AddiPack Documentation

Version 2026.7.1

Table of Contents

About Addi-Pack

Introduction 5

Getting Started 8

- Installation Guide 8
- API Programming Guide 17
- Driver Concepts Guide 33
- Addi-Pack Samples 42

Technical Documentation 70

- C API 70
- Common API 101
- C++ API 70

Help & Informations 160

- Maintenance Information 160

License & Legal Notice 167

See PDF version

Trademarks

- ADDI-DATA, APCI-1500, MSX-Box and MSX-E are registered trademarks of ADDI-DATA GmbH.
- Turbo Pascal, Delphi, Borland C, Borland C++ are registered trademarks of Borland Software Corporation.
- Microsoft .NET, Microsoft C, Visual C++, MS-DOS, Windows XP, Windows 7, Windows 10, Windows Server 2000, Windows Server 2003, Windows Embedded and Internet Explorer are registered trademarks of Microsoft Corporation.
- Linux is a registered trademark of Linus Torvalds.
- LabVIEW, LabWindows/CVI, DASYLab, DIAdem are registered trademarks of National Instruments Corporation.
- CompactPCI is a registered trademark of PCI Industrial Computer Manufacturers Group.
- VxWorks is a registered trademark of Wind River Systems, Inc.
- RTX is a registered trademark of IntervalZero.

Welcome to Addi-Pack

Addi-Pack brings together all the essential components to develop, run, and interact with ADDI-DATA hardware, from drivers to applications.

It provides a unified framework for:

- **Driver management** (KMDF / DKMS / RTX)
- **Hardware control** (digital I/O, timers, counters, watchdogs)
- **Application development** (C, C++, C#, Python)

Tip

Addi-Pack brings together all ADDI-DATA technologies — from low-level drivers to high-level IT interfaces (OPC UA) — in one consistent Driver.

Overview

Addi-Pack is divided into **three main blocks** , each with a specific purpose:

1. **Public API**

   The entry point for all user applications. Available in **C** , **native C++** , **C#** and **Python** . Highly polymorphic, the public API functions can be used with **any supported devices** .

2. **C++ Backend**

   Handles logic and parameter validation. Translates Public API calls into driver commands.

3. **Device Driver**

   The kernel-level component that interacts directly with the hardware. It performs **low-level read/write operations** , configures the device and exposes a **safe, unified interface** to the upper layer.

   Interrupt handling and other internal mechanisms are described later in the Driver Concepts Guide section.

Supported Platforms

Addi-Pack officially supports the following systems:

Operating System	Version	Supported Device
Windows	7 / 8 / 10 / 11	APCI-1500, APCIe-1500, CPCI-1500 APCI-1564, APCIe-1564 CPCI-1564, CPCIs-1564 APCI-1032, APCIe-1032 APCI-1648, APCI-1696 APCIe-1532, CPCIs-1532 APCI-2200, APCIe-2200
Debian	9 and later	
Ubuntu	18.04 LTS and later	
CentOS	Stream 9 / 10	
RTX	RTX 4.X	

Note

For support on other operating systems, please contact us at info@addi-data.com .

Introduction

What is Addi-Pack?

Addi-Pack is the official software suite for all **ADDI-DATA Hardware devices** . It provides a unified environment for **hardware control** , **data acquisition** , and **application development** on both **Windows** , **Linux** and real-time system such as **RTX** .

Its main goals are to:

- Simplify the installation and configuration of ADDI-DATA devices, allowing users to deploy the system quickly and obtain their first signal within minutes.
- Support the development of multi-modal acquisition and processing applications, covering different physical signals (digital I/O, analog channels, timers, counters, watchdogs, etc.), including real-time operation on Windows, Linux and RTX systems.
- Provide a single API for all supported devices, allowing the same code to run across devices and operating systems, which simplifies development and maintenance.
- Integrate DAQ systems into industrial environments, with support for fieldbus protocols like EtherCAT, OT/IT communication interfaces such as OPC UA, and automation systems including TwinCAT

Core Principles

Addi-Pack is built on three main ideas:

1. **Portability** - Run the same code on any of supported platforms.
2. **Modularity** - Drivers, APIs, and bindings are independent yet fully compatible
3. **Consistency** - Unified function model: identical names, parameters, and error codes across all devices

This ensures a predictable experience no matter which Operating System or programming language you use.

Architecture

Addi-Pack is organised into **four layers** , each with a specific role:

> **Tip**
>
> The same architecture applies to all supported cards, including **APCI-1500** , **APCIe-1032** , and **APCIe-1564** .

Getting Started

The following sections will help you install Addi-Pack, detect your hardware, explore the API, and try out working examples.

A typical Addi-Pack usage scenario involves:

1. **Install** – Run the installer or package (*.exe* , *.deb* , or *.rpm*)

2. **Detect** – Use CLI tools or samples to list and configure your devices

3. **Develop** – Write and build your project (CMake, Visual Studio, or Python)

4. **Deploy** – Integrate your application into your target system or environment

5. **Maintain** – Monitor logs, update drivers, and manage Addi-Pack versions

Installation Guide

Overview

This section explains how to install **Addi-Pack** , set up your environment, and verify that your ADDI-DATA cards are correctly detected.

This section will guide you through:

1. Installing the drivers and API libraries

2. Verifying that your ADDI-DATA card is correctly detected

3. Running your first sample

OS Dependencies

Windows:

- Visual Studio 2022 (or higher)
- Administrator rights during installation

Linux:

- dkms
- glibc-devel
- kernel-devel
- cmake
- make

Note

On **CentOS** , you must enable the EPEL repository before installing Addi-Pack:

```
sudo dnf install epel-release
sudo dnf install dkms glibc-devel kernel-devel cmake gcc-c++
```

The **EPEL** (Extra Packages for Enterprise Linux) repository provides additional development tools not available in default CentOS/RHEL repositories.

Installation

Inserting a PCI, PCIe or CPCI/CPCIs card

Follow these general steps to install your ADDI-DATA PCI or PCIe card safely:

1. Power Down and Unplug the PC
   - Shut down your computer completely.
   - Unplug it from the power source before opening the case.
2. Prepare Your Workspace
   - Work on a clean, non-conductive surface.
   - Use an anti-static wrist strap if available, or regularly touch a grounded metal part of the case to discharge static electricity.
3. Open the PC Case
   - Remove the side panel or cover according to your PC manufacturer's instructions.
4. Locate a Free PCI/PCIe Slot
   - Identify a compatible free slot (PCI, PCIe, CompactPCI(CPCI) or CompactPCI Serial (CPCIs) depending on the card model).
   - Remove the corresponding metal bracket on the case.
5. Insert the card Carefully
   - Hold the card by its edges. Avoid touching the gold connectors or electronic components.
   - Align the card with the slot and press it down **evenly** and **firmly** on both ends until it is fully seated.
   - Do not force it, if it doesn't fit, verify the alignment.
6. Secure the card
   - Use a screw to fasten the mounting bracket to the chassis.
7. Check for Obstructions
   - Make sure no cables or nearby components are pressing against the card.
8. Close the Case and Reconnect Power
   - Replace the case panel and reconnect the power cord.

9. **Power On and Verify Detection**
    - Start your computer.
    - On **Windows** , check the **Device Manager** .
    - On **Linux** , run `lspci` to verify the board is recognized.

Windows

1. **Download the installer**

   Download the latest installer (`.exe`) from the official ADDI-DATA website (https://www.addi-data.com/drivers).

2. **Run the installer**
    - Run `AddiPack-2025.x-Windows-x86_64.exe` with administrative privileges.
    - Follow the on-screen steps.

3. **Verify the environment variable**

   The installer adds an environment variable that points to the directory in program files. It's name is AddiPack_ROOT. You can verify the existence of said variable via the following command.

```
echo $env:AddiPack_ROOT
```

4. **Check device detection**

   From PowerShell:

```
cd "$env:AddiPack_ROOT\samples\bin\x64\generic"
.\get_devices_info.exe
```

   You should see a list of detected devices.

   For a detailed explanation and sample output, see the verify_installation section.

Linux (Debian / Ubuntu)

1. **Download the package**

   Download the latest package (`.deb`) from ADDI-DATA website (https://www.addi-data.com/drivers).

   Example: `AddiPack-2025.09.1-Linux-x86_64.deb`

2. **Install dependencies**

```
sudo apt update
sudo apt install build-essential dkms cmake
```

3. **Install Addi-Pack**

```
sudo dpkg -i AddiPack-2025.09.1-Linux-x86_64.deb
```

4. **Check device detection**

   Compile and run a sample:

```
cd $AddiPack_ROOT/samples/get_devices_info
cmake -B build && cmake --build build
./build/get_devices_info
```

Note

The variable `$AddiPack_ROOT` is automatically set during installation. On Linux, the default path is: `/opt/addi-data/addipack`

Tip

If your card is not detected, check if the kernel module is loaded:

```
lsmod | grep apci
```

Example output if loaded correctly:

```
apcie1032              40960  0
apcie1564              40960  0
```

CentOS Stream 10

CentOS uses RPM packages and requires EPEL to be enabled.

1. **Enable EPEL repository**

```
sudo dnf install epel-release
```

2. **Install dependencies**

```
sudo dnf install dkms glibc-devel kernel-devel cmake make gcc-c++
```

3. **Install Addi-Pack**

   Download the latest package (`.rpm`) from ADDI-DATA website (https://www.addi-data.com/drivers).

   Example: `AddiPack-2025.09.1-Linux-x86_64.rpm`

```
sudo rpm -ivh AddiPack-2025.09.1-Linux-x86_64.rpm
```

4. **Check device detection**

   Compile and run a sample:

```
cd $AddiPack_ROOT/samples/get_devices_info
cmake -B build && cmake --build build
./build/get_devices_info
```

⚠ Warning

Some CentOS systems use old Python versions. Upgrade your Python version to 3.8+ (previous versions aren't supported by API)

```
sudo dnf install python3.8
```

Verify Installation

After installing Addi-Pack and inserting the hardware, you should verify that the device is correctly detected by the system.

You can perform this check using either the CLI sample or the Python API, depending on your workflow.

Using the CLI sample

The *get_devices_info* utility, located in the *samples* directory, lists all detected ADDI-DATA devices and displays their hardware properties. To run it, follow the OS-specific commands shown in the installation sections above.

A typical output looks like:

```
========== Device 0 (0) ==========
Product Name: APCIe-1032
Bus: 3 | Device: 0 | Vendor ID: 0x15B8 | Device ID: 0x1032
Interrupt: 11 | Bus Type: PCIe

BAR0: 0xFE800000
BAR3: 0xFE840000

Digital Inputs: 32 | Outputs: 0 | Digital Ports: 2 | Interrupt Mask:
0x0000FFFF
```

This confirms that the driver is loaded, and the device is properly detected.

Note

If no device is listed, make sure that:

- The card is correctly inserted and securely fastened,
- The driver was installed with Administrator rights.

Warning

If `AddiPack_ROOT` is missing or incorrect, CMake, Python, or SDK tools may fail to locate Addi-Pack components.

Using Python

You can also check for detected devices programmatically using the *addipack* Python module:

```
from addipack import DeviceDiscovery
devices = DeviceDiscovery.get_devices()
for dev in devices:
   print(dev)
```

This will list the same devices returned by the CLI utility, with detailed information accessible via the API.

Ready-to-use Samples

After installation, explore our samples located in `$AddiPack_ROOT/samples` . Binaries are directly available for Windows users under `bin/x64/generic` or `bin/win32/generic` if using a 32 bits PC. Each sample source code is available under their respective subfolder.

Folder Name	Purpose
`get_devices_info`	Lists all detected cards and their capabilities. (only available in C)
`digital_io`	Contains three samples, one for inputs, one for outputs and one for ports configuration.
`timer`	Demonstrates timer start, stop and read operations.
`counter`	Show pulse counting and interrupt handling.
`watchdog`	Demonstrates watchdog timer setup and expiration behavior.

Installation Known Issues

Issue	Solution
Driver conflicts (Windows)	Uninstall any previous AddiPack driver version, reboot.
Driver conflicts (Linux)	Remove conflicting drivers such as Comedi, then reload the Addi-Pack DKMS module.
Card not detected	Check hardware slot, BIOS, or driver status. Try moving the card to a different slot if available.
Dependency errors (Linux)	Install dkms, glibc-devel, and kernel headers.
Missing environment variable	Add `AddiPack_ROOT` to `/etc/environment` manually.
Unsigned driver warning (Windows 11)	Use the official signed installer.
Wrong permissions on device files	Adjust permissions with `chmod` , or configure udev rules.

Next sections:

- Continue to API Programming Guide to start developing with Addi-Pack.
- Learn driver and interrupt mechanisms in Driver Concepts Guide .
- Try real interactive examples in Addi-Pack Samples .

API Programming Guide

Overview

The **Addi-Pack API** provides a unified programming interface for all supported **ADDI-DATA devices** . It offers high-level access to hardware functions through a consistent set of C, C++, C# and Python APIs.

You can use the API to:

- Enumerate and select connected cards
- Read or write **digital** and **analog I/O**
- Configure and control **timers** , **counters** , and **watchdogs**
- Manage **interrupts** and handle callbacks
- Monitor **status** , **errors** , and **card information**

Tip

The same API calls work seamlessly across **Windows** , **Linux** , and all ADDI-DATA devices (PCI, PCIe and CPCI/CPCIs).

Language Interfaces

Addi-Pack provides equivalent API's for three languages:

Language	Description
C	Low-level procedural API via `addi-data/addi-pack/addi-pack.h`
C++	Modern object-oriented API under the `addipack::` namespace
C#	Modern object-oriented API with a garbage collector under the `AddiPack` namespace.
Python	High-level bindings generated from the C++ API

All bindings follow the same logical structure and naming convention:

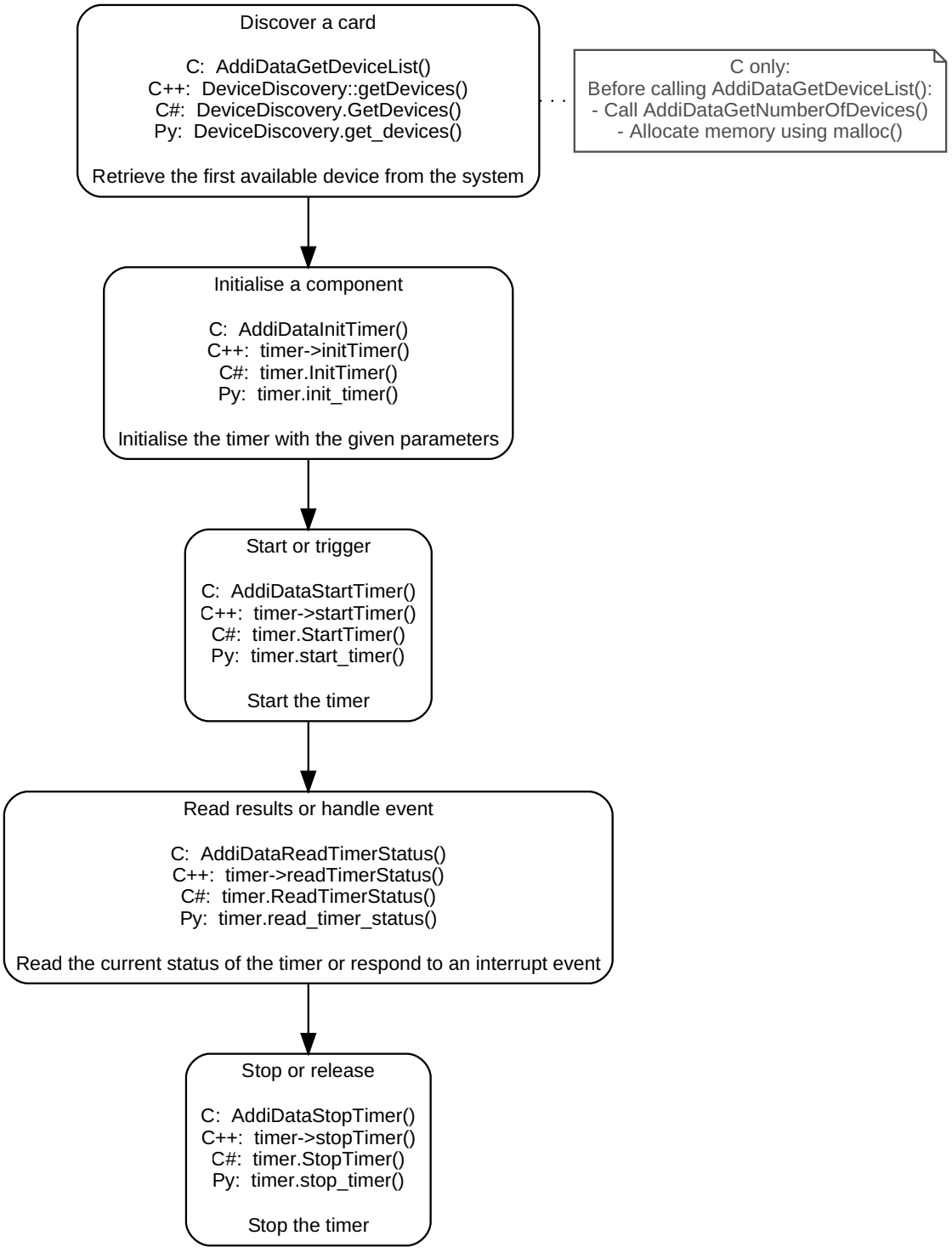
- C: `AddiData<Action><Component>()`
- C++: `component->actionComponent()`
- C#: `component.ActionComponent()`
- Python: `component.action_component()`

Example mappings:

	Init a timer	**Read a digital input**	**Start a counter**
C	AddiDataInitTimer()	AddiDataReadSingleDigitalInput()	AddiDataStartCounter()
C++	timer->initTimer()	digital->readSingleDigitalInput()	counter->startCounter()
Python	timer.init_timer()	digital.read_single_digital_input()	counter.start_counter()
C#	timer.InitTimer()	digital.ReadSingleDigitalInput()	counter.StartCounter()

Typical Workflow

The general API workflow is the same across all languages:



Typical Addi-Pack API workflow (Timer example)

