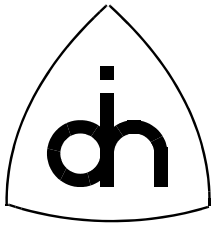
Odin TeleSystems Inc.



Programmer's Guide for OTX Hardware API

Doc. No. 1412-1-SAA-1006-1

Rev. 1.1

October 31, 2006

Changes

The material in this document is for information only and is subject to change without notice. While reasonable efforts have been made in the preparation of this document to assure its accuracy, Odin TeleSystems Inc., assumes no liability resulting from errors or omissions in this document, or from the use of the information contained herein.

Odin TeleSystems Inc. reserves the right to make changes in the product design without reservation and notification to its users.

Warranties

THE SOFTWARE AND ITS DOCUMENTATION ARE PROVIDED “AS IS” AND WITHOUT WARRANTY OF ANY KIND. ODIN TELESYSTEMS EXPRESSLY DISCLAIMS ALL THE WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE. ODIN TELESYSTEMS DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET ANY REQUIREMENTS, OR THAT THE OPERATIONS OF THE SOFTWARE WILL BE UNINTERRUPTED OR ERROR-FREE, OR THAT DEFECTS WILL BE CORRECTED. FURTHERMORE, ODIN TELESYSTEMS DOES NOT WARRANT OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE SOFTWARE OR ITS DOCUMENTATION IN TERMS OF THEIR CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE. NO ORAL OR WRITTEN INFORMATION OR ADVISE GIVEN BY ODIN TELESYSTEMS OR ODIN TELESYSTEMS’ AUTHORIZED REPRESENTATIVE SHALL CREATE A WARRANTY. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY.

UNDER NO CIRCUMSTANCE SHALL ODIN TELESYSTEMS INC., ITS OFFICERS, EMPLOYEES, OR AGENTS BE LIABLE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES (INCLUDING DAMAGES FOR LOSS OF BUSINESS, PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION) ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE AND ITS DOCUMENTATION, EVEN IF ODIN TELESYSTEMS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT WILL ODIN TELESYSTEMS’ LIABILITY FOR ANY REASON EXCEED THE ACTUAL PRICE PAID FOR THE SOFTWARE AND ITS DOCUMENTATION. SOME JURISDICTIONS DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL AND CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY.

Odin TeleSystems Inc.
http://www.OdinTS.com

This document is published by:
Odin TeleSystems Inc.
800 East Campbell Road, Suite 334
Richardson, Texas 75081-1873
U. S. A.

Printed in U. S. A.

1. Table of Content

2. Introduction

The OTX (Odin Telecom FrameworX) Hardware Driver provides the software needed by the PC Host processor to communicate with the OTX Telecom Adapters. The OTX Driver Software implements the hardware dependent access scheme to the resources on the various OTX Adapters. The same hardware driver supports all of the OTX PCI, CompactPCI, PCMCIA and CardBus Adapters.

More specifically, the OTX Hardware Driver performs the following functions:

- Abstracts the Hardware Interface into a higher level Application Programming Interface (API).
- Passes commands and data between software applications and OTX hardware.
- Handles interrupts generated by the adapters and dispatches the notifications to the appropriate software modules and applications.
- Manages the hardware resources allowing multiple applications and/or multiple processes and threads to use multiple adapter boards with minimum blocking.

The OTX hardware Application Programming Interface (API) is a C-language function call interface. The API allows software applications to be written without intimate knowledge of the hardware operation and of the low-level hardware interface. The different interfaces of the OTX hardware driver are illustrated in Figure 1.

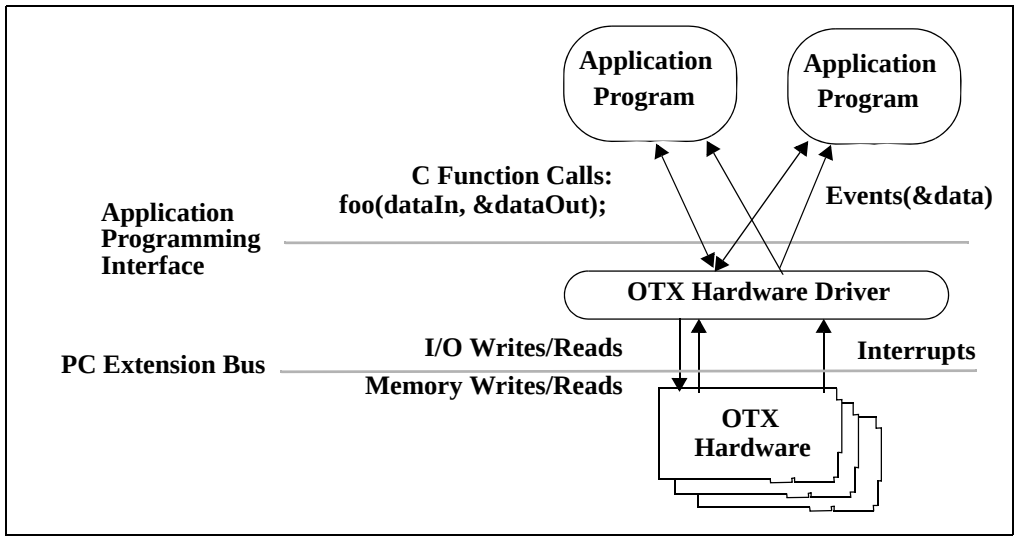
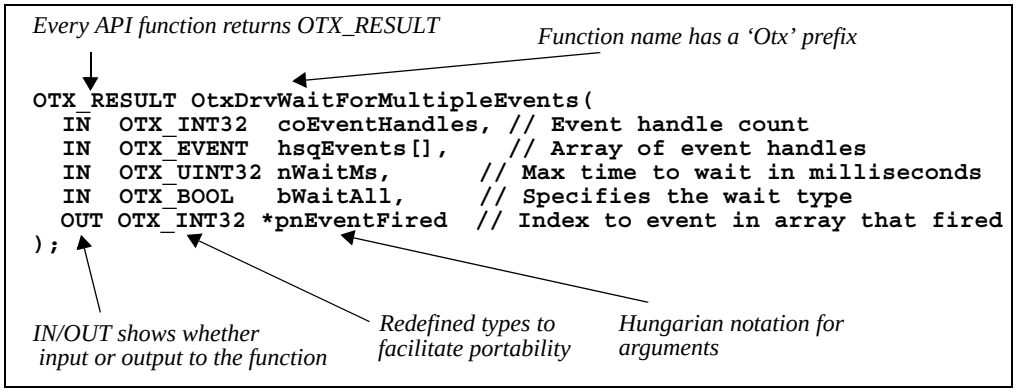


Figure 1. OTX Hardware Driver Interfaces.

