HTDC - API guide

Aurea Technology

Created on : Octorber, 2021

Last updated : July, 2022

Contents

Introduction

The High performance Time to Digital Converter (ChronoXea) can be controlled and used with a computer through the USB connection. To do that we provide an API (Application Programming Interface) based on a library developed in C/C++ language for all operating systems (Windows, MacOS, Linux).

Through it, you can develop your own HTDC control interface. In order to help you, we provide the library files, and some examples in C++ and Python for all operating systems.

CHAPTER 1

Software Installation Guide

The following section describes how to install the ChronoXea software on Windows, MacOS and Linux operating systems.

1.1 Windows

1.1.1 Operating Systems Requirements

- Windows 7 or higher
- Application and examples are working on 32bit and 64bit systems.

1.1.2 Installation Step

1. Run the setup file locate in provided directory.
2. Connect ChronoXea device to your computer with the USB cable.
3. Start Aurea-ChronoXea application or start Aurea-Launcher and then click on your device to use the software.

1.2 MacOS

- Aurea-Launcher Installation :
  1. Double click on Aurea-Launcher.dmg file.
  2. Drag Aurea-Launcher in the Applications folder.
- Aurea-ChronoXea Installation :
  1. Double click on Aurea-ChronoXea.dmg file.
  2. Drag Aurea-ChronoXea in the Applications folder
  3. Connect ChronoXea device to your computer with the USB cable.
  4. Launch Aurea-ChronoXea or Aurea-Launcher application by clicking on it.

1.3 Linux

- Aurea-Launcher Installation :
  1. Unzip Aurea-Launcher-package.zip
  2. Go to Aurea-Launcher-package/Aurea-Launcher and double-click on Aurea-Launcher-Installer.
  3. Follow the installer instructions and make sure to install all Aurea Technology software in the same directory.
- Aurea-ChronoXea Installation :
  1. Unzip Aurea-ChronoXea-package.zip
  2. Go to Aurea-ChronoXea-package/Aurea-ChronoXea and double-click on Aurea-ChronoXea-Installer.
  3. Follow the installer instructions and make sure to install all Aurea Technology software in the same directory.
  4. Connect ChronoXea device to your computer with the USB cable.
  5. Launch Aurea-ChronoXea or Aurea-Launcher application by executing the following command in the installation directory.

```
./Aurea-Launcher.sh
```

```
./Aurea-ChronoXea.sh
```

CHAPTER 2

Custom Application

The following section guides you through the developement of your own application.

2.1 Windows

2.1.1 Requirements

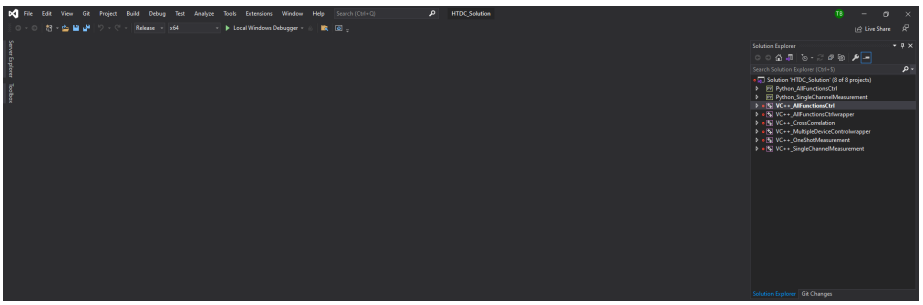
Software : Visual Studio (2019)

Visual Studio extensions :

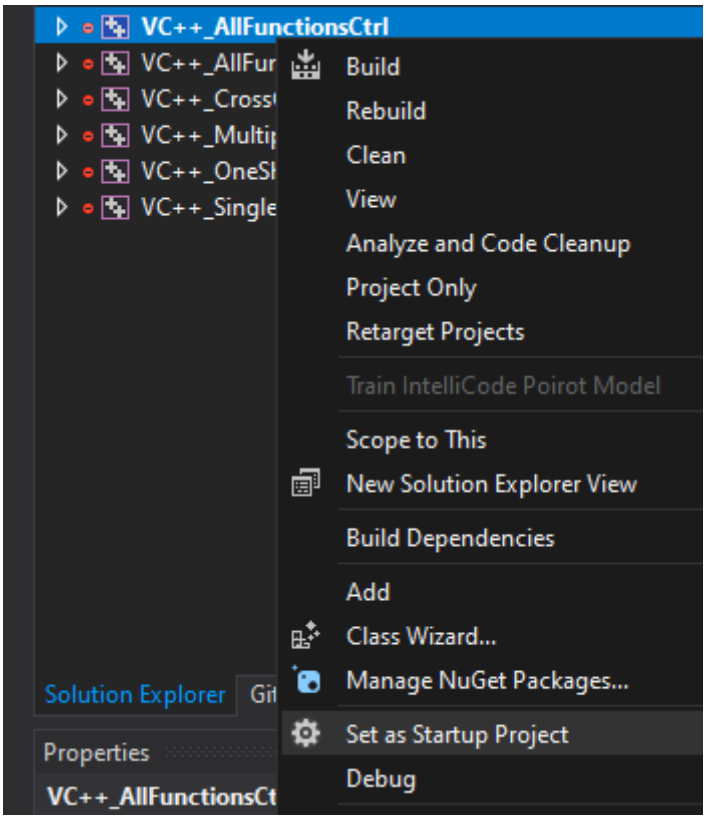
- Desktop Development C++
- Development Python

2.1.2 Create C++ application using the existing Visual Studio project

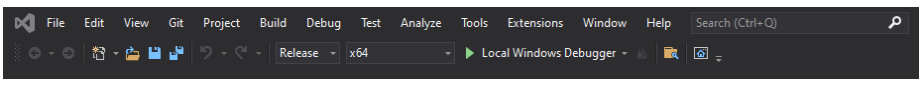
- Locate the HTDC-API folder and go to “HTDC-API/Applications/”.
- Open “HTDC_Solution.sln”.