Examples

C

```c
#include <addi-data/addi-pack/addi-pack.h>
#include <stdio.h>

int main() {
   uint8_t number_of_cards = 0;
   AddiDataGetNumberOfDevices(&number_of_cards);

   AddiDataDeviceStruct* device_list =
malloc(sizeof(AddiDataDeviceStruct) * number_of_cards);
   if (!device_list || AddiDataGetDeviceList(device_list) !=
ADDIDATA_SUCCESS_CODE) {
      free(device_list);
      exit(EXIT_FAILURE);
 }

   uint8_t timer_id = 0;
   uint16_t reload_value = 500;
   TIMEBASE_UNITS timebase = MILLISECOND;
   uint32_t options = SINGLE_CYCLE;

   AddiDataInitTimer(&device, timer_id, timebase, reload_value,
options);
   AddiDataStartTimer(&device, timer_id);

   uint32_t value = 0;
   uint8_t status = 0, triggered = 0, has_expired;

   AddiDataReadTimerStatus(id, timer_id, &value, &status,
&triggered);

   for (int i = 0, i < reload_value; i++) {
      printf("Timer: value=%u | status=%u | triggered=%u |
expired=%u\r", value, status, triggered, has_expired);
      fflush(stdout);
      usleep(1000);
   }

   AddiDataStopTimer(id, timer_id);
   return 0;
}
```

C++

```
#include <addipack/device_discovery.hpp>
#include <addipack/timer_interface.hpp>
#include <chrono>
#include <thread>

int main() {
    auto devices = DeviceDiscovery::getDevices();
    auto timer = device->timer();

    uint8_t timer_id = 0;
    uint16_t reload_value = 500;
    TIMEBASE_UNITS timebase = MILLISECOND;
    uint32_t options = SINGLE_CYCLE;

    timer->initTimer(timer_id, timebase, reload_value, options);
    timer->startTimer(timer_id);

    uint32_t value = 0;
    uint8_t status = 0, triggered = 0, has_expired;

    std::this_thread::sleep_for(std::chrono::milliseconds(200));
    timer->readTimerStatus(timer_id, value, status, triggered,
has_expired);

    std::cout << "Timer: value=" << value << " | status=" <<
int(status)
              << " | triggered=" << int(triggered) << " | expired="
<< int(has_expired) << std::endl;

    timer->stopTimer(timer_id);
    return 0;
}
```

Python

```python
import time
from addipack import (
   DeviceDiscovery,
   TIMEBASE_UNITS
)

devices = DeviceDiscovery.get_devices()
timer = device.timer()

timer_id = 0
reload_value = 500
timebase = TIMEBASE_UNITS.MILLISECOND
options = ADDI_DATA_TCW_OPTIONS.SINGLE_CYCLE

timer.init_timer(timer_id, timebase, reload_value, options)
timer.start_timer(timer_id)
print("Timer started.")

time.sleep(0.1)
value, status, triggered, has_expired =
timer.read_timer_status(timer_id)
print(f"Timer {timer_id} - value: {value}, started: {status},
trigger: {triggered}, expired: {has_expired}")

timer.stop_timer(timer_id)
```

C#

```csharp
using AddiPack;

byte timerId = 0;
ushort reloadValue = 500;
TimebaseUnits timebase = TimebaseUnits.Millisecond;

IReadOnlyList<Device> devices = DeviceDiscovery.GetDevices();
Device? device = devices[0];
AddiPack.Timer timer = device.Timer;

timer.InitTimer(timerId, timebase, reloadValue, 0);
timer.StartTimer(timerId);
Console.WriteLine($"Timer {timerId} started. Press ENTER to stop.
\n");

while (!Console.KeyAvailable)
{
   TimerStatus status = timer.ReadTimerStatus((byte)timerId);
```

```
    Console.Write($"\rValue: {status.Value,10}
started={status.IsStarted}  expired={status.HasExpired}    ");
    Thread.Sleep(200);
}
Console.ReadKey(intercept: true);

timer.StopTimer(timerId);
```

In order for the C# bindings to work, you need to add explicitly the NuGet package to your project. This package is located in the `$AddiPack_ROOT/csharp` folder.

Note

To use our samples and C# API, you need to ensure that you installed .NET core SDK >= 8.0

Install on Linux: https://learn.microsoft.com/en-us/dotnet/core/install/linux

Install on Windows: https://learn.microsoft.com/en-us/dotnet/core/install/windows

Once `dotnet` command is available on your system, you need to define a package source from which your NuGet package will be loaded.

bash .. code-block:: bash

```
# You can verify the sources already used by nuget dotnet nuget list source

dotnet nuget add source $AddiPack_ROOT/csharp -n “My local source”

cd $AddiPack_ROOT/samples/csharp/get_device_info

# — For .NET < 9.0 — dotnet add package AddiPack –source “$AddiPack_ROOT/csharp”
# — For .NET >= 9.0 — dotnet package add AddiPack –source “$AddiPack_ROOT/csharp”

dotnet run
```

powershell .. code-block:: console

```
# You can verify the sources already used by nuget dotnet nuget list source

dotnet nuget add source $env:AddiPack_ROOTcsharp -n “My local source”

cd $env:AddiPack_ROOTsamplescsharpget_device_info

# — For .NET < 9.0 — dotnet add package AddiPack –source “$env:AddiPack_ROOTcsharp”
```

```
# — For .NET >= 9.0 — dotnet package add AddiPack –source "$env:AddiPack_ROOTcsharp"

dotnet run
```

Error Handling

All Addi-Pack C functions return a **32-bit status code** that encodes where and why an error occurred.

Error Code Layout

The error code is divided into 5 main fields:

Bits	Field	Size	Description
31	**Source**	1 bit	0 = Driver, 1 = API
30–24	**Domain**	7 bits	Functional area (Timer, Counter, Digital I/O...)
23–16	**Reserved**	8 bits	Reserved for future use
15–8	**Function**	8 bits	Function identifier (Init, Start, Stop...)
7–0	**Reason**	8 bits	Detailed cause (Invalid, Out of range...)

Example

Return codes are universal — they are interpreted the same way in C, C++, and Python (where exceptions include the code).

Error Code (Hex)	Meaning
`0x00000000`	**Success** – Operation completed successfully (`ADDIDATA_SUCCESS_CODE`).
`0x83001203`	**API / TIMER / INIT / INVALID_PARAMETER** – Invalid argument passed to `AddiDataInitTimer()` .
`0x82001905`	**API / DIGIO / WRITE_DIGOUT / NOT_SUPPORTED** – Digital output operation not supported on this device.
`0x04001309`	**Driver / WATCHDOG / START / INVALID_STATE** – Watchdog cannot be started (invalid or inactive state).
`0x85000608`	**API / INTERRUPTS / CREATE_INTERRUPT / HARDWARE_FAILURE** – Hardware refused interrupt creation.
`0x81001804`	**API / COUNTER / READ_STATUS / OUT_OF_BOUNDS** – Counter index outside the valid range.
`0x8000000A`	**API / GENERIC / NONE / CONFIG_ERROR** – Configuration issue detected in API call.

Note

This list shows only a few representative examples. The complete list of all error codes is available in the Common Error Codes .

Working with Structures and Types

Structure *AddiDataDeviceStruct*

Describes a detected ADDI-DATA device and its capabilities.

This structure is returned by discovery functions such as `AddiDataGetDeviceList()` . It contains hardware information like the product name, PCI/PCIe location, and supported features (timers, counters, digital I/O, etc.).

Only the `id` field (a `const char*`) is used in API calls. This string uniquely identifies the device and must be passed to all `AddiData*()` functions.

```
typedef struct AddiDataDeviceStruct {
    char id[ADDI_STRING_SIZE];
    AddiDataGeneralInformations general;
    AddiDataFunctionInformations functions;
    AddiDataPCIBusInformations bus;
} AddiDataDeviceStruct;
```

You obtain it through:

```
AddiDataDeviceStruct device;
device->general.product_name

Example:
printf("\n========== Device(%s) ==========\n", device->id);
printf("Product Name: %s\n", device->general.product_name);
printf("Bus: %u | Device: %u | Vendor ID: 0x%04X | Device ID:
0x%04X\n", device->bus.bus_number,
        device->bus.device_number, device->bus.vendor_id,
device->bus.device_id);
printf("\nDigital Inputs: %u | Outputs: %u | Digital Ports: %u |
Interrupt Mask: 0x%08X\n",
            device->functions.digital.number_inputs,
device->functions.digital.number_outputs,
            device->functions.digital.number_of_ports,
device->functions.digital.interrupt_mask);
```

Timer / Counter / Watchdog Identifiers

Each ADDI-DATA card exposes a set of timers, counters, and watchdogs, identified by **zero-based IDs** .

The number of available components depends on the specific card model, and is provided dynamically through the discovery structure.

To determine valid IDs, use the fields inside `AddiDataDeviceStruct` :

```
uint8_t max_timer_id = device.functions.timer.number_timers - 1;
uint8_t max_counter_id = device.functions.counter.number_counters -
1;
uint8_t max_watchdog_id = device.functions.watchdog.number_watchdogs
- 1;
```

For example, if `number_timers == 3` , then the valid timer IDs are `0` , `1` , and `2` .

Card	Timers ID
APCI-1500	Timers 0–2
APCIe-1564	Timers 0–1
APCIe-1032	Timer 0
APCI-1032	None

Example usage:

```
AddiDataStartTimer(&device, 0);  // Start timer 0
```

To determine how many are available:

Use the sample get_devices_info to query the detected card and list its capabilities (number of timers, counters, watchdogs, digital inputs/outputs, ports, etc.).

Run (CLI):

Linux

```
$AddiPack_ROOT/samples/get_device_info/build/get_device_info
```

Windows (PowerShell)

```
& "$env:AddiPack_ROOT\samples\bin\x64\generic\get_devices_info.exe"
```

The output includes cards summary with counts for timers/counters/watchdogs and digital I/O.

Digital I/O Channels

Digital channels are identified numerically and can be accessed either individually (bit-level) or collectively (port-level).

1. Single-bit access

   Use this when you want to read or write a specific channel:

```
uint8_t value = 0;
AddiDataReadSingleDigitalInput(device.id, 4,
&value);  // Read input channel 4

AddiDataWriteSingleDigitalOutput(device.id, 7,
ADDI_DATA_ENABLE);     // Set output channel 7 to HIGH
```

2. Multi-bit (mask-based) access

   This allows reading or writing all digital inputs or outputs at once using a bitmask:

```
uint32_t inputs = 0;
AddiDataReadAllDigitalInputs(device.id,
&inputs);     // Read all 32 inputs

AddiDataWriteAllDigitalOutputs(device.id, 0x0000FFFF); // Set first
16 outputs to HIGH
```

3. Digital Input Events (Interrupts)

   Some ADDI-DATA cards support **event-based monitoring** of digital input lines. This allows the application to react to signal changes instead of polling continuously.

   To configure digital input interrupts:

   1. Select the **input port** to monitor (e.g., Port 0 or 1),
   2. Define the **bitmask** of input lines to watch,
   3. Choose the **event type** (rising, falling, any edge, on zero (state off), on one (state on)),
   4. Select the **logic mode** (trigger if *any* line matches with logic **OR** , or *all* with logic **AND**),
   5. Enable digital and device-level interrupts.

4. Digital Output Events (VCC / CC Interrupts)

Digital output interrupts are triggered when the board detects:

- **VCC interrupt** : power supply failure on digital outputs
- **CC interrupt** : short-circuit condition on digital outputs

These events are card-level safety mechanisms and are independent of individual output channel states.

Example (C code):

```
uint8_t port = 0;
uint32_t mask = 0x00000010; // Channel 4
DIGITAL_EVENT_TYPE event = RISING_EDGE;
DIGITAL_EVENT_LOGIC logic = OR;

AddiDataSetInterruptEventForInputs(device.id, mask, event);
AddiDataSetInterruptLogicForPort(device.id, port, logic);
AddiDataSetDeviceInterruptCallback(device.id, &MyCallback);
AddiDataEnableDeviceInterrupts(device.id);
AddiDataEnableDigitalInterrupt(device.id);
```

Note

- **Event logic is configured per port**, not per channel.
- Cards may support different **event types**: `RISING_EDGE`, `FALLING_EDGE`, `ANY_EDGE`, `ON_ZERO`, `ON_ONE`.
- **Logic mode** (`OR` or `AND`) determines if the interrupt fires on *any* active line, or *only when all* specified inputs match.

Tip

You can retrieve the number of available ports and the interrupt mask using the discovery structure.

```
uint32_t interrupt_mask =
selected_device->functions.digital.interrupt_mask;
uint8_t max_port =
selected_device->functions.digital.number_of_ports;
```

Enumerations

Enumerations standardize parameters such as timebases, modes, and event types.

Common enumerations:

Enumeration	Purpose
`TIMEBASE_UNITS`	Defines timer units (µs, ms, s)
`ADDI_DATA_TCW_OPTIONS`	Timer/Counter/Watchdog initialization options
`DIGITAL_EVENT_TYPE`	Rising/Falling edge, Level High/Low
`DIGITAL_EVENT_LOGIC`	Logical combination (AND / OR)

Example:

```
TIMEBASE_UNITS timebase = MILLISECOND;
AddiDataInitTimer(&device, 0, timebase, 1000, 0);
```

Next section: Driver Concepts Guide — Learn how drivers, interrupts, and DMA work internally.

Driver Concepts Guide

Overview

At the core of **Addi-Pack** lies a powerful driver layer designed to manage all communication between your application and ADDI-DATA hardware — across **Windows (KMDF)** , **Linux (DKMS)** and real-time environments such as **IntervalZero RTX** and other **RTOS-based systems** .

Understanding how this layer works helps you optimise performance, reliability, and responsiveness.

This section explains:

- The difference between **polling** and **interrupts**
- How **device identification** and **card selection** work
- How to use Compatibility DLLs to support applications built with the older API
- The fundamentals of **timers** , **counters** and **watchdogs**

Tip

Mastering these concepts is especially useful for high-performance or multi-card systems.

Driver Architecture

Addi-Pack drivers follow a **two-layer architecture** :

1. **Kernel Layer (Driver)**
    - Handles PCI/PCIe bus communication, hardware interrupts, and DMA transfers.
    - Implemented as a **KMDF** driver on Windows and a **DKMS** module on Linux, and compatible with **real-time extensions** such as **IntervalZero RTX** and other **RTOS environments** .
2. **User Layer (API)**
    - Provides a unified programming interface in **C** , **C++** , and **Python** .
    - Communicates with the driver using **IOCTL calls** under the hood.
    - Exposes high-level features like timers, counters, digital I/O, and event handling.

Interrupts vs Polling

Most Addi-Pack features support both **polling** and **interrupt-driven** modes.

Mode	Description
Polling	The application checks periodically for a state change. Simple to use but inefficient for high-frequency tasks.
Interrupts	The hardware notifies the application only when an event occurs. Efficient and real-time capable. Recommended for timers and counters.

Example scenarios:

- **Digital I/O:** Polling is acceptable when reading every 100–200 ms.
- **Timers / Counters:** Prefer **interrupts** to avoid missing fast events.

Why Use Interrupts?

Interrupts are essential when you need to:

- **React to fast or unpredictable events** without missing them,
- **Reduce CPU usage** by avoiding constant polling,
- **Handle multiple events** from different subsystems (inputs, timers, counters).

Without interrupts, your application would need to poll the hardware in a tight loop — which can:

- Miss short pulses or transitions between reads,
- Consume unnecessary CPU cycles,
- Become difficult to scale or integrate with other tasks.

Using interrupts ensures that your application is **notified precisely when something happens** , allowing you to respond immediately and efficiently.

Typical examples where interrupts are critical:

- **Timer expiration** : Trigger an action exactly every N milliseconds,
- **Counter overflow** : React when a threshold is reached,
- **Digital input change** : Detect a rising edge on an external signal (e.g., sensor, button).

- **Digital output fault detection** : Detect a **power supply failure (VCC)** or a **short-circuit (CC)** on digital outputs.

Understanding Interrupt Handling

To use interrupts, you need to:

1. **Enable device-level interrupt support**

   (`AddiDataEnableDeviceInterrupts(id)`)

2. **Configure the interrupt source** For example, select which **inputs or timers** should trigger an interrupt.

   (`AddiDataEnableTimerInterrupt(id, timer_id)`)

3. **Set the interrupt mask** This defines **which bits (channels)** will be monitored by the hardware.

4. **Register a callback** In interrupt mode, your application will receive a notification function to handle events.

   (`AddiDataSetDeviceInterruptCallback(id, &interruptCallback)`)

5. **Process the event** In the callback, you typically:

   - Check the cause of the interrupt (e.g., which input changed),
   - Recording the time between interrupts
   - Read values (status, input lines, timer/counter),
   - Optionally acknowledge or reset the event flag.

Note

The interrupt mask is a bitfield that selects **which channels or sources** are monitored.

For example, `0x00000001` monitors channel 0, `0x0000000F` monitors channels 0–3.

C Example:

```
void my_callback(AddiDataDeviceStruct device, InterruptData data) {
   printf("[INTERRUPT] source=0x%X | mask=0x%X\n",
         data.interrupt_source, data.interrupt_mask);

   interrupt_happened_flag = 1;
}
```

```
AddiDataDeviceStruct device;
const char* id = device.id;

AddiDataEnableDeviceInterrupts(id);          // Enable global card
interrupts
AddiDataSetInterruptEventForInputs(id, 0x10, RISING_EDGE);  //
Monitor input 4
AddiDataSetInterruptLogicForPort(id, 0, OR);
AddiDataSetDeviceInterruptCallback(id, &my_callback);
AddiDataEnableDigitalInterrupt(id);
```

Tip

All information about the device is stored in the `AddiDataDeviceStruct` structure, which is filled by `AddiDataGetDeviceList()` .

Always check this structure before enabling interrupts to know:

- Which inputs/ports are available,
- The supported interrupt mask (e.g., `0xFFFF` , `0xFFFF0` , etc.),
- Whether the card supports digital events or not.

Device Identification

Each ADDI-DATA card has a **unique device ID** automatically assigned at system startup. You can list all connected cards and retrieve their properties using the **`get_device_info`** sample.

CLI Example:

```
$AddiPack_ROOT/samples/get_device_info/build/get_device_info
```

Information shown includes:

- Device name and model (e.g. *apcie1500*)
- PCI bus and slot
- Supported functions (digital I/O, timer, counter, watchdog)
- Driver version and status

This information can be used to:

- Select a specific card in multi-device systems
- Confirm that all driver modules are properly loaded

Want to configure multiple timers with different interrupts?

```
TIMEBASE_UNITS timebase = MILLISECOND;
AddiDataInitTimer(&device, 0, timebase , 100, 0);
AddiDataInitTimer(&device, 1, timebase , 250, 0);

AddiDataEnableTimerInterrupt(&device, 0);
AddiDataEnableTimerInterrupt(&device, 1);
```

Compatibility DLLs (older API Support)

Some users already operate software built using the **former Windows KMDF driver** and its associated C API, providing functions such as:

- `i_PCI1032_Read1DigitalInput()`
- `i_PCI1564_Set1DigitalOutputOn()`

To support these use cases, Addi-Pack provides **Compatibility DLLs** that expose the same function names and structures as the former API, while internally redirecting calls to the modern Addi-Pack API.