3. API Coding Convention

All functions, data types, and macros within the OTX API follow a naming convention. All names have a *prefix* 'OTX' (for Odin Telecom frameworX). Example 1 illustrates a typical function definition in the OTX API and highlights the use naming conventions.



Example 1. OTX API Coding Conventions.

As shown in Example 1, all types have been redefined to facilitate portability. For example, a 32-bit integer type is called *OTX_INT32* in the API. The built in types used in the driver are listed in Example 2.

```
// OTX API Basic Data Types
OTX_INT8   : Signed integer, 8 bits
OTX_INT16  : Signed integer, 16 bits
OTX_INT32  : Signed integer, 32 bits
OTX_INT64  : Signed integer, 64 bits
OTX_UINT8  : Unsigned integer, 8 bits
OTX_UINT16 : Unsigned integer, 16 bits
OTX_UINT32 : Unsigned integer, 32 bits
OTX_UINT64 : Unsigned integer, 64 bits
OTX_CHAR   : Character type (8 bits)
OTX_BOOL   : Boolean
```

Example 2. OTX API Basic Data types.

The function arguments in the OTX API and variables in the driver demo programs are named using the hungarian notation. The used notation is explained in detail in Table 1,

TABLE 1. Hungarian Notation used in OTX API and Demo Programs.

prefix	Meaning
b	Boolean.
c	Char

TABLE 1. Hungarian Notation used in OTX API and Demo Programs.

prefix	Meaning
cb	Same as ‘n’, but specifically used as a count of octets/bytes (e.g. size of array).
co	Same as ‘n’, but used as a count of arbitrarily sized objects.
e	Enumerated type, only certain integer values allowed.
g_	global variable
gs_	static global variable
h	Handle to a driver object
i	Same as ‘n’ but specifically used as index.
n	Number, any type of integer number, size and sign not significant.
p	Pointer ‘*’
sq	Sequence of objects, a generic collection; e.g. arrays.
sz	‘\0’ terminated c-type string (char *)

4. Driver Objects and Handles

Now when we have familiarized ourselves with the used coding conventions, it is time to explore some of the fundamentals in the OTX driver and the OTX hardware API. The concepts essential for understanding the OTX Hardware API are the notions of “Driver Objects” and “Handles.” All the hardware and driver resources are represented to the user (user meaning the application program) as various objects. All operations are always targeted to a specific driver object and all the events are initiated from an object. The driver objects are identified and accessed using handles of type *OTX_HANDLE*.

The OTX Hardware API supports the following types of driver objects:

- **Physical Devices**
- **Logical Devices**
- **Event Queues**
- **Pipes**

4.1 Handles

All the objects are addressed and accessed from an application using handles. The data type of a object handle is *OTX_HANDLE*, which is a 32-bit unsigned integer (independent of the platform). The *OTX_HANDLE* data type and other globally used types are defined in the “*OtxType.h”* header file. Example 3 illustrates how to declare handles with an application program code.

```
#include "OtxType.h"

OTX_HANDLE hMyHandle;        // Declaration of a single handle
OTX_HANDLE hMyHandleArr[4];  // Declaration of a array of handles
```

Example 3. Declaration of Handles.

Handles to driver objects are supplied by the OTX driver. A handle of type OTX_HANDLE can be used to identify any type of driver object: A physical device, a logical device, a pipe, or an event queue. A valid handle is a positive 32-bit value. Negative values are not valid, especially -1 denotes an invalid handle (*OTX_INVALID_HANDLE_VALUE*). The value of a handle also indicates the type of devices it points to, as shown in Table 2.

TABLE 2. OTX_HANDLE bit fields explained.

Bit #	Value	Meaning
31	0	Must be 0 for a valid handle
30-28	001	Handle to a Physical Device
	010	Handle to a Logical Device
	011	Handle to an Event Queue
	100	Handle to a Pipe
27-24	0/1	Reserved for Future use
23-0	any	Sequence number of the handle of its type

4.2 Events and Event Queues

Events are a method for the driver to provide notifications to an application. Events can occur spontaneously and asynchronously (e.g. link down) or as a result of a user incurred action. The purpose of events is to provide a means to the driver to inform the application of a driver status change, of a completed driver operation, or to request an action from the application. The OTX API supports two types of events:

- **Driver Events**
- **Notification Events**

4.2.1 Driver Events

Events initiated by driver objects are called driver events. Every driver event has a data structure of type *OtxEventDataS* associated with it. The *OtxEventDataS* data structure provides information on when the event occurred, which driver object initiated it, and what was the reason for the event. The declaration of *OtxEventDataS* is shown in Example 4.

```
struct OtxEventDataS {
  OTX_DATETIME m_nTimeStamp;  // Absolute time event was recorded
  OTX_UINT32   m_eDeviceType; // Type of device generating the event
  OTX_UINT32   m_nSourceId; // ID of the device generating event
  OTX_HANDLE   m_hDevice;   // Handle to the Device generating event
  OTX_NOTIFY_CAUSE m_nCode; // Device specific cause code indicating
                            // ..what happened.
  OTX_TASK_REF  m_nRequestId; // If the message is for a previous
                              // .. user request. User assigned ID
  OTX_UINT32    m_nParam;   // Optional parameter
};
```

Example 4. Declaration of event structure for driver events.

The driver events are generated internally in the driver and can be retrieved by the application by calling the *OtxDrvGetEventData()* function.

4.2.2 Notification Events

The notification events can be used by an application to wait efficiently for driver events. Notification events behave like WIN32 Events, as a matter of fact on WIN32 platforms the OTX notification events are WIN32 events. On other platforms the notification events are implemented by the driver to simulate the WIN32 Event behavior.

A notification event can be created with the *OtxDrvCreateEvent( )* function as demonstrated in Example 5.

```
  OTX_EVENT  hMyNotificationEvent;
  OTX_RESULT nRresult

  // Creation of a notification event
  nResult = OtxDrvCreateEvent(&hMyNotificationEvent);
```

Example 5. Creation of a notification event.

The use of notification events is covered in more detail in Chapter 7: "Event Driven vs. Polling Operation".

4.2.3 Event Queues

An Event Queue a driver object that is used to serialize the asynchronous driver events. The OTX driver supports multiple events queues: A separate event queue can be associated with every driver device, or multiple driver devices can share an event queue. An event queue can be created with the *OtxCreateEventQueue()* function:

```
OTX_RESULT nResult;
OTX_EVENT  hMyNotificationEvent;
OTX_HANDLE hMyEventQueue;

// Create Event Queue (max size 128 Events). Register event
// hMyNotificationEvent that will be set when driver events
// available.
// Handle to the new event queue returned in hMyEventQueue
nResult = OtxDrvCreateEventQueue(128, hMyNotificationEvent, &hMy-
EventQueue);
```

Example 6. Creation of an event queue.