- Three differents C++ programs have been developed to help you, to change the selected applications right click on the desired example and select “Set as Startup Project”.

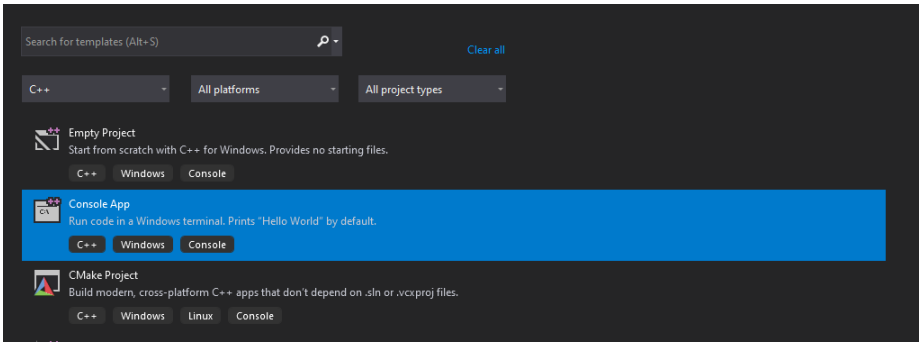


- To test those examples, make sure you have selected “Release” and “x64”, and click on “Run”.

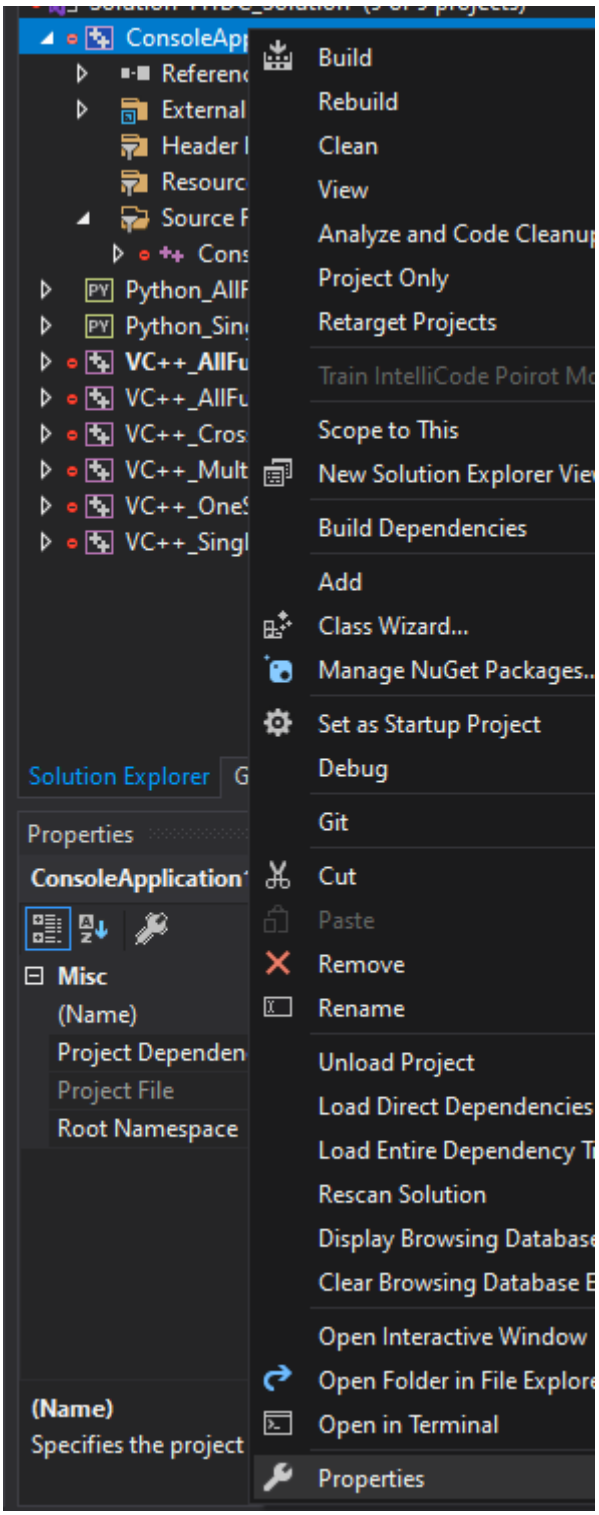


- To create your own application click on “File” > “New” > “Project...” Then select “Console App”, choose an application name, select “Add

to solution" and click on "Create".



