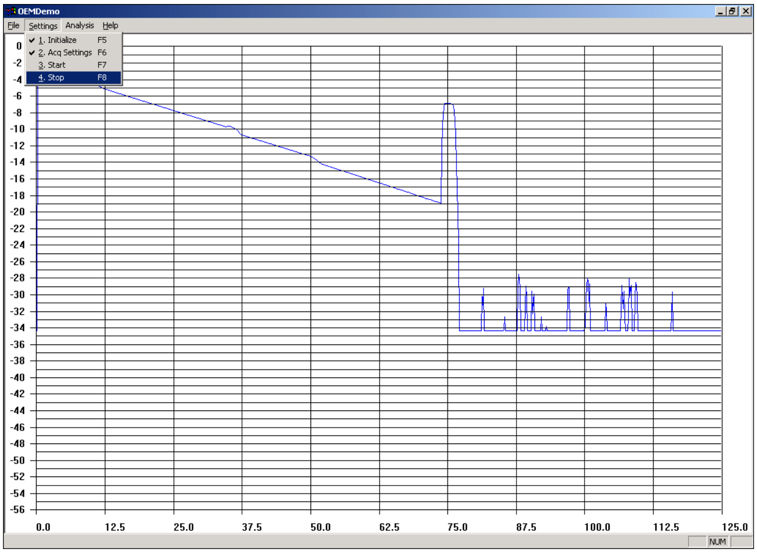


*Figure 6-7: Select "Stop"*

7. Select **Analysis > Display Analysis** to display FAS dialog box.

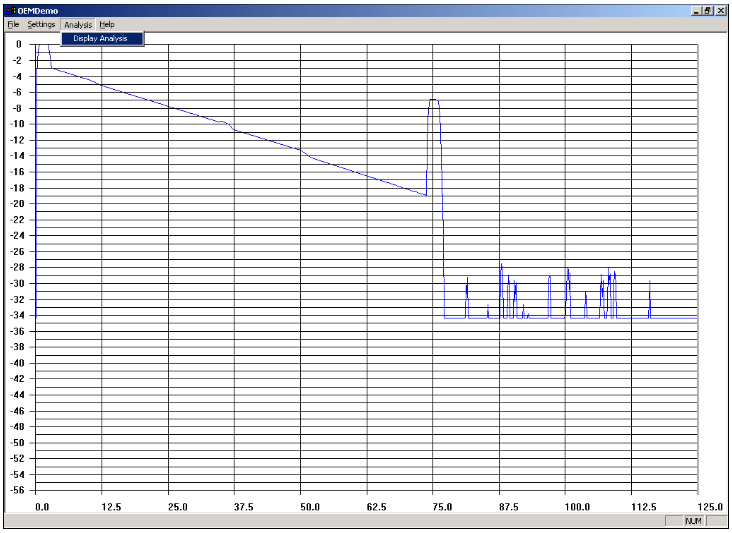
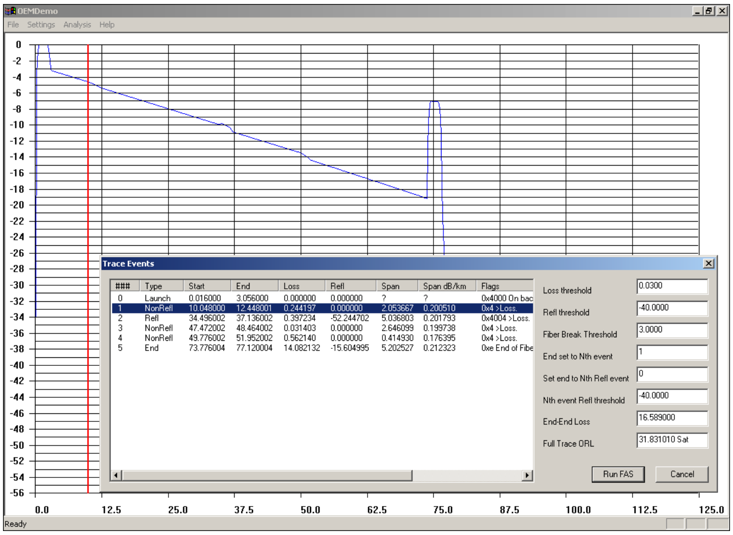