When an event queue is created, the user can specify the maximum size of the queue in number of events. The user can also register a notification event that will be signalled by the driver every time driver events are inserted to the event queue.

The driver events can be retrieved from the event queue with function *OtxDrvGetEventData()*, as demonstrated in Example 7.

```
OtxEventDataS eventData;
OTX_HANDLE hMyEventQueue;
OTX_RESULT nResult;

// Check for driver events from previously created event queue
nResult = OtxDrvGetEventData(hMyEventQueue, &eventData);

if (nResult == OTX_S_OK) {
  // We have an event
}
```

Example 7. Retrieval of driver events from an event queue.

As was shown in Example 4, the event data contains a driver object specific cause code for the driver event. The cause code can be translated into a string describing the cause with the *OxtDrvEventCode2String( )* function.

4.3 Physical Devices

OTX API Physical Devices represent real (physical) devices on the OTX Adapters. The number of physical devices is set and cannot change dynamically. This is the main difference between physical devices and logical devices: While every OTX adapter has a predetermined number of physical devices available for use, new logical devices can be requested and created by the user when needed.

A physical device can be an entire adapter board (**a Board Device**) or an integrated circuit (**a Chip Device**) populated on a board. A board device corresponds to one OTX Network Interface Card (NIC) or to an OTX ASM Daughter Board Module. The

board device is the place-holder and controller of other physical devices on the board. A chip device represents a physical integrated circuit on the board performing one or several functions. Typical chip devices include:

- **Line Interface (LI) Devices**: A hardware device that interfaces with the external network and converts/maps the external data interface to the internal serial data highways. A LI Device typically includes a transceiver chip and analog front end circuitry. Examples of LI devices include:
  - *OTX_DEVICE_LI_POTS*: Analog Phone Line Interface Device.
  - *OTX_DEVICE_LI_T1E1*: T1/E1 Line Interface Device.
- **Processor Devices**: A processor device is general purpose micro processor or Digital Signal Processor (DSP) implementing certain functionality (logical devices) under software control:
  - *OTX_DEVICE_DSP*: Digital Signal Process Device.
  - *OTX_DEVICE_PROCESSOR*: General Purpose Micro Processor Device.
- **Data Transfer Device:** A device that transfers data from a serial format (PCM highways to a parallel format (e.g. the 32-bit PCI bus)
  - *OTX_DEVICE_BURST:* Tranfers data to and from a PCM highway to the user mode in the host PC via 32-bit DMA burst transfers.
- **Switch Device**: A device connecting and disconnecting data-paths on the board:
  - *OTX_DEVICE_TSS:* Time space switch cross-connecting PCM highways.

Figure 2 illustrates the physical devices available on a Thor-2-PCI-Plus Adapter equipped with a Vidar-55x4-ASM daughter board.

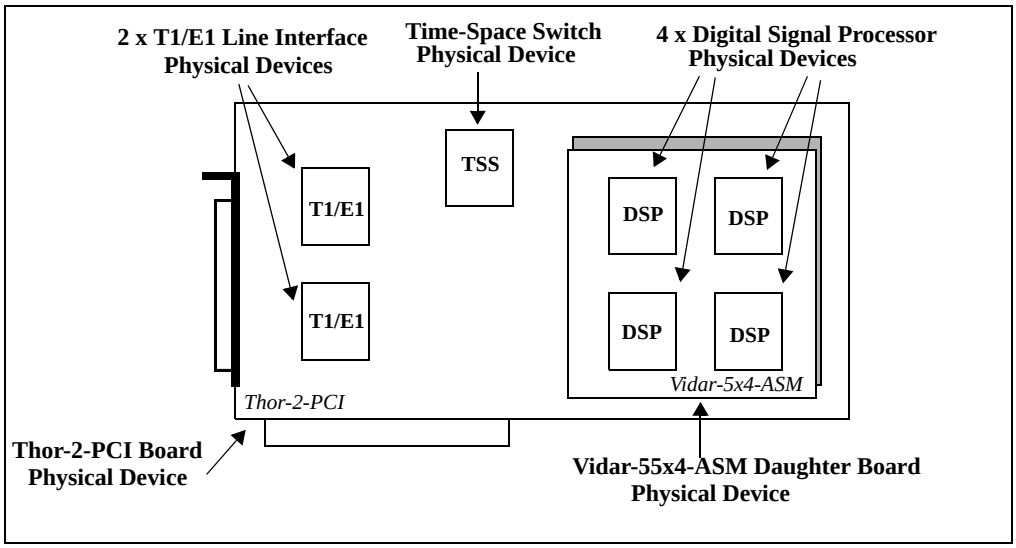


Figure 2. Physical Devices Available on a Thor-2-PCI-Plus & Vidar-55x4-ASM combination.

Each physical devices can be identified with 3 parameters:

- **Device Type**: Each Physical Device is of a certain device type. The device types supported by the driver is enumerated in *OtxPhysicalDeviceTypeE*, which is defined in the *OtxDev.h* header file. For example, Thor-2-PCI-Plus contains two Line Interface physical devices of type *OTX_DEVICE_LI_T1E1*.
- **Parent Device**: The parent device represents a containment relationship. For example, all chip devices always have a board device as a parent. In the case of Thor-2-PCI-Plus, the *OTX_DEVICE_LI_T1E1* have a parent device of *OTX_DEVICE_THOR_2_PCI_PLUS*. The board devices representing the network interface cards are the only devices that do not have parent physical devices.
- **Sequence Number**: Each physical device has a sequence number. The sequence number is a zero-counted number of devices of the same type within one parent device. For example, the Thor-2-PCI-Plus contains two *OTX_DEVICE_LI_T1E1* physical devices with sequence numbers 0 and 1. If we had another Thor-2-PCI-Plus board installed, the LIs on that board would have the same sequence numbers. However, the LI's could be differentiated by the fact that they have different parent devices.

An application can open a physical device for use with the *OtxDrvOpenPhysicalDevice()* function call, as illustrated in Example 8.

```
#define THOR_SOURCE_ID 5
#define LI_SOURCE_ID 6
OTX_RESULT nResult;
OTX_HANDLE hMyThor;
OTX_HANDLE hMyLi;
OTX_HANDLE hMyEventQueue;

// Open a Thor-2-PCI-Plus Board device #0
// Since this is a board device it has no parent (0).
// Returns handle to the board 'hMyThor'
nResult = OtxDrvOpenPhysicalDevice(0, OTX_DEVICE_THOR_2_PCI_PLUS,
0, hMyEventQueue, THOR_SOURCE_ID, &hMyThor);

// Open a T1/E1 Line Interface Device #1 within the
// Thor-2-PCI-Plus board. Use the same event queue.
// Returns handle to the Line Interface device.
nResult  =  OtxDrvOpenPhysicalDevice(hMyThor,  OTX_DEVICE_LI_T1E1,
1, hMyEventQueue, LI_SOURCE_ID, &hMyLi);
```

Example 8. Opening of physical devices.