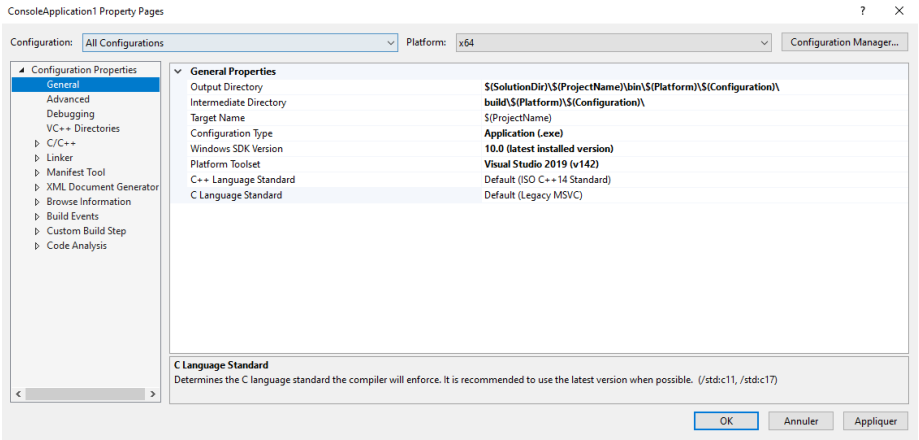
- Now, you need to configure your project, right click on it and then click on "Properties".



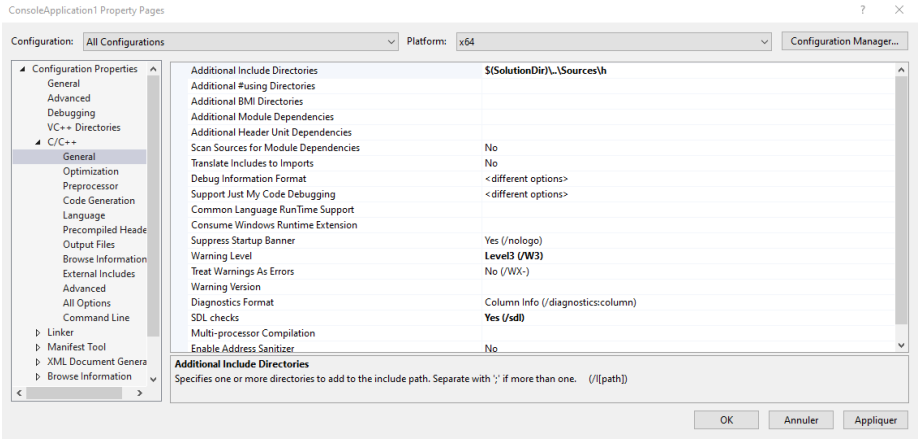
- Then make sure Configuration section is set to “All Configurations”

and Platform is set to “x64”.

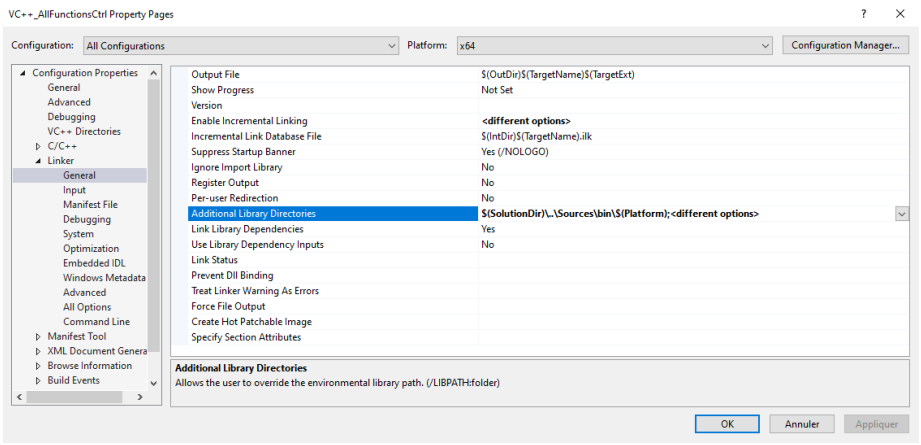
- Go to “Configuration Properties” > “General” > “General Properties” and set the “Output Directory” to “$(SolutionDir)\$(ProjectName)\bin\$(Platform)\$(Configuration)\”.
  Then set the “Intermediate Directory” to “build\$(Platform)\$(Configuration)\”.



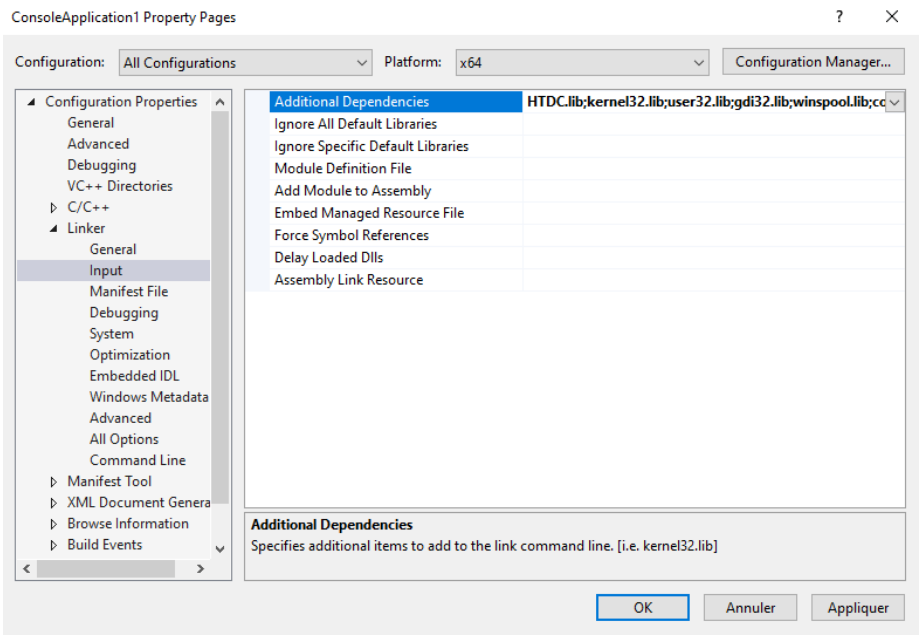
- Go to “Configuration Properties” > “C/C++” > “General” and set the “Additional Include Directories” to “$(SolutionDir)\..\Sources\h”.