Note

If the code uses functions with an 'A' before PCI (e.g. i_APCI1500_SetBoardIntRoutineWin32), the Compatibility DLL delivered in the present ADDI-PACK software will not be enough to run the associated code. If it is needed to run a former code using said functions, contact ADDI-DATA. (**info@addi-data.com**)

How It Works

When a program calls the old API:

```
i_PCI1032_Read1DigitalInput()
```

the Compatibility DLL transparently redirects this call to:

```
AddiDataReadSingleDigitalInput()
```

The original function names stay valid, but the execution now uses the **new Addi-Pack driver** .

Why This Matters

This mechanism allows users to:

- Keep existing Windows applications running **without rewriting a single line of code** ,
- Move from the legacy KMDF driver to Addi-Pack safely,
- Replace hardware (PCI → PCIe) while keeping the same software stack,
- Migrate gradually to the new API.

How to Use Compatibility DLLs (Windows)

1. Install Addi-Pack normally.
2. The Compatibility DLLs are installed in system32
3. Run the program normally — no source code changes required.

> **Note**
>
> For all new development, the native Addi-Pack C, C++, or Python API is recommended for improved maintainability, performance, and cross-platform behaviour. The Compatibility DLL only ensures that *existing code keeps working with the same model it was written for* , but under Addi-Pack.

Use PCI Samples (Old Windows KMDF) on PCIe Hardware (Windows) using PCI DLL

This feature lets the user keep their code unchanged when switching from APCI-XXXX to APCIE-XXXX. With this feature, the user can allow PCIe to be used with PCI functions and their DLL. By default, this feature is disabled (Windows 32/64 Bits version). To enable it, the user has to manually configure a Windows Register.

To enable the feature, the user can follow these steps :

- Install Addi-Pack as described in the section Installation Guide
- Open *Run* console by using shortcut *WIN+R* and type *regedit* (admin rights are needed).
- **Once** *regedit* **opened, edit the register named** *Compatibility DLL UsePCIeAsPCI* **. Depending on the architecture, and on the binary that needs to run, the register has a different location.**
    - **Native Windows execution** (64-bit executable running on Windows 64-bit, or 32-bit executable running on Windows 32-bit) : The register to update is located under the path : *HKEY_LOCAL_MACHINE\SOFTWARE\Addi-Data GmbH\AddiPack\CompatibilityDLL UsePCIeAsPCI*
    - **Windows on Windows 32-Bit execution** (32-bit executable on a Windows 64-bit) : The register to update is located under the path : *HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Addi-Data GmbH\AddiPack\CompatibilityDLL UsePCIeAsPCI*
- To enable the feature, the register must be set to **1** . Otherwise, the register is set to **0** by default, which disables the feature.

Fundamentals

Timer

A timer generates periodic events based on a configured timebase (µs/ms/s) and reload value. When the timer expires, it reloads and optionally triggers an interrupt.

In Addi-Pack, a timer is configured by:

- selecting a timebase,
- choosing a reload value,
- applying optional modes (single-cycle or continuous),
- then starting the timer.

Timers are ideal for periodic sampling, recurring control loops, or triggering regular actions (e.g., every 1 ms, 10 ms, or 100 ms).

Counter

A counter increments based on external pulses coming from a digital input. It is used to measure frequency, count events, or track edge transitions from sensors, encoders, or pulse sources.

In Addi-Pack, a counter is configured by:

- selecting the counter index,
- setting a reload value,
- setting options if needed like edge type (rising, falling, or both) or optional gate/trigger modes (depending on the card),
- then starting the counter.

Each time the counter reaches its reload value, it can optionally raise an interrupt. The current value and status can be read at any time.

Counters are ideal for pulse counting, speed or frequency measurement, encoder signals, or any application that needs precise tracking of external events.

Watchdog

A watchdog is a safety timer that must be refreshed (“kicked”) before its configured timeout expires. If the application fails to refresh it in time, the watchdog triggers an interrupt to signal a stall, deadlock, or timing violation.

In Addi-Pack, a watchdog is configured by:

- selecting the watchdog index,
- choosing a timebase (µs/ms/s),
- setting a reload value (timeout),
- applying optional modes depending on the board,
- then starting the watchdog.

Watchdogs are ideal for detecting stalled loops, monitoring task deadlines, and ensuring that critical cyclic processes continue to run correctly.

Next sections:

- Head over to Addi-Pack Samples to try interactive examples for timers, counters, and digital I/O.

Addi-Pack Samples

Overview

This section contains detailed explanations of every **Addi-Pack sample program** provided with Addi-Pack. Samples are provided in **C** , **C++** , **C#** and **Python** , and demonstrate how to interact with ADDI-DATA cards through the API in both **interactive** and **automated** ways.

Each sample:

- Is **interactive by default** (if no command-line arguments are provided),
- Can also be run in **fully scripted or automated mode** using command-line options,
- Is designed to be a **complete, working program** ready to build and execute.

Sample Utilities

Goal

Provide common functions used by all samples. Such as user prompts, argument parsing, device selection, and interrupt handling.

These utilities form the foundation of all Addi-Pack samples.

The SampleUtils and ArgParser modules have distinct responsibilities:

- **ArgParser** Provides the command-line argument interface used by the samples. It handles: - parsing options such as `--mode=interrupt` or `--card=apcie1032` , - checking whether an option or flag is present, - retrieving option values (e.g. `--timebase ms`), - validating argument syntax.
- **SampleUtils** Implements all interactive behaviour of the samples. It is responsible for: - deciding whether to run in scripted or interactive mode (based on whether an option is set), - prompting the user when an argument is missing or invalid, - selecting a device according to capabilities, - selecting modes, - validating numeric and hexadecimal parameters, - helper utilities (interrupt callback, timing helpers, exit helpers, etc.).

Together, these modules allow each sample to support both automated command-line execution and interactive question-based execution. They are not part of the public API, but they can be reused for rapid prototyping.

Tip

Even though *ArgParser* is technically a private helper, you can safely use it in your own C/C++ projects to build interactive CLI tools consistent with the official samples.

Using it has several benefits:

- **No external dependency** : the parser is fully embedded in Addi-Pack and does not require any third-party library.
- **Cross-platform behavior** : the same command-line arguments work identically on Windows and Linux.
- **Consistent user experience** : your tools behave exactly like the official samples (same options, same style, same error handling).
- **Faster prototyping** : no need to implement argument parsing, validation or help messages yourself.
- **Reliable and tested** : the parser is used internally by all ADDI-DATA samples, ensuring stable behavior.

Key Concepts

1. **Interactive vs Non-Interactive Mode**
   - If the user passes options (e.g. *–reload=1000 –mode=polling*), the sample runs automatically.
   - Otherwise, prompts are displayed dynamically to ask for missing parameters.
2. **Automatic Device Selection**

   The function *AddiDataUtilsSelectDeviceByCapability()* filters all detected devices and selects one based on user input or CLI arguments.
3. **Input Validation**

   Utilities like *AddiDataUtilsGetValidInt()* or *AddiDataUtilsGetValidString()* ensure that all parameters are valid before starting the test or operation.
4. **Interrupt Management**

   Functions such as *AddiDataUtilsInterruptCallback()* and *AddiDataUtilsHasInterruptHappened()* are shared across interrupt-based samples.

C Utilities

Main header: `#include <addi-data/addi-pack/utilities/sample_utils.h>`

Below are the main utility functions available in **C** samples:

Function	Purpose
AddiDataUtilsPromptString()	Prompt the user for a string, with optional accepted values and default.
AddiDataUtilsPromptInt()	Prompt the user for an integer within a specified range *[min, max]* .
AddiDataUtilsPromptHex()	Prompt the user for a hexadecimal value within a specified range *[min, max]* .
AddiDataUtilsGetValidString()	Get a validated string from CLI args or fallback to prompt.
AddiDataUtilsGetValidInt()	Get a validated integer from CLI args or fallback to prompt.
AddiDataUtilsGetValidHex()	Get a validated hexadecimal value from CLI args or fallback to prompt.
AddiDataUtilsMatchesCardFilter()	Check whether a device matches a card filter string.
AddiDataUtilsSelectDeviceByCapability()	Select a device matching a predicate from a list of devices.
AddiDataUtilsSelectMode()	Select a string mode from a list, using CLI –*<name>* or prompt.
AddiDataUtilsParseTimebase()	Convert a timebase string (*“us”* , *“ms”* , *“s”*) into a *TIMEBASE_UNITS* enum value.

Function	Purpose
AddiDataUtilsInterruptCallback()	Default interrupt handler that logs the source, mask, and elapsed time since the last interrupt.
AddiDataUtilsResetInterruptFlag()	Reset the interrupt flag and internal timer.
AddiDataUtilsHasInterruptHappened()	Check whether an interrupt has occurred since the last reset.
AddiDataUtilsExitOnUserInput()	Wait for the user to press ENTER and exit the program.
AddiDataUtilsPrintAvailableTcwOptions()	Print a list of available TCW (Timer-Counter-Watchdog) options supported by the device.
AddiDataUtilsGetTcwOptionsFromUser()	Prompt the user to select TCW options interactively.
AddiDataUtilsGetValidTimebaseList()	Get a list of valid timebases supported by the device.
AddiDataUtilsDisplayAllPortsConfigurations()	Display the configuration (input/output) of all digital ports.

Example – Using the utilities in C

```
// Used to filter cards with at least one timer
int hasTimer(const AddiDataDeviceStruct* device) { return
device->functions.timer.number_timers > 0; }

// Initialise the argument parser
AddiDataArgParserInit();
AddiDataArgParserAddOption("card", 'c', "Filter by card name", 1);
AddiDataArgParserAddOption("reload", 'r', "Reload value", 1);
AddiDataArgParserParse(argc, argv);
```

```
// Get the number of available devices
uint8_t number_of_cards = 0;
AddiDataGetNumberOfDevices(&number_of_cards)

// Allocate and populate the device list
AddiDataDeviceStruct* devices = malloc(sizeof(AddiDataDeviceStruct)
* number_of_cards);
AddiDataGetDeviceList(devices);

// Select a device with at least one timer
const AddiDataDeviceStruct* device =
AddiDataUtilsSelectDeviceByCapability(devices, number_of_cards,
"card", hasTimer, "timer");

// Get a valid reload value from command-line or prompt
int reload = AddiDataUtilsGetValidInt("reload", "Enter reload
value", 1, 10000);

// Display selected card and reload value
printf("Using %s | Reload = %d\n", device->general.product_name,
reload);
```

C++ Utilities

In C++, the same utilities are provided via wrappers and helper classes.

Example – Using ArgParser and SampleUtils in C++

```
#include <addi-data/addi-pack/cpp/sample_utils.hpp>
#include <addi-data/addi-pack/cpp/timer_interface.hpp>

using namespace addidata;
using namespace addipack;
using namespace utils;

bool hasTimer(const std::shared_ptr<Device>& device) {
   return
device->getDeviceInformation().functions.timer.number_timers > 0;
}

int main(int argc, char* argv[]) {
    ArgParser::init();
    ArgParser::addOption("card", 'c', "Filter by card name");
    ArgParser::addOption("reload", 'r', "Reload value");
    ArgParser::parse(argc, argv);

    auto devices = DeviceDiscovery::getDeviceList();
    auto device = selectDeviceByCapability(devices, "card",
hasTimer, "timer");
```

```
    int reload = getValidInt("reload", "Enter reload value", 1,
10000);
    std::cout << "Selected device: " <<
device->getDeviceInformation().id << " | Reload = " << reload <<
std::endl;
}
```

Python Utilities

Example – Using SampleUtils in Python

```
from sample_utils import *

def has_timer(device):
   return device.get_card_information().functions.timer.number_timers
> 0

def parse_args():
 parser = argparse.ArgumentParser(description="example")
 parser.add_argument("-c", "--card", help="Filter device by name
(apcie1032, apci1500 ...)")
 parser.add_argument("-r", "--reload", type=int, help="Reload value")
 return parser.parse_args()

def main():
   args = parse_args()
   device = select_device_by_capability(args.card, has_timer,
"timer")
   info = device.get_card_information()
   reload_value = get_valid_int(args.reload, "reload", 1, 10000)

print(f"Selected {info.id} with reload={reload_value}")
```

C# Utilities

Example – Using SampleUtils in C#

```
using AddiPack;
using AddiPack.Samples;


string? card = SampleUtils.GetOption(args, "card", "c");
bool isInterruptModeSet = SampleUtils.IsFlagSet(args, "interrupt",
"i");
static bool HasTimer(Device d) =>
d.GetCardInformation().Functions.Timer.NumberTimers > 0;

AddiPack.Timer timer = device.Timer;
Interrupt interrupt = device.Interrupt;

int timers = info.Functions.Timer.NumberTimers;
int timerId = SampleUtils.GetValidInt(null, "Timer to use", 0,
timers - 1);

string[] validTimebases = new string[] { "us", "ms", "s", "c" };
string timebaseStr = SampleUtils.GetValidString(
```

```
   null,
"Enter timebase (us, ms, s). Enter c for custom, if the card supports
it.  [us/ms/s/c]: ", validTimebases);
int timebase = Array.IndexOf(validTimebases, timebaseStr);
ushort maxReload = info.Functions.Timer.MaxReloadValue[timebase];
ushort minReload = info.Functions.Timer.MinReloadValue[timebase];
int reload = SampleUtils.GetValidInt(null, $"Reload value",
minReload, maxReload);

Console.WriteLine($"Selected {timebase.ToString()} with
reload={reload}");
```

get_devices_info

Goal

List all detected ADDI-DATA cards connected to the system and display their detailed properties.

This is the **first sample to run after installation** , as it verifies that:

- The Addi-Pack drivers are correctly installed,
- The API can communicate with the hardware,
- Cards are properly enumerated and accessible.

This sample calls the core device discovery functions:

- *AddiDataGetNumberOfDevices()* → returns the number of cards detected
- *AddiDataGetDeviceList()* → fills a list of *AddiDataDeviceStruct* entries

It then prints for each card:

- Product name and type (PCI or PCIe)
- Bus information (bus, slot, vendor and device IDs)
- Function capabilities (digital I/O, timers, counters, watchdogs)
- Interrupt number and BAR addresses

The user can **filter cards interactively** (by name, type, or model) or directly using a **command-line argument** such as *–card=apcie1032* .

Key Features

1. **Dynamic card detection**

   Enumerates all ADDI-DATA devices automatically, regardless of model.

2. **Flexible filtering**

   - *–card=apcie* → lists only PCIe cards
   - *–card=apci* → lists only PCI cards
   - *–card=1032* → lists only 1032 cards

3. **Automatic behavior**

   - If only one card is found, it's automatically selected.
   - If multiple cards are found, the user can choose or filter interactively.

4. **Cross-platform output**

   Works identically on **Windows** and **Linux** , and displays unified information.

Example Usage

Interactive mode (default):

```
./get_device_info
One device found. Automatically selecting it.
========= Device 0 (apcie-1032) =========
Product Name: APCIe-1032
Bus: 3 | Device: 0 | Vendor ID: 0x15B8 | Device ID: 0x1032
Interrupt: 17 | Bus Type: PCIe
Digital Inputs: 32 | Outputs: 0 | Ports: 1 | Mask: 0xFFFF
Timers: 0 | Counters: 0 | Watchdogs: 0
```

Filtered mode:

```
./get_device_info --card=apci1500
```

It displays only matching cards (here apci1500).

digital_input

Goal

Read digital input channels in **polling** or **interrupt** mode.

This sample demonstrates how to acquire input states from ADDI-DATA cards, either by continuously reading the lines or by waiting for hardware events.

Concept and Purpose

The *digital_input* sample is designed to validate **input acquisition** and **interrupt configuration** on any card supporting digital inputs.

It serves both as:

- A **functional test** , confirming that inputs toggle and can trigger events.
- A **reference implementation** , showing how to configure edge detection, logic, and masks.

The sample operates in **two distinct modes** :

- **read** → polling mode: the application continuously reads and displays input states.
- **interrupt** → event-driven mode: the program waits for hardware-triggered input changes.

Execution Workflow

1. **Device detection and selection**

   The sample first detects all installed ADDI-DATA cards with input capability using *AddiDataUtilsSelectDeviceByCapability()* . If multiple compatible cards are found, the user can select one interactively or provide it via the command line (e.g. *–card=apcie1032*).

2. **Mode selection**

   The user chooses between:

   - *read* mode → constant polling,
   - *interrupt* mode → asynchronous event handling.

   This can be specified via *–mode=* or through an interactive prompt.

3. **Configuration phase (interrupt mode)**

   When running in **interrupt mode** , several parameters are configured:

   - **Port index** (*–port*): selects which input port to monitor.
   - **Interrupt mask** (*–mask*): selects which input bits trigger events.
   - **Event type** (*–event*): rising/falling/any edge or level-based detection.
   - **Logic** (*–logic*): whether multiple active inputs combine with *OR* or *AND* .

   The sample validates all these parameters and applying interactive prompts as needed.

4. **Runtime phase**

   - In **read mode** , the sample prints live input states in binary form. The screen is refreshed continuously until the user presses **ENTER** .
   - In **interrupt mode** , the sample:
     - Registers the default interrupt callback *AddiDataUtilsInterruptCallback* .
     - Enables device and digital interrupts.
     - Prints a message every time an interrupt occurs, including:
       - the source,
       - the input mask,
       - and the elapsed time since the previous interrupt.

5. **Termination**

When the user presses **ENTER** , the sample:

- Disables all interrupt sources,
- Restores device state,
- Frees allocated resources,
- And exits cleanly.

Runtime Behavior

In read mode: - The program polls all input lines periodically (every 100 ms by default). - Values are printed as a live binary bit field:

- **Example output:**
  Inputs: 0000000000001111

- Press **ENTER** to stop reading.

In interrupt mode: - The card triggers an interrupt whenever the configured event occurs on one of the selected bits. - Each interrupt produces a console message such as:

```
[INTERRUPT] Callback: source=0x02 | mask=0x0004 |
elapsed=12.45 ms
```

- The *elapsed* field measures the delay between consecutive events, helping assess response time and signal stability.

digital_output

Goal

Control and test digital output lines on ADDI-DATA cards, either **individually** or **as a group** .

This sample demonstrates how to write digital output values using the Addi-Pack API, in both interactive and scripted (CLI) modes.

Concept and Purpose

The *digital_output* sample is designed to validate the correct functioning of the **digital outputs** on any compatible card. It allows developers or test engineers to quickly verify that all output lines can be toggled as expected.