The first three arguments to the *OtxDrvOpenPhysicalDevice()* function (the parent device, the physical device type, and the sequence number) identify the device to be opened. The next parameter allows the user to provide a handle to an event queue that will be used for driver events generated by this device. The source Id parameter allows the user to provide an identification number that will be assigned to each event generated by this device. The last parameter to the function is an OUT parameter and returns a handle to the opened physical device.

4.4 Logical Devices

A logical device is a driver object representing a logical entity performing certain well defined functions. An OTX adapter can implement logical devices with hardware, firmware, or software. A logical device is always implemented by a physical device.

The logical devices are differentiated from the physical devices by the fact that the number of logical devices is not pre-determined. More logical devices can be requested by the user and allocated by the system when needed. Examples of logical devices include HDLC Receiver and Sender, Tone Detector, Tone Generator, etc.

The following logical devices classes are supported by the OTX Driver:

- **Logical Tone Devices**: Logical devices for tone generation and reception (Defined in *"OtxTone.h"*).
- **Logical Data Devices**: Devices for sending, receiving, and manipulation of raw serial data (Defined in *"OtxData.h"*).
- **Logical HDLC Devices**: Logical devices implementing HDLC Sender and Receiver Devices (Defined in *"OtxHdlc.h"*).

- **Logical CAS Devices**: Devices implementing Channel Associated Signalling for (CAS) functions (Defined in *"OtxCas1.h"*).
- **Logical Modem Devices**: Devices implementing Modem emulation.
- **Logical Fax Devices**: Devices implementing Fax emulation.
- **Logical Voice Devices**: Devices implementing Voice Codecs, echo cancellation, silence suppression, etc.

New logical devices can be instantiated with the *OtxDrvCreateLogicalDevice()* function. Example 9 illustrates a creation of a logical DTMF detector device. The logical device in question is implemented with DSP software, thus the host device for the logical device is a physical DSP chip device. As with physical devices, the logical devices can be associated with event queues and provided with source identifiers to be used with generated events.

```
#define DTMF_SOURCE_ID 10
OTX_HANDLE hMyDsp;
OTX_HANDLE hMyDtmfDetector
OTX_HANDLE hMyEventQueue;

// Instantiate a new DTMF detector logical device. The device is
// implemented by software in a DSP physical device. Returns a
// handle 'hMyDtmfDetector' to the newly created logical device
OtxDrvCreateLogicalDevice(hMyDsp, OTX_LDEVICE_TONE_DTMF_DETECTOR,
hMyEventQueue, DTMF_SOURCE_ID, &hMyDtmfDetector);
```

Example 9. Creation of a logical device.

As was mentioned above, logical devices are dynamic. For example, the same adapter board can be made to support a completely new set of logical devices by simply loading new programs into the on-board Digital Signal Processors (DSPs).

4.5 Pipes

A Pipe device is a logical entity representing a connection between two physical devices. A pipe is used to represent serial Time-Division Multiplexed (TDM) data transfer between two connected devices. The connected devices can reside on a single board or on separate boards. This location of the devices and the needed cross-connects are all transparent to the user. Once the user has connected any two physical devices with a pipe, the physical cross-connects (locally on one board, or through the H.100/H.110 bus) are created automatically.

A pipe can be created for a any data rate between 8 kbit/s and 8.192Mbit/s with 8 kbit/s increments. A pipe can correspond to a single time-slot on a physical highway or it can comprise of a superchannel of multiple time slots. The requested capacity is specified when the pipe is created. Pipes can be created with the driver function

OtxDrvCreatePipe( ), which takes the requested capacity as a parameter and returns a handle to the newly created pipe. The creation of a pipe is illustrated in Example 10.

```
OXT_RESULT nResult;
OTX_HANDLE hMyPipe;

// Creation of a 64 kbits/s Pipe. Returns a Handle to the Pipe
nResult = OtxDrvCreatePipe(64, &hMyPipe);
```

Example 10. Creation of a Pipe.

After a pipe has been created, it exists as a purely logical entity with a specified capacity. The next task of an application is to connect both ends of the pipe to two physical devices. The connection is performed with the *OtxDrvConnectPipe()* function. At this time the pipe can be mapped to a physical highway and on to specific time-slots. The mapping can be done by providing a channel mask as a parameter to the *OtxDrvConnectPipe()* function. A channel mask is a array of 32 Bytes, where each byte represents one of the 32 time-slots in a 2.048 Mbit/s highway. Within each byte, one of the eight bits represent a bit in a 64kbit/s time-slot. Thus, each bit represents a 8 kbit/s sub-channel. Writing ‘1’ to any bit position within the channel mask will cause the pipe to be mapped to use that physical channel. Connecting a pipe to a physical device and mapping a pipe to specific time-slots is illustrated in Example 11.

```
// Channel mask for enabling time-slot 12
OTX_UINT8 sqTimeSlot12Mask[32] =
{ 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
  0xff,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
  0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};

OTX_RESULT nResult;
OTX_INT32  dummy;

// Connect output of a pipe to the T1/E1 Physical Device
// using its Highway #1
// Force connection to use time-slot 12 on that highway
nResult = OtxDrvConnectPipe(hMyPipe, OTX_PIPE_OUTPUT, hLi[0], 1,
sqTimeSlot12Mask, 32);
```

Example 11. Connecting a Pipe to a Physical Device and mapping of the pipe to a specific time-slot.

If the user does not care which physical time-slots are used, the user can specify “don’t care” to the mapping parameters, and the driver will automatically use the most suitable physical time-slots.

A pipe has a direction; i.e., it has an input and an output. Thus, two Pipes and four calls to the *OtxDrvConnectPipe()* function are needed for a full duplex connection between two devices.

5. OTX Driver Model

Now when we have a solid understanding of the various driver objects, it is time to put all this information together to form a Driver Model for the OTX Hardware driver and the API. The Physical and Logical Device driver objects contain four interfaces: Serial Data Stream In, Serial Data Stream Out, Control In, and Events Out. The OTX Driver device model is shown in Figure 3.