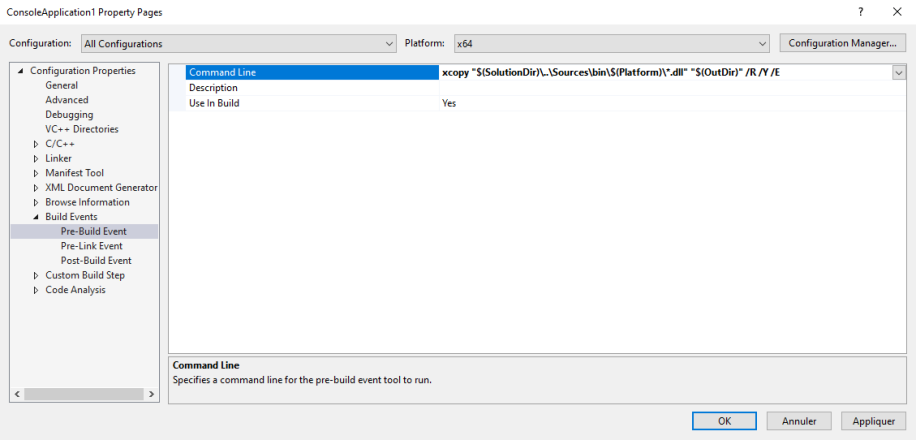
- Go to “Configuration Properties” > “Linker” > “General” and set the “Additional Library Directories” to “$(SolutionDir)\..\Sources\bin\$(Platform)”.



- Go to “Configuration Properties” > “Linker” > “Input” and set the “Additional Dependencies” to “HTDC.lib”.



- Finally, in the goal to locally run the application, add the copy of the library on the output folder. Go to “Configuration Properties” > “Build Events” > “Pre-Build Event” and set the “Command Line” to “xcopy “$(SolutionDir)\..\Sources\bin\$(Platform)*.dll” “$(OutDir)” /R /Y /E”.

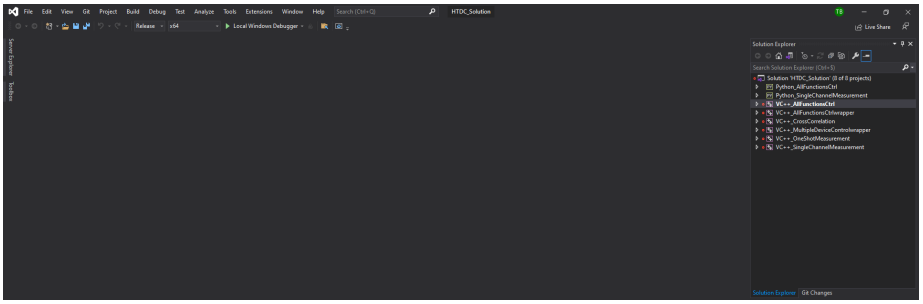


Note: If you use windows 7, the xcopy command may give you an error. To solve this issue, add this path to your environment path : "C:WindowsSystem32". You also can remove the xcopy command and manually copy the HTDC.dll file in the same directory as your application executable file.

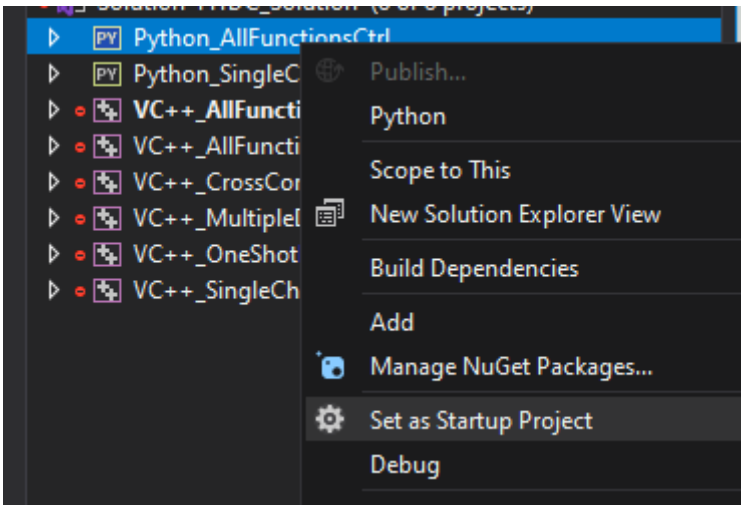
Your C++ application is now ready to use. Please refer to Section *Code Examples* to access basic code.