It provides two complementary operating modes:

- **single mode:** interactively set individual output channels one by one.
- **mask mode:** apply a bitmask to modify multiple outputs simultaneously.
- **interrupt mode:** monitor hardware fault events on digital outputs

Execution Workflow

1. **Device detection and selection**

   The sample first detects all installed ADDI-DATA devices using *AddiDataUtilsSelectDeviceByCapability()* , filtering only those with output capability. If several compatible cards are present, the user can choose interactively or provide a *–card* filter (e.g. *–card=apci1564*).

2. **Mode selection**

   The user selects between:

   - *single* → manually set each output line.
   - *mask* → apply one value to all lines defined by a bitmask.
   - *interrupt* → enable and monitor on digital output interrupts.

   This can be done interactively or using *–mode=single* / *–mode=mask* .

3. **Configuration phase**

   Depending on the chosen mode, the sample requests or parses additional parameters:

   - **Mode = single**
     - Channel index to modify (*0* → *number_outputs - 1*)
     - Output value to write (*0* or *1*)
   - **Mode = mask**
     - Output mask (*–mask=0xFF*): defines which bits to affect
     - Value (*–value=0* or *–value=1*): sets all selected outputs high or low

- Mode = interrupt
  - Output interrupt mask (*–mask*):
    - Bit *0x01* → enable **VCC (power supply failure)** interrupt
    - Bit *0x02* → enable **CC (short-circuit)** interrupt

4. **Output operation**
   - For **single mode** , the program loops indefinitely, asking the user for a channel and a value until they press **ENTER** .
   - For **mask mode** , the selected mask is written once with the chosen value.
   - For **interrupt mode** , the program registers the default interrupt callback *AddiDataUtilsInterruptCallback* , enables device and output interrupts, and prints a message each time an interrupt occurs, including the source, mask, and elapsed time since the last event.

5. **Termination**

   When finished, all resources are released, and the program exits cleanly after waiting for user confirmation.

Runtime Behavior

Single Mode (interactive control)

- The user specifies one channel and a value (0 or 1).
- The output state is immediately applied and confirmed in the console.
- The loop continues until the user presses **ENTER** on an empty line.

Typical output:

```
Select output channel [0-15] (empty = quit): 3
Set value for this channel [0-1]: 1
Channel 3 set to 1

Select output channel [0-15] (empty = quit): 5
Set value for this channel [0-1]: 0
Channel 5 set to 0
```

Mask Mode (group operation)

- The user defines a hexadecimal mask and a single value (0/1).

- All output lines corresponding to bits set in the mask are updated simultaneously.

Example:

```
Enter output mask in hexadecimal (max 0xFFFF): 0x0F
Set value for the entire mask [0-1]: 1
Writing value 1 to mask 0x0F...
Digital outputs updated successfully.
```

In this example, the first four outputs (0–3) are set to logical 1.

digital_port

Goal

Configure and interact with digital I/O ports on ADDI-DATA cards.

This sample demonstrates how to:

- Discover devices and select one with digital I/O port capabilities
- Configure port directions (input/output)
- Read digital inputs from a port
- Write digital outputs to a port using a mask

It is designed to be a **complete, working program** ready to build and execute.

Key Features

- **Device Discovery and Selection:** The user can filter devices by name (e.g., “apci1696”) to find compatible hardware. The sample checks if the selected device has digital I/O ports.
- **Display Initial Configuration:** The sample reads and displays the current direction and status of all digital ports. It shows which ports are configured as inputs or outputs and their current values.
- **Port Direction Configuration:** The user can enter a hexadecimal mask to set the direction of all ports at once. For example, a mask of 0xF would set ports 0-3 as outputs and the rest as inputs.
- **Mode Selection:** The user can choose between “display” mode to continuously show port statuses or “write” mode to set outputs. In “display” mode, the sample refreshes the port

status every 200ms until the user presses ENTER. In “write” mode, the user selects a specific output port and provides a mask and value to set the outputs.

Workflow

1. **Device Discovery & Selection:** - The user can filter devices by name (e.g., “apci1696”) to find compatible hardware. - The sample checks if the selected device has digital I/O ports.
2. **Display Initial Configuration:** - The sample reads and displays the current direction and status of all digital ports. - It shows which ports are configured as inputs or outputs and their current values.
3. **Port Direction Configuration:** - The user can enter a hexadecimal mask to set the direction of all ports at once. - For example, a mask of 0xF would set ports 0-3 as outputs and the rest as inputs.
4. **Mode Selection:** - The user can choose between “display” mode to continuously show port statuses or “write” mode to set outputs. - In “display” mode, the sample refreshes the port status every 200ms until the user presses ENTER. - In “write” mode, the user selects a specific output port and provides a mask and value to set the outputs.

timer

Goal

Initialise, start, and read hardware timers in **polling** or **interrupt** mode.

This sample demonstrates how to configure and operate hardware timers available on ADDI-DATA cards, using both continuous polling and event-driven (interrupt) approaches.

Concept and Purpose

The *timer* sample is designed to validate the **timer subsystem** of ADDI-DATA cards. It shows how to configure timebases, reload values, and operational options dynamically, with support for both standard and interrupt-driven operation.

It serves as both a **reference implementation** and a **diagnostic tool** for timing features on cards such as:

- **APCI-1500 / APCIe-1500**
- **APCI-1564 / APCIe-1564**
- **APCIe-1032**

The timer is typically used for periodic events, time measurements, or watchdog triggers.

Execution Workflow

1. **Device detection and selection**

   The program enumerates all installed cards and filters those with timer capability using *AddiDataUtilsSelectDeviceByCapability()* . If multiple cards are detected, the user can select one interactively or specify it through *–card=<name>* .

2. **Mode selection**

   Two operating modes are available:

   - **polling** → the sample periodically reads and prints the timer status.
   - **interrupt** → the program waits for timer expiration events, notified by hardware interrupts.

   This can be set interactively or by passing *–mode=polling* or *–mode=interrupt* .

3. **Timer configuration**

   Once the device and mode are selected, the program interactively requests or parses:

   - **Timer index** (*–timer*): which timer to configure. Most cards have multiple independent timers (e.g. 3 on the APCI-1500).
   - **Timebase** (*–timebase*): defines the unit for reload timing — microseconds (*us*), milliseconds (*ms*), or seconds (*s*).
   - **Reload value** (*–reload*): defines the countdown duration before expiration. The maximum value depends on the card and the timebase.
   - **Timer options** : optional configuration flags such as continuous or single-shot operation, retrieved via *AddiDataUtilsGetTcwOptionsFromUser()* .

4. Initialization and startup

   The timer is configured and started in three steps:

   - *AddiDataInitTimer()* → apply configuration (index, timebase, reload, options)
   - *AddiDataTriggerTimer()* → trigger initial value
   - *AddiDataStartTimer()* → start periodic operation

   When interrupt mode is selected, the program also registers the default callback *AddiDataUtilsInterruptCallback* and enables timer interrupts via:

   - *AddiDataEnableDeviceInterrupts()*
   - *AddiDataEnableTimerInterrupt()*

5. Runtime phase

   - In **polling mode** , the program repeatedly calls *AddiDataReadTimerStatus()* to display: - current value, - status flag, - trigger state.
   - In **interrupt mode** , messages are printed each time the timer interrupt fires, including timestamps and elapsed times.

6. Termination

   When the user presses **ENTER** , the timer is stopped and interrupts are disabled. The program then frees allocated memory and exits cleanly.

Runtime Behavior

Polling mode:

- Displays live timer countdown values every 100 ms.
- Example output:

```
Timer: value=487 | status=1 | triggered=0 | expired=0
Timer: value=238 | status=1 | triggered=0 | expired=0
```

The display refreshes until **ENTER** is pressed.

Interrupt mode:

- The card raises a hardware interrupt each time the timer expires.
- The registered callback prints timing information such as:

```
[INTERRUPT] Timer event received | ID=0 | elapsed=100.03 ms
```

- The user can observe event frequency and verify timing accuracy.

counter

Goal

Initialise, start, and read hardware counters in **polling** or **interrupt** mode.

This sample demonstrates how to configure and operate hardware counters available on ADDI-DATA cards, for tasks such as event counting, frequency measurement, or pulse detection.

Concept and Purpose

The *counter* sample is designed to validate and illustrate the behavior of **hardware counters** on ADDI-DATA cards. It shows how to configure reload values, counting modes, and options, while supporting both continuous polling and interrupt-driven operations.

It serves as a **reference implementation** for developers and testers working on:

- Counting external pulses or encoder signals,
- Measuring frequencies or durations,
- Detecting input edges and generating interrupts on threshold events.

Supported cards include (but are not limited to):

- **APCI-1500 / APCIe-1500**
- **APCI-1564 / APCIe-1564**

Execution Workflow

1. **Device detection and selection**

   The sample detects all installed ADDI-DATA cards and filters those supporting counters using *AddiDataUtilsSelectDeviceByCapability()* .

   The user can then select the desired device interactively or provide a filter via *–card* (e.g. *–card=apcie1564*).

2. Mode selection

   The program offers two operation modes:

   - **polling** → continuously read and print counter values.
   - **interrupt** → wait for counter overflow or event-driven notifications.

   The mode can be selected through *–mode=* or via an interactive menu.

3. Counter configuration

   Once the mode is chosen, the program requests the following parameters:

   - **Counter index** (*–counter*): selects which hardware counter to use. Some cards (e.g. APCI-1500) have multiple counters (typically 3).
   - **Reload value** (*–reload*): defines the threshold after which the counter resets or triggers an interrupt.
   - **Counter options** : retrieved interactively via *AddiDataUtilsGetTcwOptionsFromUser()* , allowing configuration such as counting direction or gate control.

4. Initialization and startup

   The counter is configured and started in three steps:

   - *AddiDataInitCounter()* → apply reload value and mode options.
   - *AddiDataStartCounter()* → enable counting.
   - Optionally, *AddiDataEnableCounterInterrupt()* and *AddiDataSetDeviceInterruptCallback()* if interrupt mode is selected.

   When interrupts are used, the callback *AddiDataUtilsInterruptCallback* prints timestamped events each time a counter interrupt occurs (e.g. overflow or external trigger).

5. Runtime phase

   - In **polling mode** , the program continuously calls *AddiDataReadCounterStatus()* to display: - the current count value, - the counter status flag, - and the trigger state.
   - In **interrupt mode** , the program remains idle until hardware events trigger an interrupt.

6. Termination

   On user input (press **ENTER**), the counter is stopped, interrupts are disabled, and all resources are released before the program exits.

Runtime Behavior

Polling mode:

- The current counter value is printed at regular intervals (100 ms by default).
- Example output:

```
Counter: value=52 | status=1 | triggered=0 | expired=0
Counter: value=103 | status=1 | triggered=0 | expired=0
```

- This mode is useful for verifying real-time increments when an external signal (pulse train) is applied.

Interrupt mode:

- The card raises a hardware interrupt when the counter reaches its reload value or an event condition.
- The callback prints messages such as:

```
[INTERRUPT] Counter event | ID=1 | elapsed=250.12 ms
```

- This helps evaluate signal stability and interrupt timing precision.

watchdog

Goal

Configure and test the hardware watchdog mechanism available on ADDI-DATA cards.

This sample demonstrates how to initialise, start, trigger, and observe watchdog expiration events. It validates both **interrupt handling** and **system safety behavior** using the Addi-Pack API.

Concept and Purpose

The *watchdog* sample is designed to verify that the **hardware watchdog** correctly monitors system activity and generates interrupts when it is not regularly triggered.

A watchdog is a **safety timer** that resets or signals the system if it is not “fed” (triggered) within a defined period. This mechanism is crucial for ensuring system reliability in industrial environments.

This sample shows how to:

- Configure watchdog parameters (timebase, reload value, options),
- Trigger it periodically (software “kick”),
- Detect expiration events via interrupts.

Supported cards include (depending on model and firmware):

- **APCI-1500 / APCIe-1500**
- **APCI-1564 / APCIe-1564**

Execution Workflow

1. **Device detection and selection**

   The sample enumerates all installed devices and filters those with watchdog capability using *AddiDataUtilsSelectDeviceByCapability()* .

   The user can specify a card via *–card=<name>* or select interactively.

2. **Watchdog configuration**

   The user specifies:

   - **Watchdog index** (*–watchdog*): ID of the watchdog to configure.
   - **Timebase** (*–timebase*): *us* , *ms* , or *s* for microsecond, millisecond, or second precision.
   - **Reload value** (*–reload*): defines the duration before expiration (e.g., 500 ms or 10 s).

   Additional options are set through *AddiDataUtilsGetTcwOptionsFromUser()* , allowing configuration of continuous cycle or hardware gate modes.

3. **Initialization phase**

   The watchdog is initialized via *AddiDataInitWatchdog()* . Device and watchdog interrupts are enabled with:

   - *AddiDataEnableDeviceInterrupts()*
   - *AddiDataEnableWatchdogInterrupt()*

   A default callback (*AddiDataUtilsInterruptCallback*) is registered to report expiration events.

   For visual verification, all digital outputs are set high (*AddiDataWriteAllDigitalOutputs(..., 1)*) so that they toggle on watchdog events.

4. **Triggering and monitoring phase**

   The watchdog is started (*AddiDataStartWatchdog()*), then **manually triggered three times** by calling *AddiDataTriggerWatchdog()* with a 1-second delay between each call.

   After the third trigger, the sample waits for the watchdog to expire naturally. When it does, an interrupt message is displayed, confirming proper expiration handling.

5. **Termination**

   Once expiration is detected or the user presses **ENTER** , the program:

   - Stops the watchdog (*AddiDataStopWatchdog()*),
   - Disables interrupts,
   - Resets all outputs,
   - Frees resources and exits cleanly.

Runtime Behavior

Example console output:

```
Using card: apcie-1564
Watchdog will be triggered 3 times...
Triggering Watchdog [1/3]
Triggering Watchdog [2/3]
Triggering Watchdog [3/3]
Waiting for watchdog to expire...
[INTERRUPT] Watchdog event | ID=0 | elapsed=5000.12 ms
Watchdog expired. Press ENTER to exit...
```

The elapsed time shown corresponds to the time between the last trigger and the hardware expiration interrupt.

Card-Specific Notes

Some cards exhibit hardware-dependent behaviors that differ from the generic TCW (Timer/Counter/Watchdog) model. The following notes help avoid unexpected results during testing or integration.

APCI-1564 / APCIe-1564

- A watchdog interrupt will **not** occur if you simply wait for expiration. One of the following must happen **before expiration** :
  - Trigger manually via `AddiDataTriggerWatchdog()` ,
  - **or** set all digital outputs to `1` using `AddiDataWriteAllDigitalOutputs()` .
- Without one of these actions, the watchdog remains **inactive** , meaning no expiration and no interrupt.
- Behavior varies depending on activation method:
  - Manual trigger → typically **one interrupt** on expiration.
  - Setting all outputs high → sometimes **two interrupts** (expiration + internal reset).

Note

These quirks are specific to the 1564 family and may vary slightly depending on firmware version. It is recommended to always perform either a trigger or output write before waiting for expiration on this card.

APCI-1500 / APCIe-1500

- All **three timers share the same timebase** (µs, ms, s). Changing the timebase for one timer **updates it for all timers** .
- To avoid unexpected timing mismatches, always configure **the same timebase for every enabled timer** on this card.
- `reload_value` remains independent per timer — only the **timebase** is global.

Note

This constraint applies to both timers and watchdog timing behavior on APCI-1500 family cards.

APCI-2032 / APCIe-2032

- The watchdog has an attribute “has_expired”, which is set to true when the watchdog reaches 0.

- PCI cards do not generate an interrupt on expiration; instead, the application must poll “has_expired” to detect expiration events.
- APCI-2032 / APCIe-2032 are currently the only ADDI-DATA cards that support digital output interrupts (VCC / CC fault detection).

APCI-2200 / APCIe-2200

- Relays are exposed through the digital-output API; `GetDigitalOutputsStatus` is a real register read-back (2-state: open / closed).
- APCI-2200 (PCI) has no general-purpose timer and does not expose interrupts; poll the watchdog `has_expired` flag. APCIe-2200 (PCIe) has two timers and full DI event-logic / interrupt support.

Interrupt Source Mapping

The meaning of `interrupt_source` bits is *not universal* across cards. This means:

- `interrupt_source = 0x1` **does not necessarily mean the same thing** on APCIx-1500, APCIx-1032 or APCIx-1564.
- Always refer to the **card-specific interrupt handler** and mapping table.

Typical examples by card:

APCI-1500 / APCIe-1500

- `0x01` → Port A interrupt
- `0x02` → Port B interrupt
- `0x04` → Timer 1 interrupt
- `0x08` → Timer 2 interrupt
- `0x10` → Timer 3 interrupt
- `0x20` → Watchdog interrupt
- `0x40` / `0x80` → Voltage / short-circuit faults

APCIe-1032

- `0x01` → DigitalInputEventLogic
- `0x04` → Timer interrupt (PCIe only)

APCI-1564 / APCIe-1564

- `0x01` → Digital Input Event Logic
- `0x02` → Digital Output interrupt
- `0x04` → Timer interrupt (per timer bitmask in `interrupt_mask`)
- `0x08` → Counter interrupt (per counter bitmask in `interrupt_mask`)

APCI-2032 / APCIe-2032

- `0x01` → Digital Output interrupt PCI
- `0x02` → Digital Output interrupt PCIe
- `0x04` → Watchdog interrupt

APCIe-2200

- `0x01` → Digital Input event-logic interrupt
- `0x02` → Timer interrupt

Technical Documentation

C API

C GENERIC

group **Generic**

Generic utilities and helper functions for the C API.

This group includes base types, macros, and helper functions that are shared across modules. Useful for common operations required by multiple modules.

Functions

```
addi_data_code AddiDataGetNumberOfDevices ( uint8_t *
number_of_devices )
```

Get the Number Of Connected Devices.

Retrieves the number of devices currently detected by the system. This function should be called before AddiDataGetDeviceList() .

Parameters :

number_of_devices – **[out]** Pointer to a `uint8_t` receiving the number of devices.

Returns :

ADDIDATA_SUCCESS_CODE on success.

```
addi_data_code AddiDataGetDeviceList ( AddiDataDeviceStruct *
device_list )
```

Fill device_list with a table of strings containing a strucure list of device Infos.

Fills a preallocated array of AddiDataDeviceStruct with information for each detected device. The array referenced by `device_list` must be large enough to store all detected devices. Use AddiDataGetNumberOfDevices() beforehand to determine the required size.

Parameters :

device_list – **[out]** Pointer to a preallocated array of AddiDataDeviceStruct that will receive the information for each detected device.

Returns :

ADDIDATA_SUCCESS_CODE on success.

```
addi_data_code AddiDataGetDeviceStructFromDeviceName ( const char *
device_id , AddiDataDeviceStruct * device )
```

Allow to easily recover the device information structure from the device id.