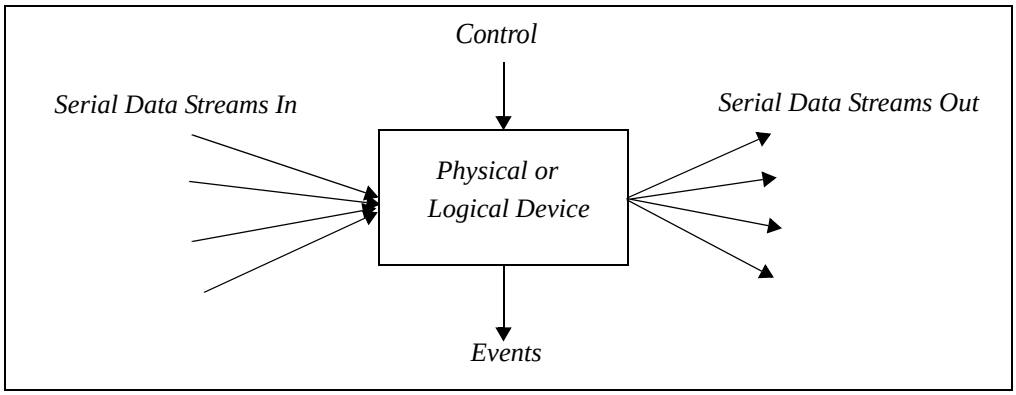


Figure 3. OTX Driver device interface model.

The device control interface is used by the application to manage the device and to access user data. The control interface is accessed using the handle to the device. The device uses the event interface to inform the application on driver events. The serial data interfaces transfer serial data stream in and out from the device. Each device can have multiple data stream coming and leaving the device.

A pipe device has 3 interfaces: Serial Data In, Serial Data Out, and Control In. A model for a pipe object is shown in Figure 4.

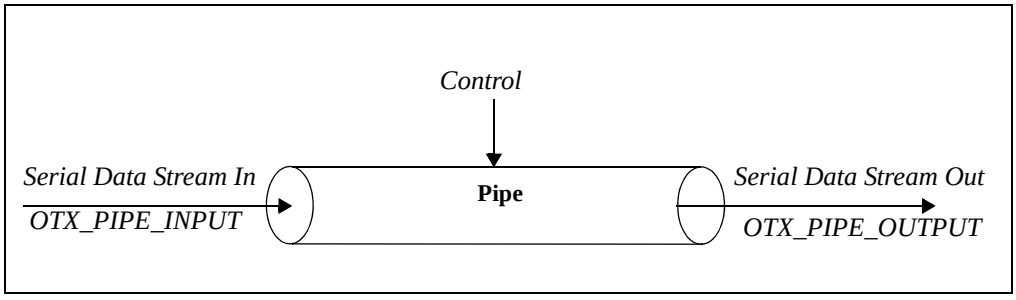


Figure 4. Pipe Object Interface Model.

A pipe transmits serial data stream from one physical device to another. A pipe is one directional; i.e. it has an input and an output.

An event queue can be modeled as shown in Figure 5.

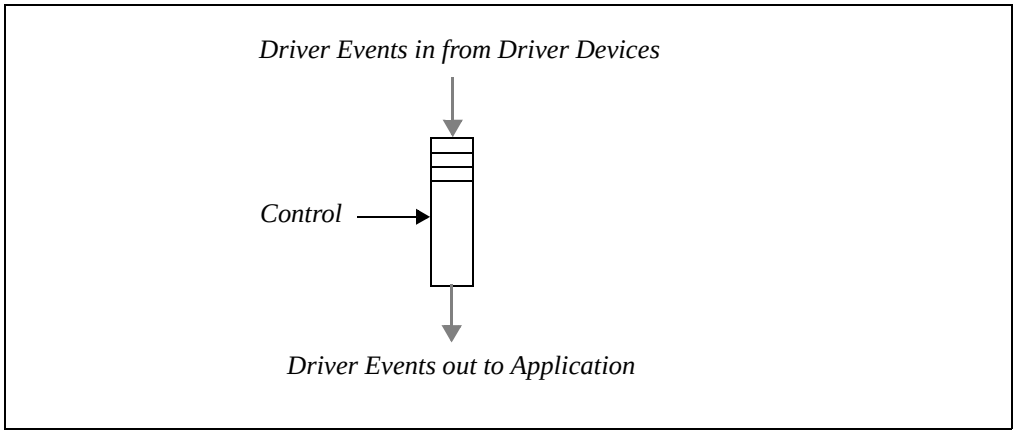


Figure 5. Event Queue interface model.

An event queue takes in driver events from driver devices and queues them for retrieval by an application.

By putting the above models together, we can now develop an overall model for the OTX driver. Figure 6 shows one example of how the OTX driver could be configured.

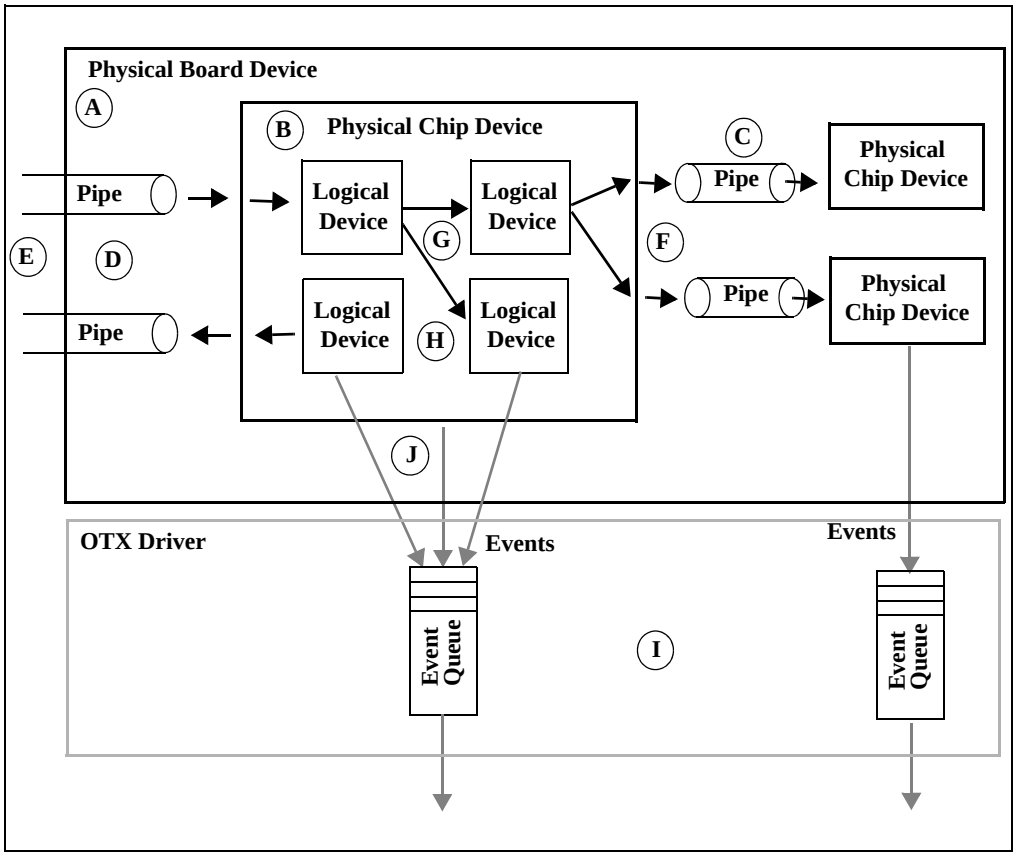


Figure 6. Example of a OTX driver configuration.