2.1.3 Create Python application using the existing Visual Studio project

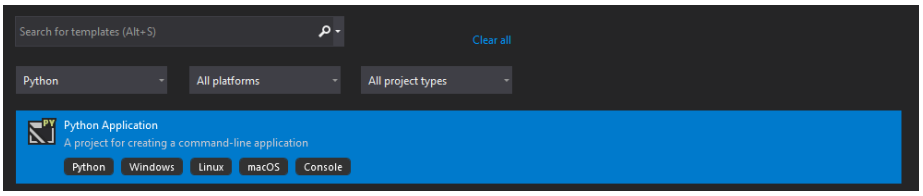
- Locate the HTDC-API folder and go to "HTDC-API/Applications/".
- Open "HTDC_Solution.sln".



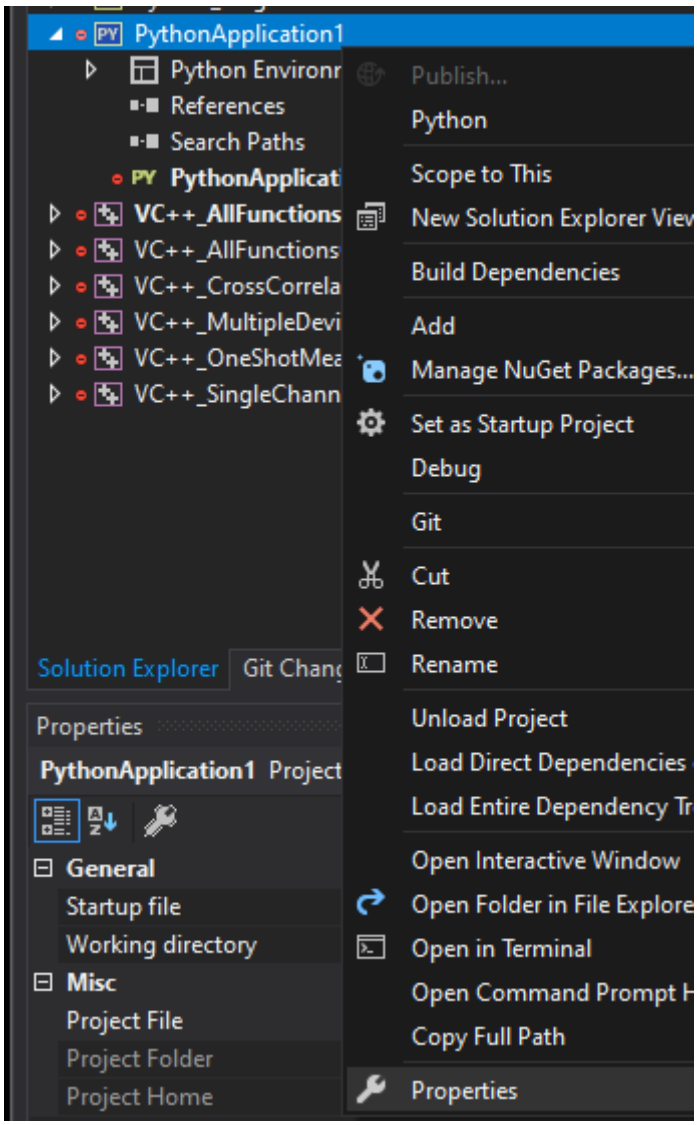
- One Python program has been developed to help you, to change the selected applications right click on the desired example and select "Set as Startup Project".



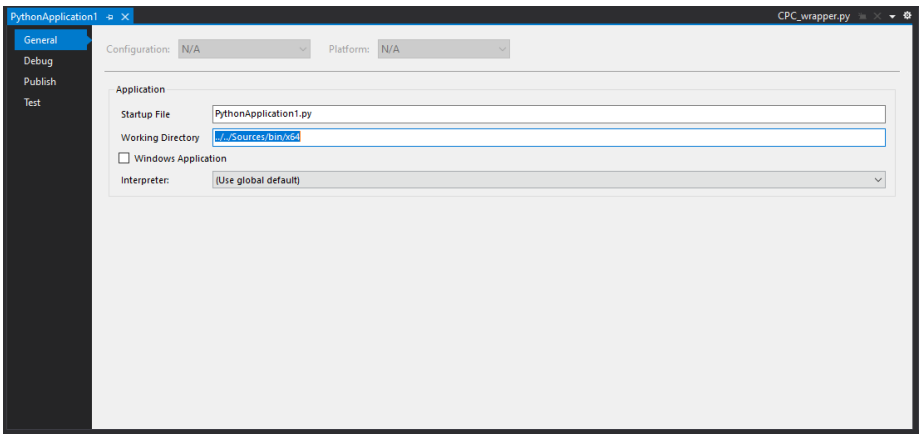
- To create your own application click on “File” > “New” > “Project...” then select “Python Application”, choose an application name, select “Add to solution” and click on “Create”.