*Figure 6-8: Select "Display Analysis"*

8. Click **Run FAS** and then double click on an event in the list. The selected event is then highlight on the trace grid with the red cursor.



| ### | Type | Start | End | Loss | Refl | Span | Span dB/km | Flags |
|---|---|---|---|---|---|---|---|---|
| 0 | Launch | 0.016000 | 3.056000 | 0.000000 | 0.000000 | ? | ? | 0x4000 On bac |
| 1 | NonRefl | 10.048000 | 12.448001 | 0.244197 | 0.000000 | 2.053667 | 0.200510 | 0x4 >Loss. |
| 2 | Refl | 34.496002 | 37.136002 | 0.397234 | -52.244702 | 5.036803 | 0.201793 | 0x4004 >Loss. |
| 3 | NonRefl | 47.472002 | 48.464002 | 0.031403 | 0.000000 | 2.646099 | 0.199738 | 0x4 >Loss. |
| 4 | NonRefl | 49.776002 | 51.952002 | 0.562140 | 0.000000 | 0.414930 | 0.176395 | 0x4 >Loss. |
| 5 | End | 73.776004 | 77.120004 | 14.082132 | -15.604995 | 5.202527 | 0.212323 | 0xe End of Fibe |

| | |
|---|---|
| Loss threshold | 0.0300 |
| Refl threshold | -40.0000 |
| Fiber Break Threshold | 3.0000 |
| End set to Nth event | 1 |
| Set end to Nth Refl event | 0 |
| Nth event Refl threshold | -40.0000 |
| End-End Loss | 16.589000 |
| Full Trace ORL | 31.831010 Sat |

*Figure 6-9: Trace Events with 1st Event Selected*

# Chapter 7: Transferring Trace and Header Data to Your Application

The Data structure "OTDRWaveformHeader" contains the waveform acquisition parameters. The Data array "OTDRWaveformData" contains the Waveform data, expressed in dB * 1000. For example, a value of 31465 would represent a dB value of 31.465 dB.

```
typedef unsigned short  OTDRWaveformData;
typedef unsigned long   OTDRWaveformSignFlags;

typedef struct
{
    unsigned short     updateData;         // version of PC card
    unsigned short     DAUVer;             // version of the DAU file
    char               PIModel[32];        // kind of plugin
    BYTE               LsrCool;            // Value of Laser Cool register
    char               PISerial[32];       // serial number of plugin
    char               SWDriverVer[16];    // version of sw driver
    double             Wavelength;             // nanometers
    double             Range;                  // kilommeters
    double             PulseWidth;             // meters
    double             PointSpacing;           // meters
    DWORD              FPOffset;               // Front panel distance(m)
    from first data point.
    double             BaseLine;
    double             Reference;
    double             DynRange;           // dynamic range in dB * 1000
    unsigned long     Averages;            // # of acquisitions
    unsigned short    PreAverages;         // 255
    double             Noise;              // dB Rms noise * 1000
    int                NumPts;             // # of data points
    long               LinOffset;          // linear offset
    long               LinOffsetAddr;      // address of offset
    unsigned short     Filter;             // width of filter in points
    double             rBS;                // dBns * 1000, at current
    pulsewidth
    void               *LTSigParms;        // proprietary data
    size_t             LTParmsSize;        // size of proprietary data
    int                LTParmsVersion;     // revision of proprietary data
    OTDRWaveformSignFlags   *SignFlags;        // array of sign flags
    size_t             SignFlagSize;       // size of sign flag array
    double             LinZeroRef;         // ADC reference level.
    void               *D;                 // proprietary data
    BYTE               gainID;             // Current gain id
    BYTE               gainSpliceFailed;   // Gain splice failed flag
    BYTE               ginMaxID;           // Max gain id
} OTDRWaveformHeader;


#define MAX_CARDS           8
#define MAX_WAVES           4
#define MAX_RANGES          16
#define MAX_PULSES          16
#define MAX_AVERAGES        3
#define MAX_SPACINGS        13
#define MAX_DIST_RANGES     10
#define MAX_WFM_POINTS      0x40000
#define REAL_TIME_SCAN      1
#define LIVE_FIBER_DETECT   0x00010000
#define GAIN_SPLICE_ON      2
```