From Figure 6 we can deduce several important points to remember:

A. A physical board device representing a Network Interface Card (NIC) is always the root device that contains physical chip devices and possibly other board devices (representing daughter board modules). The number of physical devices available is set and cannot be changed. (Please refer to *OtxDrvOpenPhysicalDevice( ))*

B. An arbitrary number of Logical Devices implementing certain functions can be instantiated within physical devices. (Please refer to *OtxDrvCreateLogicalDevice( )*)

C. Pipes are used to connect physical chip devices. (Please refer to *OtxDrvConnectPipe()*)

D. Pipes are one-directional. Two pipes are needed for a full duplex connection. (Please refer to *OtxDrvCreatePipe( )*)

E. External interfaces can be described with pipes that has only one end connected to chip device and one end is left unconnected representing the fact that the other end of the link is beyond the scope of the driver. For example, on the Thor-2-PCI adapter the external T1/E1 links can be represented with pipes which have one end connected to the *OTX_DEVICE_LI_T1E1* devices and one end has been left unconnected.

F. Physical devices can be connected to multiple pipes.

G. Logical devices can be connected in series and in parallel to form advanced configurations. Only logical devices within one physical device can be connected together. (Please refer to *OtxDrvConnectLogicalDevice( )*)

H. Not all the devices have both input and output for serial data streams. Certain devices have only input. These devices are called data sink. An example of a data sink is a *OTX_LDEVICE_TONE_DTMF_DETECTOR* logical device. Certain devices have an data output. These devices are called data sources. An example of an data source is a *OTX_LDEVICE_HDLC_SENDER* device.

I. The OTX Driver can support multiple event queues (Please refer to *OtxDrvCreateEventQueue( )*)

J. Multiple driver devices (both physical and logical) can be associated with the same event queue.

6. Blocking vs. Non-Blocking Function Calls

Certain OTX API function calls cannot always be completed without waiting for device operations to complete. The time to wait can vary depending on external factors, such as a line interface events or timers. These types of functions pose a problem to the application writer as they can take a significant time to complete. It is often not acceptable for the application to stop and wait for a function call to the driver to complete. To solve this problem, the OTX driver provides two calling methods for these types of functions:

- **Blocking**: The driver function attempts to complete the whole operation before returning. The calling program will wait for the function return. A time-out value can be provided to the function to force a return and to prohibit a potential deadlock situation.
- **Non-Blocking:** The driver function returns immediately. If the function was able to complete operation without waiting, it will return the result of the operation. Otherwise the function will return *OTX_S_PENDING* and schedule a task to be completed when the needed data becomes available. The application can now continue its operations while the requested task will be completed in the background. Once the task has been completed, a driver event will be provided to the application through the appropriate event queue. The event will contain a reference value that was provided in the non-blocking function call.

By default, the OTX API function calls are blocking. Functions that provide the non-blocking calling option contain arguments for task reference and timeout as shown in Example 12.

```
// Prototype of a driver function with non-blocking calling option
OTX_RESULT OtxXXXFoo(
  IN OTX_HANDLE hObject,      // Handle to target driver object
    :                         // Other parameters
  IN OTX_TASK_REF nTaskRef,   // Task Reference
  IN OTX_TIME nWaitMaxMs      // Timeout
);
```

Example 12. Prototype of a non-blocking function call.

A blocking function call can be made blocking by providing *OTX_TASK_SYNC* as the task reference. An example of a blocking function call with a 5 second time-out and without a time-out are shown in Example 13.

```
OTX_RESULT nResult;
OTX_HANDLE hMyDev;

nResult = OtxDrvOpenPhysicalDevice(...., &hMyDev);

// Blocking call with 5 second time-out
nResult = OtxDrvEnable(hMyDev, OTX_TASK_SYNC, 5000);

// Blocking call without time-out
nResult= OtxDrvDisable(hMyDev, OTX_TASK_SYNC, OTX_TIME_INFINITY);
```

Example 13. Blocking call examples using a non-blocking function.

A positive (>= 1) task reference value will cause a non-blocking function call and a creation of a background task. Once the task has completed or if the time-out has expired, a driver event will be provided to the user. The driver event contains the task reference value as given to the function. This allows the application to match the events to the tasks. An example of a non-blocking function call is shown in Example 14.

```
#define MY_TASK_REF

// Non-Blocking Function call ...
nResult = OtxToneDtmfDial(hMyDev, "19726640100", 11, MY_TASK_REF,
5000);

if (nResult == OTX_S_PENDING) {
   // Task created, execution continues immediately
} else {
   // Function call completed, Success or failure
}
```

Example 14. An example of a non-blocking function call.

If the user application does not care to keep track of what tasks have been completed, the application can use a special *OTX_TASK_ASYNC* task reference, or alternatively the same positive task reference can be used over and over again.

A pending task can be cancelled with the *OtxDrvCancelTask()* function, as shown in Example 15. Also, a task is automatically cancelled at the end of the specified timeout period. If a task is cancelled due to time-out, a driver event is sent to the application.

```
// Cancel the task created in Example 14.

nResult = OtxDrvCancelTask(hMyDev, MY_TASK_REF);
```

Example 15. Cancelling of a task.

7. Event Driven vs. Polling Operation

It is recommended that the application utilizing the OTX driver and OTX API is written in event driven fashion. The OTX API provides notification events and wait functions to write efficient applications for multi-tasking environments. The API functions allow the applications to go to sleep when nothing is happening and wake up when driver reports events or when a user requests action.

As was mentioned earlier, the function *OtxDrvCreateEvent( )* can be used to create a notification event to be used to signal the application that driver events are available. The creation of notification events was demonstrated in Example 5. The created notification can then be passed on to a event queue to be used to signal the application. The registration of the notification event with an event queue was shown in Example 6.

Once the notification event has been registered, the application can call *OtxDrvWaitForSingleEvent( )* function to wait efficiently for a notification event. The use of the *OtxDrvWaitForSingleEvent( )* function is demonstrated in Example 16.

```
OTX_EVENT hMyEvent;
OTX_RESULT nResult;

// Create a new notification event
nResult = OtxDrvCreateEvent(&hMyEvent);

// Go to sleep waiting for the event to be signalled.
// Wait max 5 seconds
nResult = OtxDrvWaitForSingleEvent(hMyEvent, 5000);

if (nResult == OTX_S_SIGNALLED) {
  // Event was signalled ....
} else if (nResult == OTX_E_TIMEOUT) {
  // Time-out, notification event not received
}
```

Example 16. Waiting for a notification event to be signalled.