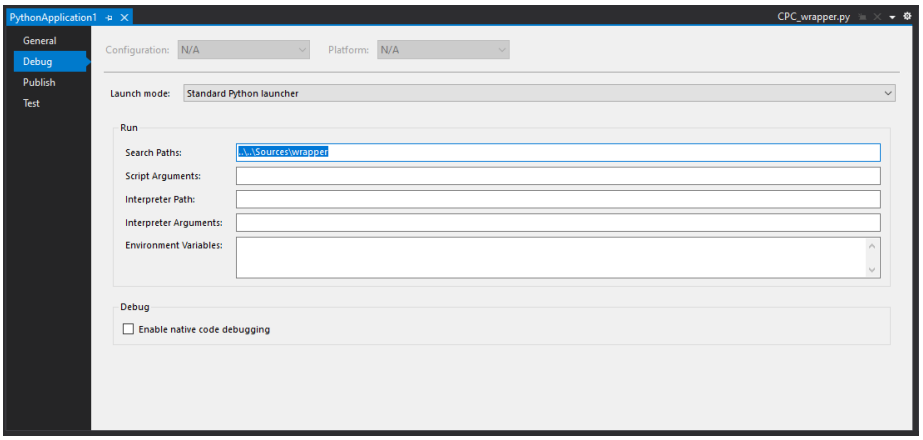
- Now, you need to configure your project, right click on it and then click on “Properties”.



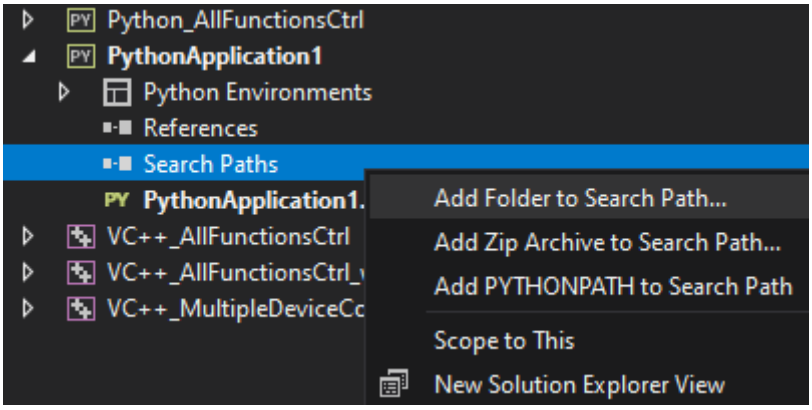
- In order to not locally copy the library, you can adjust the "Working Directory" with the library path "../../Sources/bin/x64"



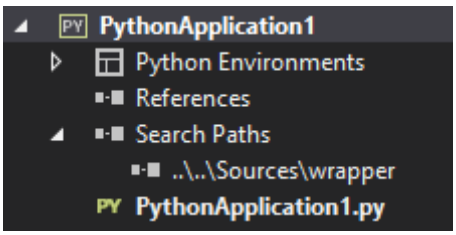
- In “Debug” set the “Search Paths” to “..\..\Sources\wrapper” allowing to specify the wrapper package location



- Finally, right click on “Search Paths” and select “Add Folder to Search Path…”.



- Then locate and select “HTDC-API/Sources/wrapper”.



Your Python application is now ready to use. Please refer to Section *Code Examples* to access basic code.

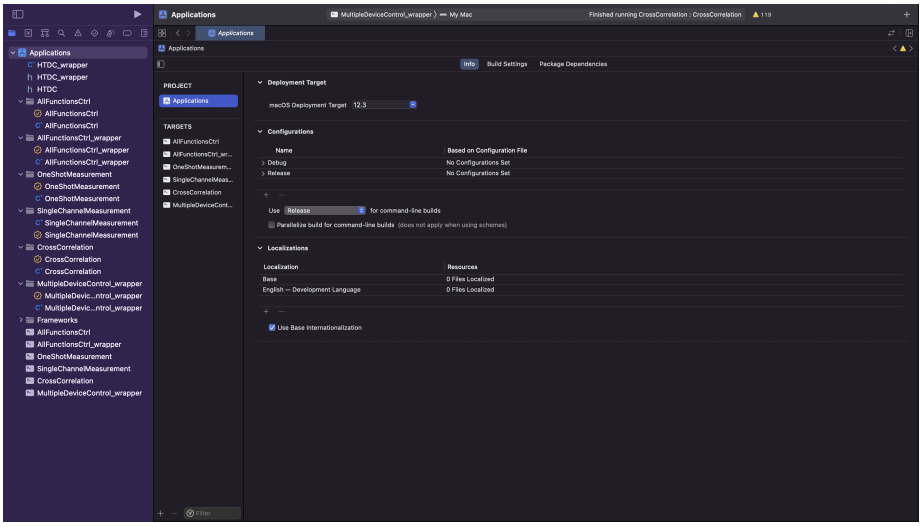
2.2 MacOS

2.2.1 Requirements

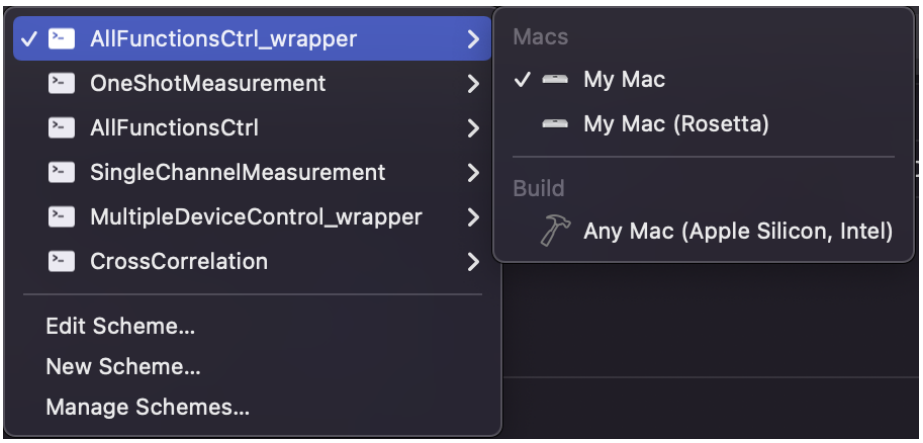
Software : XCode.