Looks up a device by its unique identifier and fills the provided AddiDataDeviceStruct with the corresponding information.

Parameters :

- **device_id** – **[in]** ID of the device
- **device** – **[out]** Pointer to a AddiDataDeviceStruct receiving the device information.

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

C COUNTER

group **Counter**

All Counter functions.

Functions to initialize, start, stop, and read hardware counters. Provides accurate counting for timing and events.

Functions

```
addi_data_code AddiDataInitCounter ( const char * device_id ,
uint8_t counter_id , uint32_t reload_value , uint32_t options )
```

Initialize a hardware counter.

Configures a counter with a given reload value and a set of optional configuration flags. The supported options depend on the device model.

Available flags include:

- COUNT_DIRECTION_UP / COUNT_DIRECTION_DOWN (APCIx-1564 only)
- COUNTER_FALLING_EDGE_COUNT / COUNTER_RISING_EDGE_COUNT / COUNTER_BOTH_EDGE_COUNT (APCIx-1564 only)
- SINGLE_CYCLE (APCIx-1500 only)
- EXTERNAL_TRIGGER (APCIx-1500 only)
- HARDWARE_GATE (APCIx-1500 only)

Parameters :

- **device_id** – **[in]** ID of the device
- **counter_id** – **[in]** Which counter to configure
- **reload_value** – **[in]** Reload value applied when reaching zero.
- **options** – **[in]** Bitfield of configuration options (see ADDI_DATA_TCW_OPTIONS).

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataStartCounter ( const char * device_id ,
uint8_t counter_id )
```

Start the counter.

Starts the specified counter on the selected device.

Parameters :

· **device_id** – **[in]** ID of the device

· **counter_id** – **[in]** Counter index to start

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataStopCounter ( const char * device_id ,
uint8_t counter_id )
```

Stop the counter.

Stops the specified counter on the selected device.

Parameters :

· **device_id** – **[in]** ID of the device

· **counter_id** – **[in]** Counter index to stop

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataReadCounterValue ( const char * device_id ,
uint8_t counter_id , uint32_t * read_value )
```

Read the current counter value.

Reads the current value of the specified counter on the selected device.

Parameters :

- **device_id** – **[in]** ID of the device
- **counter_id** – **[in]** Counter to read
- **read_value** – **[out]** Returned counter value

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataReadCounterStatus ( const char * device_id ,
uint8_t counter_id , uint32_t * read_value , uint8_t * is_started ,
uint8_t * software_trigger , uint8_t * has_expired )
```

Read the current counter value and status.

Reads the current value and status of the specified counter on the selected device.

Parameters :

- **device_id** – **[in]** ID of the device
- **counter_id** – **[in]** Counter to read
- **read_value** – **[out]** Returned counter value
- **is_started** – **[out]** 1 if counter running, otherwise 0
- **software_trigger** – **[out]** 1 if counter has been reset through software since last read, otherwise 0
- **has_expired** – **[out]** 1 if the counter has reached zero since the last read, otherwise 0.

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataEnableCounterInterrupt ( const char *
device_id , uint8_t counter_id )
```

Enable counter interrupts.

Enables interrupts for the specified counter on the selected device.

Parameters :

· **device_id** – **[in]** ID of the device

· **counter_id** – **[in]** Which counter to enable

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataDisableCounterInterrupt ( const char *
device_id , uint8_t counter_id )
```

Disable counter interrupts.

Disables interrupts for the specified counter on the selected device.

Parameters :

· **device_id** – **[in]** ID of the device

· **counter_id** – **[in]** Which counter to disable

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

C DIGITAL INPUT/OUTPUT

group **Digital I/O**

Digital input/output control.

Functions to configure and control digital input and output channels. Handles reading and writing digital signals safely.

Functions

```
addi_data_code AddiDataWriteSingleDigitalOutput ( const char *
device_id , uint8_t channel , uint8_t value )
```

Write a value 1 or 0 to the given channel number.

Writes a digital output value to a single channel on the specified device.

Parameters :

- **device_id** – **[in]** ID of the device
- **channel** – **[in]** which channel to write to
- **value** – **[in]** Logical value (0 = off, 1 = on)

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataWriteAllDigitalOutputs ( const char *
device_id , uint32_t channels , uint8_t value )
```

Write a value 1 or 0 on masked channels.

Writes a digital output value to multiple channels on the specified device, as specified by a channel mask.

Parameters :

- **device_id** – **[in]** ID of the device
- **channels** – **[in]** Bitmask selecting channels to write, where each bit represents a channel (1 = write, 0 = ignore)
- **value** – **[in]** Logical value (0 = off, 1 = on)

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataGetDigitalOutputsStatus ( const char *
device_id , uint32_t * output_status )
```

Get the stored Digital Output status i.e. the known state of outputs from the driver. On certain cards, it doesn't strictly get the info from the card. It gets the info from the driver. Under normal circumstances, this is equal to the card outputs.

Parameters :

- **device_id** – **[in]** ID of the device
- **output_status** – **[out]** Pointer receiving the current outputs state (bitmask).

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataReadSingleDigitalInput ( const char *
device_id , uint8_t channel , uint8_t * value )
```

Will fill the first bit of value with the current value of channel.

Reads the digital input value from a single channel on the specified device.

Parameters :

- **device_id** – **[in]** ID of the device
- **channel** – **[in]** channel to read
- **value** – **[out]** Pointer receiving the logical value (0 or 1).

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataReadAllDigitalInputs ( const char *
device_id , uint32_t * channel_values )
```

Will fill channel_values with the value from each channel, one bit per channel.

Reads the digital input values from all channels on the specified device.

Parameters :

- **device_id** – **[in]** ID of the device
- **channel_values** – **[out]** Pointer receiving the input bitmask.

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataSetInterruptEventForInputs ( const char *
device_id , uint32_t channels , DIGITAL_EVENT_TYPE event_type )
```

Set the interrupt behaviour For Inputs, Old values are be kept and not erased, so multiple calls can be made to set each channels.

Parameters :

- **device_id** – **[in]** ID of the device
- **channels** – **[in]** Bitmask selecting input channels to configure.
- **event_type** – **[in]** Event type that triggers an interrupt.

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataSetInterruptEventForOneInput ( const char *
device_id , uint8_t channel , DIGITAL_EVENT_TYPE event_type )
```

Set the interrupt behaviour for a single input. Old should be kept and not erased.

Parameters :

- **device_id** – **[in]** ID of the device
- **channel** – **[in]** channel to set the interrupt
- **event_type** – **[in]** Event type that triggers an interrupt.

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataSetInterruptLogicForPort ( const char *
device_id , uint8_t port_number , DIGITAL_EVENT_LOGIC event_logic )
```

Set the Interrupt Logic For the given port_number. On cards that have one port (i.e. card that are not apcix1500), port_number is ignored.

Parameters :

- **device_id** – **[in]** ID of the device
- **port_number** – **[in]** port number to set the interrupt logic. Card that have only one port will ignore this parameter.
- **event_logic** – **[in]** DIGITAL_EVENT_LOGIC::OR or DIGITAL_EVENT_LOGIC::AND

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataSetOutputMemory ( const char * device_id ,
uint8_t enable )
```

Activating the output memory will save and use the output state when setting outputs. This means that with output memory, setting a given output will not have influence on other outputs. Without output memory, setting a given output will reset to 0 the other outputs. Defaults to ADDI_DATA_ENABLE.

Parameters :

- **device_id** – **[in]** ID of the device
- **enable** – **[in]** ADDI_DATA_ENABLE or ADDI_DATA_DISABLE

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataEnableDigitalInterrupt ( const char *
device_id )
```

Enable Interrupts for Digital functions.

Parameters :

device_id – **[in]** ID of the device

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataDisableDigitalInterrupt ( const char *
device_id )
```

Disable Interrupts for Digital functions.

Parameters :

device_id – **[in]** ID of the device

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataSetDigitalOutputInterrupt ( const char *
device_id , uint8_t enable_vcc , uint8_t enable_cc )
```

Configure digital output interrupt sources (VCC / CC)

Parameters :

- **device_id** – **[in]** ID of the device
- **enable_vcc** – **[in]** Enable VCC interrupt
- **enable_cc** – **[in]** Enable CC interrupt

Returns :

ADDIDATA_SUCCESS_CODE on success

```
addi_data_code AddiDataSetAllDigitalPortsDirection ( const char *
device_id , uint32_t direction_mask )
```

Set the direction of all digital ports at once.

Parameters :

- **device_id** – **[in]** ID of the device
- **direction_mask** – **[in]** A bitmask representing the direction of each port. A 0 bit indicates `PORT_INPUT` , a 1 bit indicates `PORT_OUTPUT` . The least significant bit corresponds to port 0.

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataGetAllDigitalPortDirections ( const char *
device_id , uint32_t * configuration )
```

Get the direction of all digital ports at once.

Parameters :

- **device_id** – **[in]** ID of the device
- **configuration** – **[out]** A bitmask representing the direction of each port. A 1 bit indicates `PORT_OUTPUT` , a 0 bit indicates `PORT_INPUT` . The least significant bit corresponds to port 0.

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataReadAllDigitalInputsFromPort ( const char *
device_id , uint8_t port_number , uint8_t * value )
```

Read all channels of a port and return the values as an integer.

Parameters :

- **device_id** – **[in]** ID of the device
- **port_number** – **[in]** The number of the port to read
- **value** – **[out]** The values of each channel of the port, where a 1 indicates a high value and a 0 indicates a low value

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataWriteSingleDigitalOutputOnPort ( const char *
device_id , uint8_t channel , uint8_t value , uint8_t port_number )
```

Write a single value to a channel of a port.

Parameters :

- **device_id** – **[in]** ID of the device
- **channel** – **[in]** The channel to write (0 to 7 for a port)
- **value** – **[in]** The value to write (0 or 1)
- **port_number** – **[in]** The number of the port

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataWriteAllDigitalOutputsOnPort ( const char *
device_id , uint8_t channels_mask , uint8_t value , uint8_t
port_number )
```

Write all channels of a port. Only channels with a corresponding 1 in the channels_mask will be modified.

Parameters :

- **device_id** – **[in]** ID of the device
- **channels_mask** – **[in]** A bitmask representing the channels to modify (1 to modify, 0 to leave unchanged)
- **value** – **[in]** The value to write to the masked channels (0 or 1)
- **port_number** – **[in]** The number of the port

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataGetDigitalOutputsStatusOnPort ( const char *
device_id , uint8_t port_number , uint8_t * output_status )
```

Get the current output values of a port. Only channels with a corresponding 1 in the channels_mask will be returned.

Parameters :

- **device_id** – **[in]** ID of the device
- **port_number** – **[in]** The number of the port to query
- **output_status** – **[out]** The values of the masked channels, where a 1 indicates a high value and a 0 indicates a low value

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

C INTERRUPT

group **Interrupt**

All Interrupt functions.

Functions to configure, enable, and handle hardware interrupts. Ensures proper callback registration and event handling.

Functions

```
addi_data_code AddiDataSetDeviceInterruptCallback ( const char *
device_id , void ( * interrupt_callback ) ( AddiDataDeviceStruct
interrupted_device , InterruptData data ) )
```

Set the interrupt callback for the device.

Registers a callback function to be invoked when an interrupt occurs on the specified device.

Parameters :

- **device_id** – **[in]** ID of the device
- **interrupt_callback** – **[in]** Callback function to handle interrupts, receiving the interrupted device and interrupt data.

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataEnableDeviceInterrupts ( const char *
device_id )
```

Enable device-level interrupts.

Allow the device to forward interrupt events to the callback mechanism.

Parameters :

device_id – **[in]** ID of the device

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataDisableDeviceInterrupts ( const char *
device_id )
```

Disable device-level interrupts.

Prevents the device from forwarding interrupt events.

Parameters :

device_id – **[in]** ID of the device

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataTestDeviceInterrupts ( const char *
device_id )
```

Tests device interrupts by sending a “false” interrupt.

Sends an internal test interrupt through the driver. Useful to verify callback registration and interrupt handling.

Parameters :

device_id – **[in]** ID of the device

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataClearInterruptFifo ( const char * device_id )
```

Asks the driver to clean its internal interrupt fifo. Under normal circumstances, this function is not used. However, if the interrupt routine always as a “Fifo full” flag set, calling this function might unlock the situation.

Parameters :

device_id – **[in]** ID of the device

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

C TIMER

group **Timer**

All Timer functions.

Functions to initialize, start, stop, and read timers. Useful for precise scheduling and measuring durations.

Functions

```
addi_data_code AddiDataInitTimer ( const char * device_id , uint8_t
timer_id , TIMEBASE_UNITS timebase , uint16_t reload_value ,
uint32_t options )
```

Initialize a timer.

Configures a timer using a timebase (µs/ms/s), a reload value, and optional flags. For frequency-based devices (e.g., APCIx-1500), the API automatically converts the timebase and reload value to the appropriate hardware frequency. In some cards, only 1 TIMEBASE_UNITS should be used for all timer (e.g. PCIX1500 series).

- Available flags include:
- SINGLE_CYCLE (APCIx-1500 only)
- EXTERNAL_TRIGGER (APCIx-1500 only)
- HARDWARE_GATE (APCIx-1500 only)

Parameters :

- **device_id** – **[in]** ID of the device
- **timer_id** – **[in]** Timer index to configure
- **timebase** – **[in]** Timebase unit (0=µs, 1=ms, 2=s, 3=custom)
- **reload_value** – **[in]** Value that will be reloaded when the flag SINGLE_CYCLE is not set
- **options** – **[in]** Bitfield of configuration flags (see ADDI_DATA_TCW_OPTIONS).

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataSetCustomTimerFrequency ( const char *
device_id , uint32_t frequency )
```

for frequency based devices (i.e. apcix1500) set a custom frequency for Timer/ Watchdog Beware that when setting a custom frequency, value from Read timer will be dependent on this frequency. And init timer should be used with TIMEBASE_UNIT::CUSTOM.

Parameters :

- **device_id** – ID of the device
- **frequency** – custom frequency in Hz to set

```
addi_data_code AddiDataStartTimer ( const char * device_id , uint8_t
timer_id )
```

Start the timer.

Parameters :

- **device_id** – **[in]** ID of the device
- **timer_id** – **[in]** which timer to start

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataStopTimer ( const char * device_id , uint8_t
timer_id )
```

Stop the timer.

Parameters :

· **device_id** – **[in]** ID of the device

· **timer_id** – **[in]** which timer to stop

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataReadTimerValue ( const char * device_id ,
uint8_t timer_id , uint32_t * read_value )
```

Read the current timer value Reads the remaining or current counter value of the timer. On frequency-based timer cards, the returned value may differ from the theoretical reload due to internal conversion.

Parameters :

· **device_id** – **[in]** ID of the device

· **timer_id** – **[in]** which timer to read

· **read_value** – **[out]** returned value

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataReadTimerStatus ( const char * device_id ,
uint8_t timer_id , uint32_t * read_value , uint8_t * is_started ,
uint8_t * software_trigger , uint8_t * has_expired )
```

Read the current timer value and status On frequency-based timer cards, the value returned by `readTimerStatus` may not match the expected reload value directly due to conversion from timebase to frequency.

Parameters :

- **device_id** – **[in]** ID of the device
- **timer_id** – **[in]** which timer to read
- **read_value** – **[out]** returned value
- **is_started** – **[out]** 1 if timer is active, 0 otherwise.
- **software_trigger** – **[out]** 1 if timer has been reset through software since last read, 0 otherwise
- **has_expired** – **[out]** 1 if the timer has reached zero since the last read, otherwise 0.

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataTriggerTimer ( const char * device_id ,
uint8_t timer_id )
```

Trigger timer.

Parameters :

- **device_id** – **[in]** ID of the device
- **timer_id** – **[in]** which timer to trigger

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataEnableTimerInterrupt ( const char *
device_id , uint8_t timer_id )
```

Enable timer interrupts.

Parameters :

· **device_id** – **[in]** ID of the device

· **timer_id** – **[in]** which timer to enable

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataDisableTimerInterrupt ( const char *
device_id , uint8_t timer_id )
```

Disable timer interrupts.

Parameters :

· **device_id** – **[in]** ID of the device

· **timer_id** – **[in]** which timer to disable

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

C WATCHDOG

group **Watchdog**

All Watchdog functions.

Functions to initialize, start, stop, and monitor hardware watchdog timers. Helps ensure system reliability by resetting in case of failures.

Functions

```
addi_data_code AddiDataInitWatchdog ( const char * device_id ,
uint8_t watchdog_id , TIMEBASE_UNITS timebase , uint16_t
reload_value , uint32_t options )
```

Initialize a watchdog.

Configures a watchdog using a timebase (µs/ms/s), a reload value, and optional flags. For frequency-based boards (e.g. APCIx-1500), the API automatically converts timebase and reload value to hardware frequency.

Available flags include:

- SINGLE_CYCLE (APCIx-1500 only)
- EXTERNAL_TRIGGER (APCIx-1500 only)
- HARDWARE_GATE (APCIx-1500 only)

Parameters :

- **device_id** – **[in]** ID of the device
- **watchdog_id** – **[in]** Watchdog index to configure.
- **timebase** – **[in]** Timebase unit (0=µs, 1=ms, 2=s, 3=custom)
- **reload_value** – **[in]** Value that will be reloaded when the flag SINGLE_CYCLE is not set
- **options** – **[in]** Bitfield of configuration flags (see ADDI_DATA_TCW_OPTIONS)

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataStartWatchdog ( const char * device_id ,
uint8_t watchdog_id )
```

Start the watchdog.

Parameters :

· **device_id** – **[in]** ID of the device

· **watchdog_id** – **[in]** which watchdog to start

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataStopWatchdog ( const char * device_id ,
uint8_t watchdog_id )
```

Stop the watchdog.

Parameters :

· **device_id** – **[in]** ID of the device

· **watchdog_id** – **[in]** Which watchdog to stop

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataReadWatchdogValue ( const char * device_id ,
uint8_t watchdog_id , uint32_t * value )
```

Read the current watchdog value and status On frequency-based watchdog cards, the value returned by `readWatchdogValue` may not match the expected reload value directly due to conversion from timebase to frequency.