The function *OtxDrvWaitForMultipleEvents( )* can be used to wait for one of many events to be signalled. The *OtxDrvWaitForSingleEvent( )* and *OtxDrvWaitForMultipleEvents( )* are modeled after similar WIN32 system services. In WIN32 platforms the functions are implemented utilizing the corresponding WIN32 system services. On other platforms the driver implements the same functions simulating the WIN32 functionality.

The OTX API also supports the implementation of applications which are polling. The function *OtxDrvPollEvents( )* can be used to constantly poll for events. The *OtxDrvPollEvents( )* function can also be used to execute the driver Interrupt Service Routine (ISR) in case the there is a reason not to use physical interrupts. **Polling applications are not recommended** except possibly in the DOS environment.

8. Generic Driver Object Functions

The OTX driver API provides certain functions that can always be applied to any driver object to which the application owns an handle.

8.1 Enabling

As the name says, the *OtxDrvEnable( )* function enables a driver object. The function performs the following operations on the different types of drivers objects:

- **Physical or logical devices**: Makes the device to accept I/O requests and allows it start generating driver events, if an event queue has been associated with the device.

- **Pipe**: Through connects the pipe internally from input to output and begins the serial data transmission through the pipe.
- **Event Queue**: Allows an event queue to start accepting driver events and to start generating notification events to the application, if so configured.

8.2 Disabling

The *OtxDrvDisable( )* function disables a driver object:

- **Physical or logical devices**: Makes the device to reject all I/O requests. Only certain control commands can be issued to a disabled object (such as configuration, reset, and enable commands). A disabled object cannot generate events. However, events already in the associated event queue will not removed by this command.
- **Pipe**: Disconnects the pipe internally and terminates the serial data transmission through the pipe.
- **Event Queue**: Stops the event queue from accepting driver events and halts generation of notification events to the application. The events already in the event queue will remain in the queue until the queue is enabled or reset.

8.3 Checking State

The *OtxDrvGetState( )* returns the state of a driver object. Any driver object can be in one of the 3 states:

- *WORKING*
- *MANUALLY BLOCKED*
- *AUTO BLOCKED*

Disabling an object with the *OtxDrvDisable( )* function forces the device to *MANUALLY BLOCKED* state. Enabling of a driver object with function *OtxDrvEnable( )* attempts to set the device into a *WORKING* state. However, if the driver object does not operate correctly, e.g., it generates excessive amounts of driver events, the driver can automatically block the device and force it to state *AUTO BLOCKED*.

8.4 Resetting

The *OtxDrvReset( )* function software or hardware resets a driver object depending on the object. In a normal operation it should not be necessary to reset any driver objects. A reset is normally performed to resolve an error condition. Certain driver objects must be disabled before they can be reset.

- **Physical Devices:** Hardware or software resets the device. Clears all the driver events generated by the device from the associated event queue.
- **Logical Devices:** Software resets the device. Clears all the driver events generated by the device from the associated event queue.

- **Pipe**: No operation.
- **Event Queue**: Empties the event queue from all the events.

8.5 Closing

Closes a driver object and renders the handle invalid. When a parent object is closed, all the child objects are automatically closed as well.

9. Error Handling

As already shown in the previous examples, all OTX API functions return a result code of type *OTX_RESULT*. Two types of result codes exist:

- **Generic Result Codes**: These result codes are generic to the driver. Any driver object can use and return them. The generic result codes are defined in *"OtxErr.h"*.
- **Object Specific Result Codes**: Object specific result codes are only known and used by one driver object type. The Object specific result codes are specified in the appropriate header file for the driver object.

Result codes with positive value indicate success, and codes with negative values indicate an error condition. Some of the generic codes for successful operations are shown in Example 17 and some generic error codes in Example 18, respectively.

```
// Everything ok
#define OTX_S_OK                        (0x00000000L)

// Everything ok, but result is logically false
#define OTX_S_FALSE                     (0x00000001L)

// An event was signalled
#define OTX_S_SIGNALLED                 (0x00000002L)

// An wait operation timed out
#define OTX_S_TIMEOUT                   (0x00000003L)

// The data necessary to complete the operation not available.
#define OTX_S_PENDING                   (0x00000004L)
```

Example 17. Examples of generic result codes for success.

```
// Catastrophic failure
#define OTX_E_UNEXPECTED                (0x8000FFFFL)

// Not implemented
#define OTX_E_NOTIMPL                   (0x80000001L)

// Ran out of memory
#define OTX_E_OUTOFMEMORY               (0x80000002L)

// Ran out of software resources
#define OTX_E_OUT_OF_RESOURCES          (0x80000003L)

// One or more arguments are invalid
#define OTX_E_INVALIDARG                (0x80000004L)
```

Example 18. Examples of generic error codes.

The OTX API provides handy macros for checking whether a operation was a success or a failure. The use of these macros is demonstrated in Example 19.

```
OTX_HANDLE hMyObject;
OTX_RESULT nResult;


nResult = OtxDrvFoo(hMyObject);

if (OTX_SUCCESS(nResult)) {
  // one of success result codes returned
}

nResult = OtxDrvFooBar(hMyObject);

if (OTX_FAILURE(nResult)) {
  // Failure .....
}
```

Example 19. Checking of result codes.

The OTX API also provides a function, *OtxDrvResultCode2String( )*, to translate the result codes into a string containing a textual explanation for the result code. The use of *OtxDrvResultCode2String( )* is demonstrated in Example 20.

```
OTX_HANDLE hMyObject;
OTX_RESULT nResult;
OTX_CHAR   szStrBuf[256];

nResult = OtxDrvFoo(hMyObject);

if (OTX_FAILURE(nResult)) {
  // failed, print an error message
  OtxDrvResultCode2String(hMyObject, nResult, 256, szStrBuf);
  printf("OtxDrvFoo failed (%p) '%s'\n", hMyObject, szStrBuf);
}
```

Example 20. Converting result codes to strings.

10. Initialization and Cleanup

By now we have covered most of the driver level functions in the OTX API. However, there are a few steps that an application always needs to perform. The very first thing an application must do is to connect to the driver. This is performed using the *OtxDrvConnectLib( )* function, as illustrated in Example 21. The *OtxDrvConnectLib( )* function initializes the driver API library and sets up a communications channel to the kernel mode driver.

```
OTX_RESULT nResult;

nResult = OtxDrvConnectLib();

if ( nResult != OTX_S_OK) {
  printf("Unable to connect to the OTX Driver\n");
  exit(1);
}
```

Example 21. Connecting to the OTX Driver.