2.2.2 Create C++ application using the existing Xcode project

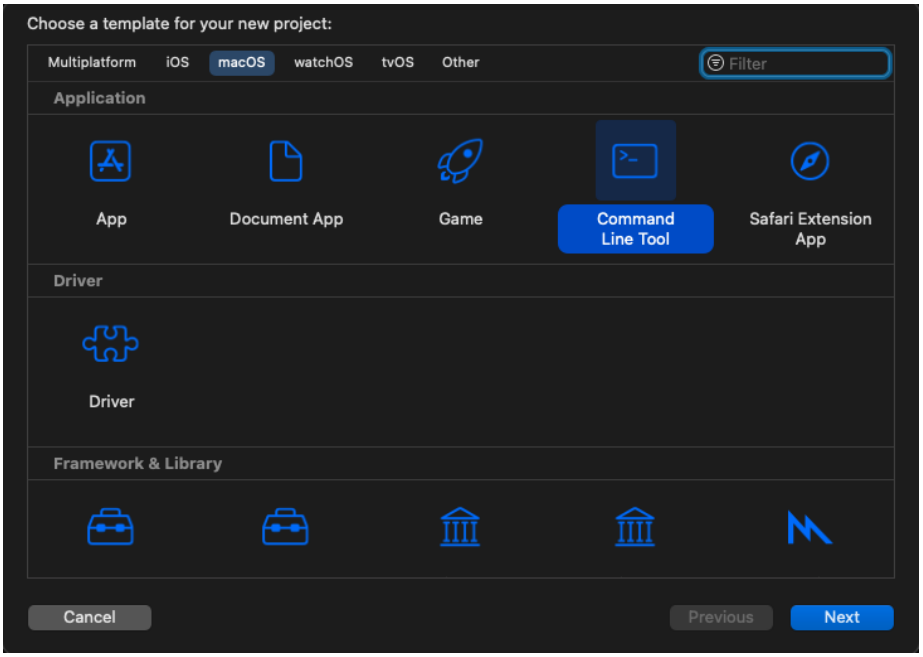
- Locate the HTDC-API folder and go to "HTDC-API/Applications/C++/".
- Open "Applications.xcodeproj".



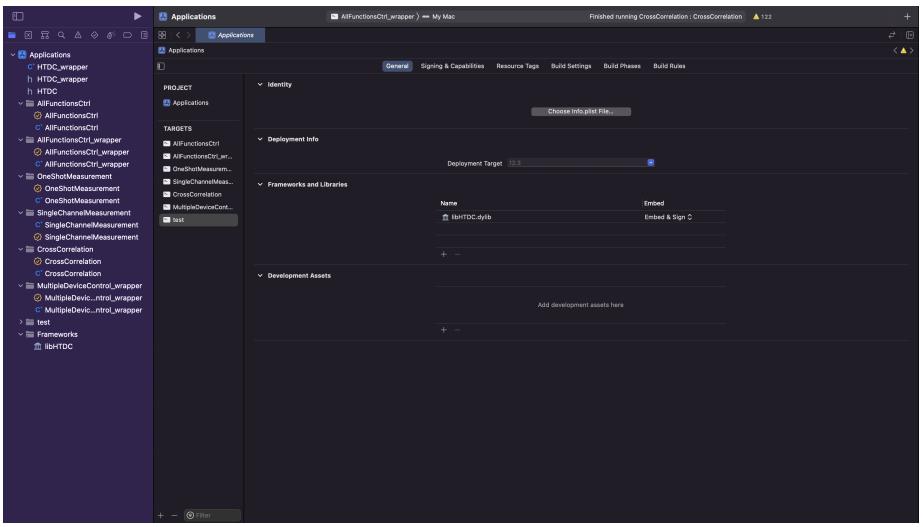
- Three differents programs have been developed to help you, to change the selected applications click on the list at the top.



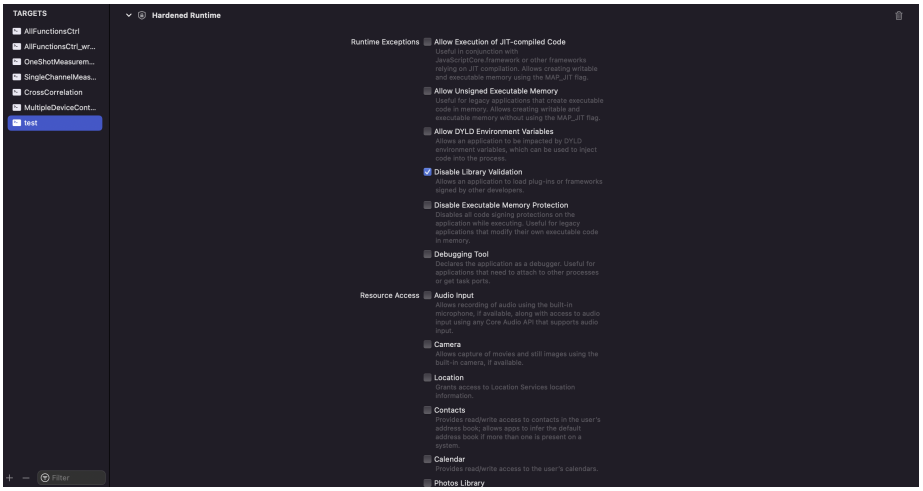
- To create your own application click on “File” > “New” > “Target…”” Then select Command Line Tools, choose an application name and click on “Finish”.