Parameters :

- **device_id** – **[in]** ID of the device
- **watchdog_id** – **[in]** Which watchdog to read
- **value** – **[out]** Returned value

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataReadWatchdogStatus ( const char * device_id ,
uint8_t watchdog_id , uint32_t * read_value , uint8_t * is_started ,
uint8_t * software_trigger , uint8_t * has_expired )
```

Read the current watchdog value On frequency-based watchdog cards, the value returned by `readWatchdogStatus` may not match the expected reload value directly due to conversion from timebase to frequency.

Parameters :

- **device_id** – **[in]** ID of the device
- **watchdog_id** – **[in]** which watchdog to read
- **read_value** – **[out]** returned value
- **is_started** – **[out]** Value is 1 if watchdog is running, 0 otherwise
- **software_trigger** – **[out]** Set to 1 if Watchdog has been reset through software since last read, 0 otherwise
- **has_expired** – **[out]** 1 if the watchdog has reached zero since the last read, otherwise 0.

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataTriggerWatchdog ( const char * device_id ,
uint8_t watchdog_id )
```

Trigger watchdog.

Parameters :

· **device_id** – **[in]** ID of the device

· **watchdog_id** – **[in]** Which watchdog to trigger

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataEnableWatchdogInterrupt ( const char *
device_id , uint8_t watchdog_id )
```

Enable watchdog interrupts.

Parameters :

· **device_id** – **[in]** ID of the device

· **watchdog_id** – **[in]** which watchdog to enable

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

```
addi_data_code AddiDataDisableWatchdogInterrupt ( const char *
device_id , uint8_t watchdog_id )
```

Disable watchdog interrupts.

Parameters :

- **device_id** – **[in]** ID of the device
- **watchdog_id** – **[in]** which watchdog to disable

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

Common API

COMMON GENERIC 101

COMMON INFORMATIONS STRUCTURE 104

COMMON ERROR CODES 111

COMMON UTILITIES 120

COMMON GENERIC

group **Generic**

Generic utilities and helper functions for the Common API.

This group includes base types, macros, and helper functions that are shared across modules. Useful for general operations needed by multiple parts of the library.

Defines

ADDI_DATA_DISABLE

Disable flag (0).

ADDI_DATA_ENABLE

Enable flag (1).

Enums

enum ADDI_DATA_TCW_OPTIONS

Timer/Counter/Watchdog option flags.

This bitmask controls behaviour for timers, counters, and watchdogs. Multiple flags may be combined using bitwise OR.

Available flags:

- COUNT_DIRECTION_UP / COUNT_DIRECTION_DOWN (APCIx-1564/APCIe-1532 only)
- COUNTER_FALLING_EDGE_COUNT / COUNTER_RISING_EDGE_COUNT / COUNTER_BOTH_EDGE_COUNT (APCIx-1564 only)
- SINGLE_CYCLE (APCIx-1500 only)
- EXTERNAL_TRIGGER (APCIx-1500 only)
- HARDWARE_GATE (APCIx-1500 only)

Values:

enumerator COUNT_DIRECTION_DOWN
enumerator COUNT_DIRECTION_UP
enumerator SINGLE_CYCLE
enumerator EXTERNAL_TRIGGER
enumerator HARDWARE_GATE
enumerator COUNTER_FALLING_EDGE_COUNT
enumerator COUNTER_RISING_EDGE_COUNT
enumerator COUNTER_BOTH_EDGE_COUNT

enum ADDI_DATA_DEVICE_TYPE

Indicates the hardware type of the device.

Values:

enumerator PCI

Legacy PCI device

enumerator PCIE

PCI Express device

enumerator MSXE

MSX-series device

enum DIGIO_COUNTER_TIMER_SOURCE

Digital I/O source for counter/timer operations.

Values:

enumerator SIGNAL_INPUT

enumerator TRIGGER_INPUT

enumerator GATE_INPUT

enum PORT_DIRECTION

Digital I/O Port Direction (Input or Output).

Values:

enumerator PORT_INPUT

enumerator PORT_OUTPUT

enum TIMEBASE_UNITS

Timebase units for timers and watchdogs.

Values:

enumerator MICROSECOND

µs

enumerator MILLISECOND

ms

enumerator SECOND

s

enumerator CUSTOM

Custom frequency set by user

COMMON INFORMATIONS STRUCTURE

group **Info Structure**

Structure objects which contain information about devices and modules.

This group includes information structures that are shared across modules.

Enums

```
enum DIGITAL_EVENT_TYPE
```

Supported event types for digital input interrupts.

These values represent the trigger condition used when configuring digital input interrupts.

Some values are **device-specific** :

- `ON_ZERO` and `ON_ONE` are only supported by **APCIx-1500** devices.

Values:

enumerator UNUSED

No event configured

enumerator ON_ZERO

Input should be 0 when the trigger input comes (only relevant in 1500 “AND” Logic)

enumerator ON_ONE

Input should be 1 when the trigger input comes (only relevant in 1500 “AND” Logic)

enumerator FALLING_EDGE

Trigger on falling edge

enumerator RISING_EDGE

Trigger on rising edge

enumerator ANY_EDGE

Trigger on both edges

enum DIGITAL_EVENT_LOGIC

Logic mode used when combining multiple digital input events.

Defines how multiple input lines within the same port contribute to generating an interrupt.

Values:

enumerator OR

Interrupt triggers if at least one bit in the event mask produces the configured event

enumerator AND

Interrupt triggers only if all bits in the event mask produce the configured event simultaneously.

struct TimerInformations

#include <addidata_types.h>

Defines timer capabilities for a given device.

Public Members

`uint8_t number_timers`

Number of available timers

`uint16_t max_reload_value [ NUMBER_OF_TIMEBASES ]`

Maximum reload values for each timebase

`uint32_t valid_options`

Bitmask representing valid timer options

`uint16_t min_reload_value [ NUMBER_OF_TIMEBASES ]`

Minimum reload values for each timebase

struct WatchdogInformations

#include <addidata_types.h>

Defines watchdog capabilities for a given device.

Public Members

`uint8_t number_watchdogs`

Number of available watchdogs

`uint16_t max_reload_value [ NUMBER_OF_TIMEBASES ]`

Maximum reload values for each timebase

`uint32_t valid_options`

Bitmask representing valid watchdog options

`uint16_t min_reload_value [ NUMBER_OF_TIMEBASES ]`

Minimum reload values for each timebase

struct CounterInformations

#include <addidata_types.h>

Counter subsystem capabilities.

Public Members

`uint8_t number_counters`

Number of available counters

`uint32_t max_reload_value`

Maximum reload value supported by each counter

`uint32_t valid_options`

Bitmask representing valid counter options

struct AddiDataFunctionInformations

#include <addidata_types.h>

Functional capability overview of a device.

Aggregates all functional capability information of a device. This structure groups the available features for:

- digital input/output,
- timers,
- watchdogs,
- counters.

Each sub-structure provides detailed information such as number of units, valid configuration options, and value ranges.

Public Members

`DigitalInformations digital`

Digital I/O capabilities of the device

`TimerInformations timer`

Timer subsystem information

`WatchdogInformations watchdog`

Watchdog subsystem information

`CounterInformations counter`

Counter subsystem information

struct AddiDataPCIBusInformations

#include <addidata_types.h>

Low-level PCI/PCIe bus information.

Provides low-level PCI/PCIe bus information for the device. This includes identifiers, bus location, BAR mappings, and interrupt routing information.

The structure is populated during device discovery and allows applications or diagnostic tools to inspect how the card is mapped in the system.

Public Members

`uint16_t vendor_id`

PCI vendor ID (e.g. 0x15B8 for ADDI-DATA)

`uint16_t device_id`

PCI device ID identifying the board model

`uint32_t number`

Internal device index within Addi-Pack

`uint32_t device_number`

Device number on the PCI bus

`uint32_t bus_number`

PCI bus number where the board is located

`BAR_TYPE base_address_register [ ADDI_MAX_BAR ]`

BAR (Base Address Register) mappings

`uint8_t interrupt_number`

Interrupt line or vector used by the device

`uint8_t is_pcie`

1 if the device is PCIe, 0 if legacy PCI

struct AddiDataGeneralInformations

#include <addidata_types.h>

User-visible high-level information about a device.

Contains high-level identification data returned during device discovery. This structure provides static information (model, serial number), software information (driver/library versions), and runtime identifiers such as the device path or custom user-defined ID.

Public Members

`char serial_number [ ADDI_STRING_SIZE ]`

Hardware serial number of the device

`char product_name [ ADDI_STRING_SIZE ]`

Product model name (e.g. “apcie1500”)

`char firmware_version [ ADDI_STRING_SIZE ]`

On-board firmware revision

`char library_version [ ADDI_STRING_SIZE ]`

Addi-Pack API library version

`char driver_version [ ADDI_STRING_SIZE ]`

Installed driver version

`char device_path [ ADDI_STRING_SIZE ]`

System path used to access the device (OS-dependent)

`wchar_t custom_id [ ADDI_STRING_SIZE ]`

Optional user-defined identifier (Windows only, filled by the windows device manager layer)

`ADDI_DATA_DEVICE_TYPE device_type`

Device type enumeration (PCI, PCIe, CPCI, etc.)

struct AddiDataDeviceStruct

#include <addidata_types.h>

Main structure returned by device discovery.

This structure aggregates all relevant information about an ADDI-DATA board. It is filled by the device discovery process and used throughout the API to identify and interact with a specific hardware device.

The structure groups:

- A unique device identifier (`id`)
- General identification and versioning information (`general`)
- Functional capabilities (digital I/O, timers, counters, watchdogs) (`functions`)
- PCI/PCIe bus details (`bus`)

Every API function that targets a device expects an instance of `AddiDataDeviceStruct` . Applications should typically retrieve it using `AddiDataGetDeviceList()` or the device-selection utilities provided in the samples.

Public Members

> `char id [ ADDI_STRING_SIZE ]`
>
> Unique identifier assigned to the device instance

> `AddiDataGeneralInformations general`
>
> High-level information: model, versions, serial, etc.

> `AddiDataFunctionInformations functions`
>
> Functional capabilities and limits

> `AddiDataPCIBusInformations bus`
>
> PCI/PCIe configuration and hardware bus details

struct DigitalInformations

#include <digital_io.h>

Digital I/O capability information for a device.

This structure provides the capabilities of the digital I/O subsystem for a device, including:

- number of available inputs and outputs,
- number of ports,
- which input lines support interrupt events.

Public Members

`uint8_t number_inputs`
: Number of digital input channels

`uint8_t number_outputs`
: Number of digital output channels

`uint8_t number_of_ports`
: Number of digital I/O ports

`uint32_t interrupt_mask`
: Mask of input lines supporting interrupt events

struct InterruptData

#include <interrupt.h>

Information provided by the driver when an interrupt occurs.

Public Members

`uint32_t interrupt_source`
: Bitfield indicating the interrupt source

`uint32_t interrupt_mask`
: Mask of channels/bits involved in the interrupt

`uint32_t arg_count`
: Number of extra arguments available

`uint32_t args [ ADDI_INTERRUPT_MAX_ARG_SIZE ]`
: Array of device-specific additional data

COMMON ERROR CODES

group **Error Codes**

Error codes and definitions for the Common API.

This group includes error codes and definitions that are shared across modules.

Defines

ADDIDATA_FUNCTION_SHIFT

32-bit status/error mask.

The bit layout is structured as follows:

Bits	Field	Size	Description
31	API/DRV	1 bit	0 = Driver, 1 = API
30-24	Domain	7 bits	Functional domain ID
23-16	Reserved	8 bits	Reserved for future use
15-8	Function	8 bits	Function identifier
7-0	Reason	8 bits	Error/reason code

ADDIDATA_CODE (source , domain , func , reason)

Create a complete AddiPack error code from its fields.

Parameters :
- **source** – 0 = Driver, 1 = API
- **domain** – Domain ID
- **func** – Function ID
- **reason** – Reason code

Returns : Encoded 32-bit error code.

Enums

`enum AddiDataSource`

Source of the error.

Identifies whether the error originated from the driver or the API.

Table of values

Name	Value	Description
ADDIDATA_DRIVER	0	Error from driver
ADDIDATA_API	1	Error from API

`enum AddiDataDomain`

Domains for error classification.

Each domain represents a subsystem or functional module within Addi-Pack.

Table of values

Name	Value	Description
ADDIDATA_DOMAIN_GENERIC	0	General / unspecified

Name	Value	Description
ADDIDATA_DOMAIN_COUNTER	1	Counter subsystem
ADDIDATA_DOMAIN_DIGIO	2	Digital I/O subsystem
ADDIDATA_DOMAIN_TIMER	3	Timer subsystem
ADDIDATA_DOMAIN_WATCHDOG	4	Watchdog subsystem
ADDIDATA_DOMAIN_INTERRUPTS	5	Interrupt handling
ADDIDATA_DOMAIN_LINUX_IOCTL	6	Linux IOCTL layer
ADDIDATA_DOMAIN_WIN_IOCTL	7	Windows IOCTL layer
ADDIDATA_DOMAIN_DIG_EVENT	8	Digital event
ADDIDATA_DOMAIN_PLUGIN	9	Plugin domain

`enum AddiDataFunction`

Functions for error classification.

Identifies which operation or API function generated the error.

Table of values

Name	Value	Description
ADDIDATA_FUNCTION_NONE	0	No function defined
ADDIDATA_FUNCTION_READ	1	Read operation
ADDIDATA_FUNCTION_INTERRUPT_CALLBACK	2	Interrupt callback function
ADDIDATA_FUNCTION_IOCTL_CALL	3	IOCTL call
ADDIDATA_FUNCTION_IOCTL_INIT	4	IOCTL initialization

Name	Value	Description
ADDIDATA_FUNCTION_ENABLE_INTERRUPT	5	Enable interrupt
ADDIDATA_FUNCTION_CREATE_INTERRUPT	6	Create interrupt
ADDIDATA_FUNCTION_KILL_INTERRUPT	7	Kill interrupt
ADDIDATA_FUNCTION_EVENT_FD	8	Event file descriptor
ADDIDATA_FUNCTION_WAIT_INTERRUPT	9	Wait for interrupt
ADDIDATA_FUNCTION_SELECT_EVENT	10	Select event
ADDIDATA_FUNCTION_CHECK_RELOAD_VALUE	11	Check reload value
ADDIDATA_FUNCTION_CHECK_TIMEBASE	12	Check timebase
ADDIDATA_FUNCTION_SET_DEVICE_HANDLE	13	Set device handle
ADDIDATA_FUNCTION_CALL	14	Generic function call
ADDIDATA_FUNCTION_HANDLE_RETURN	15	Handle return value
ADDIDATA_FUNCTION_CHANNEL_RANGE	16	Channel range selection
ADDIDATA_FUNCTION_INFO	17	Information retrieval
ADDIDATA_FUNCTION_INIT	18	Initialization function
ADDIDATA_FUNCTION_START	19	Start operation
ADDIDATA_FUNCTION_TRIGGER	20	Trigger function
ADDIDATA_FUNCTION_CONVERT_FREQ	21	Convert frequency
ADDIDATA_FUNCTION_STOP	22	Stop operation
ADDIDATA_FUNCTION_READ_VALUE	23	Read value

Name	Value	Description
ADDIDATA_FUNCTION_READ_STATUS	24	Read status
ADDIDATA_FUNCTION_WRITE_DIGOUT	25	Write to digital output
ADDIDATA_FUNCTION_INTERRUPT_EVENT	26	Interrupt event
ADDIDATA_FUNCTION_INTERRUPT_LOGIC	27	Interrupt logic
ADDIDATA_FUNCTION_UPDATE_DIGITAL_INTERRUPT	28	Update digital interrupt
ADDIDATA_FUNCTION_GET_DIGOUT_STATUS	29	Get digital output status
ADDIDATA_FUNCTION_ENSURE_ID	30	Ensure device ID
ADDIDATA_FUNCTION_TEST	31	Test function
ADDIDATA_FUNCTION_ZILOG_INIT_EVENT	32	Zilog initialization event
ADDIDATA_FUNCTION_INIT_DIGIT_INTERRUPT	33	Initialize digital interrupt
ADDIDATA_FUNCTION_START_STOP_TCW	34	Start/stop TCW
ADDIDATA_FUNCTION_SET_OUTPUT_MEMORY	35	Set output memory
ADDIDATA_FUNCTION_INIT_EVENT_LOGIC	36	Initialize event logic
ADDIDATA_FUNCTION_INIT_TCW_COMPATIBILITY	37	Initialize TCW compatibility
ADDIDATA_FUNCTION_API_GET_LAST_ERRROR	38	Retrieve last API error
ADDIDATA_FUNCTION_DISABLE_INTERRUPT	39	Disable interrupt
ADDIDATA_FUNCTION_GET_DEVICES	40	Get devices
ADDIDATA_FUNCTION_GET_STRING	41	Get string

Name	Value	Description
ADDIDATA_FUNCTION_GET_INSTANCE	42	Get instance
ADDIDATA_FUNCTION_DIRECT_ACCESS	43	Direct access
ADDIDATA_FUNCTION_OUTPUT_INTERRUPT	44	Set Output interrupt
ADDIDATA_FUNCTION_SET_PORT_DIRECTION	45	Set digital port direction
ADDIDATA_FUNCTION_GET_PORT_DIRECTION	46	Get port direction
ADDIDATA_FUNCTION_PORT_RANGE	47	Port range selection

`enum AddiDataReason`

Reasons for error classification.

Provides the specific reason associated with an error code.

Table of values

Name	Value	Description
ADDIDATA_REASON_NONE	0	No error reason defined
ADDIDATA_REASON_NOT_INITIALIZED	1	The module or device was not initialized
ADDIDATA_REASON_ALREADY_INITIALIZED	2	The module or device was already initialized
ADDIDATA_REASON_INVALID_PARAMETER	3	One or more input parameters were invalid
ADDIDATA_REASON_OUT_OF_BOUNDS	4	A value or index was out of valid range
ADDIDATA_REASON_NOT_SUPPORTED	5	The requested feature is not supported

Name	Value	Description
ADDIDATA_REASON_OVERFLOW	6	A buffer or counter overflow occurred
ADDIDATA_REASON_UNDERFLOW	7	A buffer or counter underflow occurred
ADDIDATA_REASON_HARDWARE_FAILURE	8	A hardware failure was detected
ADDIDATA_REASON_INVALID_STATE	9	The operation was not valid in the current state
ADDIDATA_REASON_CONFIG_ERROR	10	Configuration error detected
ADDIDATA_REASON_INTERNAL_ERROR	11	Internal or unexpected error
ADDIDATA_REASON_UNKNOWN	12	Unknown or unspecified error
ADDIDATA_REASON_INFO_UNAVAILABLE	13	Requested information is unavailable
ADDIDATA_REASON_INIT_FAILURE	14	Initialization failed
ADDIDATA_REASON_INVALID_RETURN_VAL	15	Invalid return value encountered
ADDIDATA_REASON_INVALID_ID	16	Invalid identifier or handle
ADDIDATA_REASON_INVALID_OPTION	17	Invalid or unsupported option specified
ADDIDATA_REASON_INVALID_TIMEBASE	18	Invalid timebase specified
ADDIDATA_REASON_HANDLE_ERROR	19	Invalid handle

Functions

```
size_t AddiDataGetLastError ( char * error_message , size_t
max_len )
```

Function which returns the pretty messsage from AddiPackException .

This function extracts the internally stored error message generated by the most recent AddiPackException .

Parameters :

- **error_message** – **[out]** Pointer to a buffer where the error message will be stored.
- **max_len** – **[in]** Maximum length of the buffer in bytes.

Returns :

Number of characters copied (same as `strlen(error_message)`).