After the application has connected to the driver, it may want to verify and display to the user the name and revision of the driver in use. This can be accomplished with the *OtxDrvIdent( )* function, as shown in Example 22.

```
#define MY_BUF_LEN 256
OTX_RESULT nResult;
OTX_CHAR   szNameBuf[MY_BUF_LEN];
OTX_CHAR   szRevBuf[MY_BUF_LEN];


nResult = OtxDrvIdent(MY_BUF_LEN, szNameBuf, MY_BUF_LEN, szRevBuf);

if (OTX_SUCCESS(nResult)) {
  printf("Using Driver '%s', Revision '%s'\n", szNameBuf, szRevBuf);
}
```

Example 22. Identifying the driver revision and name.

At this time the application can use the various *OtxDrvEnumXXXX( )* functions to determined what physical and logical devices that are available in the system. The application can also call the *OtxDrvGetBoardData( )* function to get the serial number and revision information on any specific board in the system.

Upon exit, it is very important that the application calls the function *OtxDrvReleaseLib( )*, as this function releases all the logical devices, pipes, and event queues created by the application. The use of *OtxDrvReleaseLib( )* is emphasized in Example 23.

```
// Delete allocated resources. IMPORTANT TO CALL BEFORE EXIT!!
OtxDrvReleaseLib();
```

Example 23. Driver cleanup upon exit.

11. Debugging and Trouble Shooting

After reading this far we are naturally so proficient with the OTX API that there will hardly be any need for debugging or trouble shooting. Our application will simply work. But just in case, let's highlight some of the debugging facilities provided by the OTX API.

The enumerate functions for physical and logical devices introduced in the previous chapter can be a valuable debugging tools as well. In addition, the OTX API provides the following functions:

- *OtxDrvGetDeviceStats( )*
- *OtxDrvGetPhysicalDeviceData( )*
- *OtxDrvGetLogicalDeviceData( )*

- *OtxDrvGetEventQueueData( )*
- *OtxDrvGetPipeData( )*

Once the application has acquired a handle to a driver object, the above listed functions can be used to retrieve the internal data kept for these objects. The debugging data can be used to verify that the objects have been connected properly and that the containment hierarchy has been set-up as it should.

Finally, the *OtxDrvGetState( )* function can be used to verify that the driver objects are in *WORKING* state. Any *AUTO BLOCKED* objects indicate a software or even possibly a hardware problem.

12. Recommended Program Flow

The OTX driver expects that the various driver objects are created, connected, and initialized in a certain order. The flow chart in Figure 7 describes a recommended flow for an application using the OTX driver.

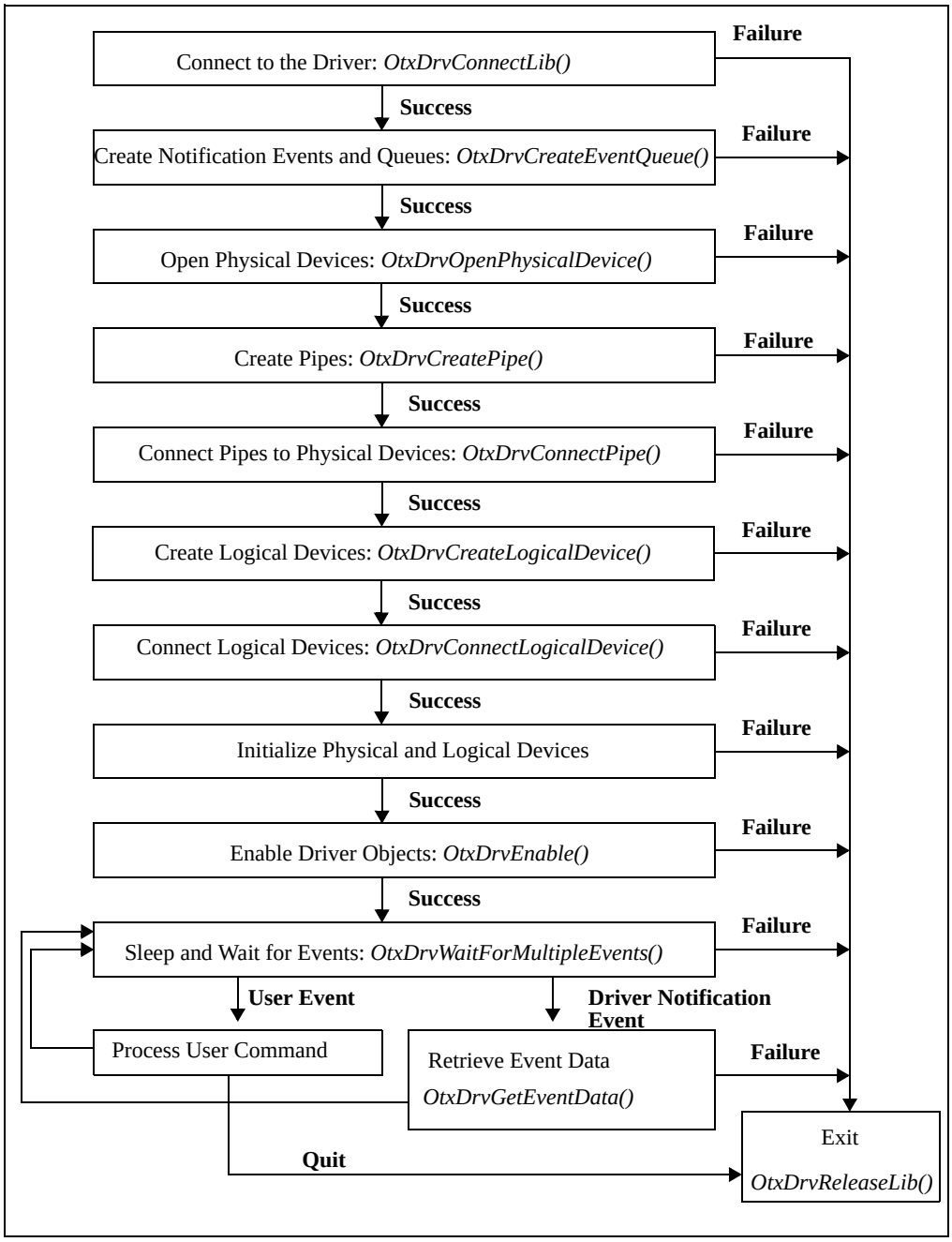


Figure 7. Recommended program flow for OTX Driver Initialization and use.

Doc. No. 1412-1-SAA-1006-1
For more information on this product, please contact:

Odin TeleSystems Inc.
800 East Campbell Road, Suite 334
Richardson, Texas 75081-1873
U. S. A.

Tel: +1-972-664-0100
Fax: +1-972-664-0855
Email: Info@odints.com
URL: http://www.odints.com/