```
addi_data_code AddiDataPrintLastError ( void )
```

Prints the last error message with the AddiDataGetLastError function. This function is a helper function. Can be used for quick checking.

Returns :

addi_data_code ADDIDATA_SUCCESS_CODE

```
addi_data_code AddiDataTestError ( const char * device_id )
```

Test function that generates a predefined error for testing purposes . Will always “fail”.

Parameters :

device_id – **[in]** The ID of the device to test (can be any valid device ID).

Returns :

addi_data_code An error code that can be used to test error handling.

COMMON UTILITIES

group **Common Utilities**

Utilities and helper functions for the Common API.

This group includes utility functions and helper methods that are shared across modules. Useful for samples for instance.

Arg Parser

group **Arg Parser**

Argument parser utilities for the Common API.

This group includes argument parsing functions.

Functions

```
void AddiDataArgParserAddOption ( const char * name , char
short_option , const char * description , int has_value )
```

Add a new command-line option.

An option may require a value (e.g. `--config file.txt`) or be value-less depending on `has_value` .

Parameters :

- **name** – **[in]** Long name (e.g., “config”)
- **short_option** – **[in]** Short option (e.g., ‘c’)
- **description** – **[in]** Description for help output
- **has_value** – **[in]** Does the option requires a value (1: yes, 0: no)

```
void AddiDataArgParserAddFlag ( const char * name , char
short_option , const char * description )
```

Add a new boolean flag.

Example:

- `--verbose` or `-v`
- Does not take a value (true if present).

Parameters :

- **name** – **[in]** Long name (e.g., “verbose”)
- **short_option** – **[in]** Short option (e.g., ‘v’)
- **description** – **[in]** Description for help output

```
void AddiDataArgParserParse ( int argc , char * argv [ ] )
```

Parse command-line arguments.

Must be called after registering all flags and options.

Parameters :

- **argc** – **[in]** Argument count
- **argv** – **[in]** Argument vector

```
int AddiDataArgParserIsFlagSet ( const char * name )
```

Check if a flag was set;.

Parameters :

name – **[in]** Long name of the flag

Returns :

1 is set, 0 otherwise

```
int AddiDataArgParserIsOptionSet ( const char * name )
```

Check if an option was set.

Parameters :

name – **[in]** Long name of the option

Returns :

1 if set, 0 otherwise

```
int AddiDataArgParserGetOptionValue ( const char * name , char *
value , size_t max_size )
```

Retrieve the value of an option (dynamically allocated).

Parameters :

- **name** – **[in]** Longname of the option (e.g., “config”)
- **value** – **[out]** Buffer to write the value into (must be allocated by caller)
- **max_size** – **[in]** Maximum number of bytes to write (including null terminator ‘\0’)

Returns :

1 if a value was found and copied, 0 otherwise

```
void AddiDataArgParserClear ( )
```

Clear the internal parser state.

Removes all options, flags and stored values. Useful when running multiple independent parses.

User I/O

group **User I/O**

Interactive utilities for AddiPack sample applications.

This module provides helper functions used by the interactive C samples (timer, counter, watchdog, digital I/O).

Enums

```
enum RELAY_STATE
```

Possible states of a relay.

Values:

enumerator OPEN

The relay is open (non-conducting).

enumerator CLOSE

The relay is closed (conducting).

enumerator OPEN

The relay is open (non-conducting).

enumerator CLOSE

The relay is closed (conducting).

enum class RELAY_STATE : std :: uint8_t

Possible states of a relay.

Values:

enumerator OPEN

The relay is open (non-conducting).

enumerator CLOSE

The relay is closed (conducting).

Functions

```
void AddiDataUtilsPrintAvailableTcwOptions ( unsigned int
valid_mask )
```

Print the list of available TCW (Timer-Counter-Watchdog) options for a given device.

This function filters all TCW options against the provided and displays only the options supported by the device. Each option is numbered sequentially starting at 1 for user selection.

Parameters :

valid_mask – **[in]** Bitmask of valid TCW options supported by the device

```
unsigned int AddiDataUtilsGetTcwOptionsFromUser ( unsigned int
valid_mask )
```

Prompt the user to select TCW (Timer-Counter-Watchdog) options interactively.

This function repeatedly prompts the user to select options from the list of valid TCW options (as previously displayed by AddiDataUtilsPrintAvailableOptions). The function updates the provided mask by applying the selected options. The loop ends when the user enters 0.

Parameters :

valid_mask – **[in]** Bitmask of valid TCW options for the current device

Returns :

Updated mask containing all options selected by the user

```
int AddiDataUtilsPromptString ( const char * message , const char *
* accepted , int accepted_count , char * output , size_t
output_size )
```

Prompt the user to input a string with optional accepted values and default.

Parameters :

- **message** – **[in]** Prompt message to display
- **accepted** – **[in]** List of accepted values (NULL if none)
- **accepted_count** – **[in]** Number of accepted values
- **output** – **[out]** Output buffer
- **output_size** – **[in]** Size of output buffer

Returns :

1 if value was entered or default used, 0 if failed

```
int AddiDataUtilsPromptInt ( const char * message , int min , int
max )
```

Prompt the user to input an integer between [min, max] with optional default.

Parameters :

- **message** – **[in]** Prompt message to display
- **min** – **[in]** Minimum allowed value
- **max** – **[in]** Maximum allowed value

Returns :

the selected integer value

```
unsigned int AddiDataUtilsPromptHex ( const char * message ,
unsigned int min , unsigned int max )
```

Prompt the user to input a hexadecimal value between [min, max].

Parameters :

- **message** – **[in]** Prompt message to display
- **min** – **[in]** Minimum allowed value
- **max** – **[in]** Maximum allowed value

Returns :

the selected hexadecimal value

```
int AddiDataUtilsGetValidString ( const char * name , const char *
message , const char * * accepted , int accepted_count , char *
output , size_t output_size )
```

Get a validate string from CLI args or fallback to prompt if invalid.

If the CLI `--name` option is present and valid, its value is returned. Otherwise the user is prompted until a valid value is entered.

Parameters :

- **name** – **[in]** Name of the CLI argument (e.g. “mode”)
- **message** – **[in]** Prompt message
- **accepted** – **[in]** List of accepted values (NULL if none)
- **accepted_count** – **[in]** Number of accepted values
- **output** – **[out]** Output buffer
- **output_size** – **[in]** Size of output buffer

Returns :

1 if value is valid and copied to output, 0 otherwise

```
int AddiDataUtilsGetValidInt ( const char * name , const char *
message , int min , int max )
```

Get a validated integer from CLI args or prompt fallback.

Parameters :

· **name** – **[in]** Name of the CLI argument (e.g. “timer”)

· **message** – **[in]** Prompt message

· **min** – **[in]** Minimum allowed value

· **max** – **[in]** Maximum allowed value

Returns :

Validated integer

```
unsigned int AddiDataUtilsGetValidHex ( const char * name , const
char * message , unsigned int min , unsigned int max )
```

Get a validated hexadecimal value from CLI args or prompt fallback.

Parameters :

· **name** – **[in]** Name of the CLI argument (e.g. “mask”)

· **message** – **[in]** Prompt message

· **min** – **[in]** Minimum allowed value

· **max** – **[in]** Maximum allowed value

Returns :

Validated hexadecimal value

```
void AddiDataUtilsExitOnUserInput ( int exit_code )
```

Waits for the user to press ENTER, then exits the program.

Parameters :

exit_code – **[in]** the code to return. 'EXIT_FAILURE' or 'EXIT_SUCCESS'.

```
void AddiDataUtilsDisplayAllPortsConfigurations ( uint16_t
port_mask , uint8_t * input_values , uint8_t * output_value ,
uint8_t max_port )
```

Get the configuration (input/output) of all digital ports on the device.

This function fills the provided output arrays with the configuration of each port specified in the port_mask.

Parameters :

- **port_mask** – **[in]** Bitmask related to the current port configuration.
- **input_values** – **[in]** Current Input value of the ports (from 0x00 to 0xFF for each port).
- **output_value** – **[in]** Current Output value of the ports (from 0x00 to 0xFF for each port).
- **max_port** – **[in]** Maximum number of ports to check (size of input_values and output_value arrays)

```
int AddiDataUtilsSetRelayState ( const char * device_id , uint8_t
channel , RELAY_STATE state )
```

Set the state of a relay.

Controls the physical state of a relay channel on the specified device.

- Use RELAY_OPEN to open the relay (non-conducting).
- Use RELAY_CLOSE to close the relay (conducting).

Parameters :

- **device_id** – **[in]** ID of the device
- **channel** – **[in]** Relay channel number (0-indexed)
- **state** – **[in]** Desired relay state (RELAY_STATE)

```
int AddiDataUtilsGetRelayState ( const char * device_id , uint8_t
channel , RELAY_STATE * state )
```

Get the state of a relay.

Reads the physical state of a relay channel from the specified device.

- Returns RELAY_OPEN if the relay is open (non-conducting).
- Returns RELAY_CLOSE if the relay is closed (conducting).

Parameters :

- **device_id** – **[in]** ID of the device
- **state** – **[out]** Pointer to store the relay state (RELAY_STATE)

Returns :

1 if relay is closed, 0 if open

```
void setRelayState ( std :: shared_ptr < Device > device , std ::
uint8_t channel , RELAY_STATE state )
```

Set the state of a relay.

```
RELAY_STATE getRelayState ( std :: shared_ptr < Device > device ,
std :: uint8_t channel )
```

Get the state of a relay.

Device

group **Device Utils**

Device utilities for the CPP API.

Device related utilities and helper functions.

Functions

```
int AddiDataUtilsMatchesCardFilter ( const AddiDataDeviceStruct *
device , const char * card_filter )
```

Check whether a device matches a card filter string.

Parameters :

- **device** – **[in]** Pointer to device struct
- **card_filter** – **[in]** Filter string (nullable or empty to match all)

Returns :

1 if matches, 0 otherwise

```
const AddiDataDeviceStruct * AddiDataUtilsSelectDeviceByCapability (
const AddiDataDeviceStruct * devices , int number_of_devices , const
char * name , int ( * predicate ) ( const AddiDataDeviceStruct * ) ,
const char * label )
```

Select a device matching a predicate from a list of devices.

If CLI `--name` is provided and matches exactly, that device is returned. Otherwise, the user is prompted to pick from all devices satisfying `predicate` .

Parameters :

- **devices** – **[in]** Array of devices
- **number_of_devices** – **[in]** Number of devices in `devices`
- **name** – **[in]** CLI argument name (e.g. “card”)
- **predicate** – **[in]** Predicate used to filter eligible devices
- **label** – **[in]** Label to display (e.g. “timer”)

Returns :

Pointer to the selected device, or NULL on failure

```
int AddiDataUtilsSelectMode ( const char * name , const char * modes
[ ] , const char * descriptions [ ] , int count , char * output ,
size_t output_size )
```

Select a string mode from a list, using CLI `--<name>` or prompt.

Parameters :

- **name** – **[in]** CLI option name (e.g. “mode”)
- **modes** – **[in]** List of available mode strings
- **descriptions** – **[in]** Optional descriptions for each mode (NULL if none)
- **count** – **[in]** Number of modes
- **output** – **[out]** Output buffer
- **output_size** – **[in]** Size of output buffer

Returns :

1 if a valid selection was made and copied to output,
0 on failure

Timer Interruptions

group **Timer Interruptions Utils**

Timer interruption utilities for the CPP API.

Timer interruption related utilities and helper functions.

Functions

```
TIMEBASE_UNITS AddiDataUtilsParseTimebase ( const char * timebase )
```

Parse a timebase string (“us”, “ms”, “s”) into a TIMEBASE_UNITS enum value.

Parameters :

timebase – **[in]** String value (“us”, “ms” or “s”)

Returns :

Parsed TIMEBASE_UNITS enum value

```
void AddiDataUtilsInterruptCallback ( AddiDataDeviceStruct device ,
InterruptData data )
```

Default interrupt callback function. Logs the source, mask and elapsed time.

- Logs:
  - interrupt source
  - interrupt mask
  - instantaneous latency since last interrupt

Parameters :

- **device** – **[in]** Device that triggered the interrupt
- **data** – **[in]** Interrupt data structure

```
void AddiDataUtilsResetInterruptFlag ( )
```

Reset the interrupt flag and internal timer.

```
int AddiDataUtilsHasInterruptHappened ( )
```

Check whether an interrupt has occurred since the last reset.

Returns :

1 if interrupt happened, 0 otherwise

```
int AddiDataUtilsGetValidTimebaseList ( const AddiDataDeviceStruct *
device , const char * * timebases )
```

Get a list of valid timebases supported by the device.

This function returns only the timebases for which a max reload value exists.

Parameters :

- **device** – **[in]** Pointer to the selected device
- **timebases** – **[out]** Output array of timebase strings (“us”,”ms,””s”,”c”)

Returns :

Number of valid timebases written to `timebases`

Windows-Linux Compatibility

group **Windows-Linux compatility Utils**

Compatibility utilities.

Windows and Linux compatibility related utilities and helper functions.

Functions

```
static inline int _kbhit ( )
```

This is not strictly an equivalent, as most linux terminal need “enter” to be pressed to relay input buffer to an application.

Returns :

0 if no input is available 1 if one is available.

C++ API

C++ DEVICE INTERFACE

group **Device Interfaces**

Device interface definitions for the CPP API.

This group includes device interface definitions that are shared across modules.

Functions

```
virtual void setAllDigitalPortsDirections ( uint32_t
direction_mask ) = 0
```

Configure the direction of all digital ports at once.

Parameters :

direction_mask – **[in]** A bitmask representing the direction of each port. A 0 bit indicates `PORT_INPUT` , a 1 bit indicates `PORT_OUTPUT` . The least significant bit corresponds to port 0.

Returns :

ADDIDATA_SUCCESS_CODE on success. Otherwise, see Common Error Codes .

class CounterInterface

#include <counter_interface.hpp>

Interface for counter operations.

This class provides an interface for managing device counters

Public Functions

```
virtual void initCounter ( uint8_t counter_id , uint32_t
reload_value , uint32_t options ) = 0
```

CounterInterface constructor.

```
virtual void startCounter ( uint8_t counter_id ) = 0
```

Start the counter.

```
virtual void stopCounter ( uint8_t counter_id ) = 0
```

Stop the counter.

```
virtual uint32_t readCounterValue ( uint8_t counter_id ) = 0
```

Read the current value of the counter.

```
virtual void readCounterStatus ( uint8_t counter_id , uint32_t &
read_value , uint8_t & is_started , uint8_t & software_trigger ,
uint8_t & has_expired ) = 0
```

Read the status of the counter.

```
virtual void enableCounterInterrupt ( uint8_t counter_id ) = 0
```

Enable counter interrupts.

```
virtual void disableCounterInterrupt ( uint8_t counter_id ) = 0
```

Disable counter interrupts.

class Device

#include <device.hpp>

Main device class providing access to all hardware interfaces.

Through this class, users can access all available interfaces, such as timers, counters, digital I/O, watchdogs, and interrupts.

Example usage:

```
auto device =
DeviceDiscoverySingleton::getInstance().getDeviceFromID("device_id");
device->timer()->initTimer(0, TIMEBASE_MS, 1000, 0); //
Access TimerInterface via Device
```

Public Functions

```
Device ( )
```

Device constructor.

```
virtual ~Device ( ) = default
```

Device destructor.

```
virtual AddiDataDeviceStruct getDeviceInformation ( ) = 0
```

Get device information.

`TimerInterface * timer ( )`

Get the Timer Interface TimerInterface .

`CounterInterface * counter ( )`

Get the Counter Interface CounterInterface .

`DigitalIOInterface * digitalIo ( )`

Get the Digital I/O Interface DigitalIOInterface .

`DigitalIOPortsInterface * digitalIoPorts ( )`

Get the Digital I/O Ports Interface DigitalIOPortsInterface .

`WatchdogInterface * watchdog ( )`

Get the Watchdog Interface WatchdogInterface .

`InterruptInterface * interrupt ( )`

Get the Interrupt Interface InterruptInterface .

class DeviceDiscovery

#include <device_discovery.hpp>

Interface for device discovery.

This class provides methods to discover and access connected devices

Public Static Functions

```
static std :: vector < std :: shared_ptr < Device > > getDevices
( )
```

Get a list of all connected devices.

```
static std :: shared_ptr < Device > getDeviceFromId ( const
std :: string & device_id )
```

Get a device from its ID.

class DigitalIOInterface : public DigitalIOCommonInterface

#include <digital_io_interface.hpp>

Interface for digital I/O operations.

This class provides an interface for managing device Digio

Public Functions

```
virtual bool readSingleDigitalInput ( uint8_t channel ) = 0
```

Will read a single digital input at the specified channel.

Parameters :

channel – which channel to read

Returns :

true if the value is 1

Returns :

false if the value is 0

```
virtual uint32_t readAllDigitalInput ( ) = 0
```

Read all 32 channels and return the values as an integer.

Returns :

int the values of each channel

```
virtual void writeSingleDigitalOutput ( uint8_t channel , bool
value ) = 0
```

Will write a single value at the specified channel.

Parameters :

- **channel** –
- **value** – The value to be written, 1 (true) or 0 (false)

```
virtual void writeAllDigitalOutput ( uint32_t channels , bool
value ) = 0
```

Write all channels.

Parameters :

value – to write to masked channels

```
virtual uint32_t getDigitalOutputStatus ( ) = 0
```

Get current output values.

```
virtual void setInterruptEventForInputs ( uint32_t channels ,
DIGITAL_EVENT_TYPE event_type ) = 0
```

Set the interrupt behaviour For Inputs, Old values should be kept and not erased, so multiple calls can be made to set each channels.

Parameters :

- **channels** – Channels mask. A 1 will represent a channel to modify the value.
- **event_type** – type of event that will trigger an interrupt in the masked channels

```
virtual void setInterruptEventForOneInput ( uint8_t channel ,
DIGITAL_EVENT_TYPE event_type ) = 0
```

Set the interrupt behaviour for a single input. Old values should be kept and not erased.

Parameters :

- **channel** – channel to set the interrupt
- **event_type** – type of event that will trigger an interrupt

```
virtual void setInterruptLogicForPort ( uint8_t port_number ,
DIGITAL_EVENT_LOGIC event_logic ) = 0
```

Set the Interrupt Logic For given Port number.

Parameters :

- **port_number** –
- **event_logic** – AND or OR

```
virtual void setOutputInterrupt ( bool enable_vcc , bool
enable_cc ) = 0
```

Configure digital output interrupts.

Parameters :

- **enable_vcc** – Enable VCC interrupt
- **enable_cc** – Enable CC interrupt

```
virtual void enableDigitalInterrupt ( ) = 0
```

Enable digital interrupts.

```
virtual void disableDigitalInterrupt ( ) = 0
```

Disable digital interrupts.

```
virtual void setOutputMemoryOn ( ) = 0
```

Clear digital interrupt status.

```
virtual void setOutputMemoryOff ( ) = 0
```

Clear digital interrupt status.

class DigitalIOPortsInterface : public DigitalIOCommonInterface

#include <digital_io_interface.hpp>

Interface for digital I/O operations using Ports configuration.

This class provides an interface for managing device Digio with Ports configuration, allowing up to 96 channels (e.g., for APCI-1696)

Public Functions

```
virtual uint32_t getAllDigitalPortDirections ( ) = 0
```

Get the direction of all digital ports at once.

Returns :

A 32 bitmask representing the direction of each port. A 0 bit indicates `PORT_INPUT` , a 1 bit indicates `PORT_OUTPUT` . The least significant bit corresponds to port 0.

```
virtual uint8_t readAllDigitalInputsFromPort ( uint8_t
port_number ) = 0
```

Read all channels of a port and return the values as an integer.

Parameters :

port_number – which port to read

Returns :

8 bitmask representing the value of each channel of the port, where a 1 indicates a high value and a 0 indicates a low value

```
virtual void writeSingleDigitalOutputOnPort ( uint8_t channel ,
uint8_t value , uint8_t port_number ) = 0
```

Will write a single value at the specified channel of a port.

Parameters :

- **channel** – which channel to write
- **value** – The value to be written, 1 (true) or 0 (false)
- **port_number** – which port to write

```
virtual void writeAllDigitalOutputsOnPort ( uint8_t
channels_mask , uint8_t value , uint8_t port_number ) = 0
```

Write all channels of a port. Only channels with a corresponding 1 in the channels_mas will be modified.

Parameters :

- **channels_mask** – A 8 bitmask representing the channels to modify, where a 1 indicates a channel to modify and a 0 indicates a channel to keep unchanged
- **value** – The value to write to the masked channels, 1 (true) or 0 (false)

```
virtual uint8_t getDigitalOutputsStatusOnPort ( uint8_t
port_number ) = 0
```

Get current output values of a port. Only channels with a corresponding 1 in the channels_mask will be returned.

Parameters :

port_number – which port to query

Returns :

8 bitmask representing the value of each masked channel of the port, where a 1 indicates a high value and a 0 indicates a low value

```
virtual void enableDigitalInterrupt ( ) = 0
```

Enable digital interrupts.

```
virtual void disableDigitalInterrupt ( ) = 0
```

Disable digital interrupts.

```
virtual void setOutputMemoryOn ( ) = 0
```

Clear digital interrupt status.

```
virtual void setOutputMemoryOff ( ) = 0
```

Clear digital interrupt status.

class InterruptInterface

#include <interrupt_interface.hpp>

Interrupt Interface class.

This class provides an interface for managing device interrupts.

Public Functions

```
~InterruptInterface ( ) = default
```

InterruptInterface constructor.

```
virtual void setDeviceInterruptCallback ( std :: function < void
( const AddiDataDeviceStruct & , const InterruptData & ) >
interrupt_callback ) = 0
```

Set the Device Interrupt Callback.

```
virtual void enableDeviceInterrupts ( ) = 0
```

Enable device interrupts.

```
virtual void disableDeviceInterrupts ( ) noexcept = 0
```

Disable device interrupts.

```
virtual void clearInterruptFifo ( ) = 0
```

Clear the interrupt FIFO.

```
virtual bool isInterruptActive ( ) = 0
```

Check if an interrupt is active.

class TimerInterface

#include <timer_interface.hpp>

Interface for Timer operations.

This class provides methods to for managing device timers

Public Functions

```
virtual ~TimerInterface ( ) = default
```

TimerInterface destructor.

```
virtual void initTimer ( uint8_t timer_id , TIMEBASE_UNITS
timebase , uint16_t reload_value , uint32_t options ) = 0
```

Initialize the timer with specified parameters.

```
virtual void setCustomTimerFrequency ( uint32_t frequency ) = 0
```

Set a custom frequency for cards that support it.

Parameters :

frequency – in Hz to set

```
virtual void startTimer ( uint8_t timer_id ) = 0
```

Start the timer.

```
virtual void stopTimer ( uint8_t timer_id ) = 0
```

Stop the timer.

```
virtual uint32_t readTimerValue ( uint8_t timer_id ) = 0
```

Read the current value of the timer.

```
virtual void readTimerStatus ( uint8_t timer_id , uint32_t &
read_value , uint8_t & is_started , uint8_t & software_trigger ,
uint8_t & has_expired ) = 0
```

Read the status of the timer.

```
virtual void triggerTimer ( uint8_t timer_id ) = 0
```

Trigger the timer via software.

```
virtual void enableTimerInterrupt ( uint8_t timer_id ) = 0
```

Enable timer interrupts.

```
virtual void disableTimerInterrupt ( uint8_t timer_id ) = 0
```

Disable timer interrupts.

class WatchdogInterface

#include <watchdog_interface.hpp>

Interface for Watchdog operations.

This class provides methods to for managing device watchdogs

Public Functions

```
virtual void initWatchdog ( uint8_t watchdog_id , TIMEBASE_UNITS
timebase , uint16_t reload_value , uint32_t options ) = 0
```

Init the watchdog.

```
virtual void startWatchdog ( uint8_t watchdog_id ) = 0
```

Start the watchdog.

```
virtual void stopWatchdog ( uint8_t watchdog_id ) = 0
```

Stop the watchdog.

```
virtual uint32_t readWatchdogValue ( uint8_t watchdog_id ) = 0
```

Read the current value of the watchdog.

```
virtual void readWatchdogStatus ( uint8_t watchdog_id , uint32_t
& read_value , uint8_t & is_started , uint8_t &
software_trigger , uint8_t & has_expired ) = 0
```

Read the status of the watchdog.

```
virtual void triggerWatchdog ( uint8_t watchdog_id ) = 0
```

Trigger the watchdog via software.

```
virtual void enableWatchdogInterrupt ( uint8_t watchdog_id ) = 0
```

Enable watchdog interrupts.

```
virtual void disableWatchdogInterrupt ( uint8_t watchdog_id ) =
0
```

Disable watchdog interrupts.

C++ ERROR CODES

group **Error Codes**

Error codes and definitions for the CPP API.

This group includes error codes and definitions that are shared across modules.

class AddiPackException : public exception , public exception

#include <addi_pack_exceptions.hpp>

Custom exception class for AddiPack-related errors.

This class extends the standard std::exception to provide detailed error handling specific to the AddiPack library. It includes error codes, messages, and methods to extract error information.

Public Functions

`AddiPackException ( )`

Default constructor for AddiPackException .

`AddiPackException ( const std :: string & message , uint32_t code )`

Constructs an AddiPackException with an optional message and error code.

Parameters :

- **message** – A descriptive error message.
- **code** – An integer error code.

`const char * what ( ) const noexcept override`

Returns a C-style string describing the error.

`uint32_t getCode ( ) const noexcept`

Retrieves the error code associated with the exception.

Returns :

The integer error code.

`AddiDataSource getSource ( uint32_t code )`

Extracts the source from the given error code.

Parameters :

code – The error code to extract the source from.

Returns :

The extracted AddiDataSource.

`AddiDataDomain getDomain ( uint32_t code )`

Extracts the domain from the given error code.

Parameters :

code – The error code to extract the domain from.

Returns :

The extracted AddiDataDomain.

`AddiDataFunction getFunction ( uint32_t code )`

Extracts the function from the given error code.

Parameters :

code – The error code to extract the function from.

Returns :

The extracted AddiDataFunction.

```
AddiDataReason getReason ( uint32_t code )
```

Extracts the reason from the given error code.

Parameters :

code – The error code to extract the reason from.

Returns :

The extracted AddiDataReason.

```
void setMessage ( const std :: string & message )
```

Sets the error message for the exception.

Parameters :

message – The error message to set.

```
void setErrorCode ( uint32_t code )
```

Sets the error code for the exception.

Parameters :

code – The error code to set.

Public Static Functions

```
static std :: string getAddiDataLastError ( )
```

Retrieves the last error message from the AddiData library.

Returns :

The last error message as a string.

C++ UTILS

group **Utilities**

Utilities and helper functions for the CPP API.

This group includes utility functions and helper methods that are shared across modules.

class ArgParser

#include <arg_parser.hpp>

Argument Parser for command line options.

This class provides methods to parse and manage command line arguments

Public Functions

```
ArgParser ( ) = delete
```

ArgParser constructor is deleted to prevent instantiation.

Public Static Functions

```
static void init ( )
```

Initialize the argument parser.

```
static void clear ( )
```

Clear all registered options and flags.

```
static void addOption ( const std :: string & name , char
short_option , const std :: string & description , bool
has_value = true )
```

Add an option with a name, short option, description, and whether it requires a value.

```
static void addFlag ( const std :: string & name , char
short_option , const std :: string & description )
```

Add a flag with a name, short option, and description.

```
static void parse ( int argc , char * argv [ ] )
```

Parse command line arguments.

```
static bool isFlagSet ( const std :: string & name )
```

Check if a flag is set.

```
static bool isOptionSet ( const std :: string & name )
```

Check if an option is set.

```
static std :: string getOptionValue ( const std :: string &
name )
```

Get the value of an option.

User I/O

group **User I/O**

Input/Output User utilities for the CPP API.

I/O related utilities and helper functions.

Functions

```
void printAvailableOptions ( unsigned int valid_mask , std :: vector
< int > & option_index )
```

Utility functions for samples and user interaction.

```
unsigned int getTcwOptionsFromUser ( unsigned int valid_mask )
```

Prompt the user to select options based on a valid mask.

```
std :: string promptString ( const std :: string & message , const
std :: vector < std :: string > & accepted = { } )
```

Prompt the user to input a string, with optional accepted values and default.

```
int promptInt ( const std :: string & message , int min , int max )
```

Prompt the user to input an integer within a specified range, with an optional default value.

```
std :: string getValidString ( const std :: string & name , const
std :: string & message , const std :: vector < std :: string > &
accepted = { } )
```

Get a valid string from the user, ensuring it is within accepted values if provided.

```
int getValidInt ( const std :: string & name , const std :: string &
message , int min , int max )
```

Get a valid integer from the user within a specified range.

```
void exitOnUserInput ( int exit_code )
```

Wait for user input before exiting, then exit with the specified code.

```
void displayAllPortsConfigurations ( std :: uint16_t
direction_mask , const std :: uint8_t * input_values , const std ::
uint8_t * output_values , std :: uint8_t max_port )
```

Display the configuration of all ports in a formatted table.

Parameters :

- **direction_mask** – A bitmask representing the direction of each port, where a 1 indicates output and a 0 indicates input
- **input_values** – An array of input values for each port
- **output_values** – An array of output values for each port
- **max_port** – The maximum number of ports on the card

Device

group **Device Utils**

Device utilities for the CPP API.

Device related utilities and helper functions.

Functions

```
std :: shared_ptr < Device > selectDeviceByCapability ( const std ::
vector < std :: shared_ptr < Device > > & devices , const std ::
string & name , bool ( * predicate ) ( const std :: shared_ptr <
Device > & ) , const std :: string & label )
```

Select a device from a list based on a capability predicate.

```
std :: string selectMode ( const std :: string & name , const std ::
vector < std :: string > & modes , const std :: vector < std ::
string > & descriptions = { } )
```

Select a mode from a list of modes with optional descriptions.

```
std :: vector < std :: string > getValidTimebaseList ( const std ::
shared_ptr < Device > & device )
```

Get the list of valid timebases supported by a device.

Timer Interruptions

group **Timer Interruptions Utils**

Timer Interruption utilities for the CPP API.

Timer interruption related utilities and helper functions.

Functions

```
TIMEBASE_UNITS parseTimebase ( const std :: string & timebase )
```

Parse a timebase string into TIMEBASE_UNITS.

```
void interruptCallback ( AddiDataDeviceStruct , InterruptData data )
```

Callback function to handle interrupts and set a flag.

```
void resetInterruptFlag ( )
```

Reset the interrupt flag.

```
bool hasInterruptHappened ( )
```

Check if an interrupt has occurred.

Help & Informations

All help and informations you needed if you encounter any issue with addipack

Maintenance Information

This section provides detailed guidance for **maintaining, updating, and troubleshooting** Addi-Pack installations. It also includes information about compatibility, driver management, and release history.

Maintenance & Update Guide

Goal

Help you keep Addi-Pack up-to-date and ensure stable, reliable operation of your ADDI-DATA hardware.

This guide covers:

- Updating drivers and libraries
- Verifying DKMS and kernel module status
- Ensuring backward compatibility
- Accessing log and diagnostic files

Updating Addi-Pack

Addi-Pack provides **automatic update support** on all platforms.

Linux

1. Download the new *.deb* or *.rpm* package from your official distribution point (NAS, GitLab CI, etc.).
2. Uninstall the previous version:

```
sudo dpkg -r addipack              # For DEB-based distros
sudo rpm -e addipack               # For RPM-based distros
```

3. Install the new version:

```
sudo dpkg -i addipack-2025.10.1.deb
sudo rpm -i addipack-2025.10.1.rpm
```

4. Reboot your system or reload the kernel modules:

```
sudo modprobe -r apcie1032
sudo modprobe apcie1032
```

Windows

1. Uninstall the previous version via **Add/Remove Programs** or from the **Installer** .
2. Run the new *.exe* installer with **Administrator rights** .
3. Reboot your PC to reload the drivers and services (optional).

> **Note**
>
> Always close your development environment (Visual Studio, Python interpreter, etc.) before updating Addi-Pack to avoid file locks.

Environment Variable

The `AddiPack_ROOT` environment variable is automatically created during installation. It defines the root path used by Addi-Pack tools and libraries to locate:

- C/C++ API libraries
- Python bindings (if installed)
- Sample and utility directories

Normally, no manual action is required. However, you can verify or adjust it for troubleshooting or custom installations.

Windows:

```
echo $Env:AddiPack_ROOT
```

Linux:

```
echo $AddiPack_ROOT
```

If needed, the variable can be redefined manually:

Windows (PowerShell):

```
setx AddiPack_ROOT "C:\Program Files\AddiData\AddiPack"
```

Linux:

```
echo "AddiPack_ROOT=/usr/local/addidata/addipack" | sudo tee -a /etc/
environment
source /etc/environment
```

Tip

The installer automatically updates this variable during upgrades.

Uninstalling Addi-Pack

Linux:

```
sudo apt remove addipack
# or
sudo dnf remove addipack
```

Windows:

Go to *Add or Remove Programs* → select **ADDI-DATA Addi-Pack** → *Uninstall* .

Note

Uninstalling Addi-Pack doesn't remove user projects or log files. For log file locations, see the section below: *Locating Log Files* . However, installing a new version **replaces the official Addi-Pack source files** located in:

- `C:\Users\Public\Addi-Data\AddiPack\` (Windows)
- `/opt/addi-data/AddiPack/` (Linux)

Locating Log Files

All sample applications and tests automatically create structured log files to help diagnose configuration or runtime issues.

Default locations:

- **Linux:** *$HOME/.addi-data/*
- **Windows:** *%USERPROFILE%\AppData\Local\addi-data*

Each log includes:

- Timestamp and card identifier
- Function calls and return codes
- Error messages and driver responses

Tip

Logs can be safely shared with ADDI-DATA support for debugging. Sensitive information such as PCI addresses is anonymized.

Backward Compatibility

Addi-Pack follows a date-based versioning scheme:

`YYYY.MM.PATCH`

- `YYYY` : Release year (e.g., 2025)
- `MM` : Release month (e.g., 09)
- `PATCH` : Incremented for bug fixes or hotfixes within the same month

This versioning ensures clear traceability of releases and simplifies long-term maintenance.

Compatibility Rules:

- Versions with the same `YYYY.MM` are **backward compatible** (only patches differ).
- A bump in `YYYY.MM` may introduce new features.

Troubleshooting & Common Issues

Frequently Asked Questions:

Question	Answer
Why is my card not detected after installation?	Check PCI/PCIe connection, power supply, and run the application with administrator/root rights.
Why do I see “No device found” when running a sample?	Make sure the Addi-Pack driver is loaded (*lsmod | grep addipack*) and that the card is correctly inserted.
Why are interrupts not triggering in my application?	Ensure the interrupt mask, event type, and logic mode are correctly configured.
Why does my timer or counter fail to start?	The reload value might be too small or the timebase might be invalid for the selected device.
Can I use Python and C++ at the same time?	Yes, but not on the same device handle. You must create separate handles per language or process.

Tip

If a card is still not detected, verify PCI/PCIe enumeration using:

```
lspci | grep ADDI
```

or on Windows:

Device Manager → System devices → ADDI-DATA PCIe card .

Contact & Support

For technical assistance or bug reports:

- info@addi-data.com
- https://www.addi-data.com

When contacting support, always include:

- The **card model** (e.g. APCIe-1500)
- The **OS and kernel version**
- The **Addi-Pack version**
- Any **log files** related to your issue
- The **Serial Number** of your product(s)

License & Legal Notice

ADDI-PACK SDK and Tools Copyright (c) 2025 ADDI-DATA GmbH. All rights reserved.
www.addi-data.com | info @ addi-data . com

This package contains components under two different licenses:

1. **Proprietary License** :

   The drivers and binary libraries (located in the `/driver` and `/lib` directories) are proprietary to ADDI-DATA GmbH and may not be reverse engineered, redistributed, or modified without explicit written permission from ADDI-DATA GmbH.

2. **BSD 3-Clause License** :

   The API headers, samples, and development tools are provided under the BSD 3-Clause license (see below), allowing integration, modification, and redistribution with attribution, provided that the copyright notice and this license appear in all copies.

BSD 3-Clause License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of ADDI-DATA GmbH nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

For more information or support:

ADDI-DATA GmbH www.addi-data.com | info @ addi-data . com

© Copyright 2025, Addi-Data GmbH.
Created using Sphinx 8.2.3.