- Now, you need to configure your target, click on it, click on “General”, on “Frameworks and Libraries” and click “+”.
- Click on “Add Ohter…” and select “Add Files”. Then add libHTDC.dylib that is locate in “HTDC-API/Sources/bin/”.



- To avoid library issue, click on "Signing & Capabilities", check box "Disable Library Validation".



Your C++ application is now ready to use. Please refer to Section *Code Examples* to access basic code.

2.2.3 Create Python application

- Locate the HTDC-API folder and go to "HTDC-API/Applications/Python/".
- An Example has been developed to help you, to develop your own, we advise you to copy AllFunctionsCrtl.py, rename it and make your modifications. Please make sure you have HTDC software installed and you are using HTDC_wrapper.py file.

Your Python application is now ready to use. Please refer to Section *Code Examples* to access basic code.

2.3 Linux

2.3.1 Requirements

Package : build-essential, libudev-dev

To install these packages, please execute the following commands :

```
sudo apt update
sudo apt upgrade
sudo apt build-essential
sudo apt install libudev-dev
sudo apt install libusb-1.0-0-dev
```

Note: If you have installed the Aurea-ChronoXea Software, you may already have the necessary packages.

2.3.2 Makefile Example

- Locate the HTDC-API folder and go to "HTDC-API/Applications/C++".
- Three differents programs have been developed to help you, To create your own application, we advise you to copy the AllFunctionsCrtl folder. Then rename the folder and the AllFunctionCrtl.cpp file. Finally replace the target name by your application name in the Makefile.

```
CC = g++
CFLAGS  = -Wall -pthread

# The build target
TARGET = AllFunctionsCtrl
INCLUDE = -I../../../Sources/h/

.PHONY: all
all: ${TARGET}

$(TARGET): $(TARGETPATH)$(TARGET).cpp
    $(CC) $(INCLUDE) $(CFLAGS) -o $(TARGET) $(TARGET).cpp -lHTDC

.PHONY: clean
clean:
    -${RM} ${TARGET}
```

- Finally edit the cpp file to develop your application.

2.3.3 Create Python application

- Locate the HTDC-API folder and go to "HTDC-API/Applications/Python/".
- An Example has been devloped to help you, to develop your own, we advise you to copy AllFunctionsCrtl.py, rename it and make your modifications. Please make sure you have HTDC software installed and you are using HTDC_wrapper.py file.

Your Python application is now ready to use. Please refer to Section *Code Examples* to access basic code.

CHAPTER 3

Code Examples

The following section present simple codes in C++ and Python to use ChronoXea device.

3.1 Communication

This first example shows how to list all ChronoXea connected to a computer and how to open and close USB communication. Device information is also recovered in this example.

3.1.1 C++ program

```
#include <iostream>
using namespace std;

#include "HTDC.h"

////////////////////////////////////////
// Channels number available
#define HTDC_N_CH       4

// Number of maximum data to recover
#define DATA_MAX        1000000
////////////////////////////////////////
```

(continues on next page)

(continued from previous page)

```
int main(int argc, const char* argv[]) {
        short iDev = 0;
        short ret;
        char* devicesList[10];
        short numberDevices;
        char* pch;
        char* next_pch = NULL;
        char version[64];
        char versionParam[3][32];
        char systemName[6];
        memset(version, ' ', 64);
        memset(systemName, '\0', 6);

        /*      listDevices function    */
        // List Aurea Technology devices: MANDATORY BEFORE EACH␣
↪OTHER ACTION ON THE SYSTEM
        if (HTDC_listDevices(devicesList, &numberDevices) == 0) {
                if (numberDevices == 0) {
                        cout << endl << "       Please connect␣
↪device !" << endl << endl;
                        do {
                                delay(500);
                                HTDC_listDevices(devicesList, &
↪numberDevices);
                        } while (numberDevices == 0);
                }
        }

        // If multiple HTDC devices are detected, select one␣
↪else open it
        if (numberDevices > 1) {
                for (int i = 0; i < numberDevices; i++) {
                        printf(" -%u: %s\n", i, devicesList[i]);
                }
                cout << endl << "Select device to drive: ";
                cin >> iDev;

                if (HTDC_openDevice(iDev) != 0) {
                        cout << "Failed to open HTDC" << endl;
                }
        }
        else {
                iDev = 0;
                if (HTDC_openDevice(iDev) != 0) {
                        cout << "Failed to open HTDC" << endl;
                }
        }
```

(continues on next page)

(continued from previous page)

```
        // Recovery of the system version
        if (HTDC_getSystemVersion(iDev, version) == 0) {
                cout << endl << "  * Hardware-Firmware version:"
↪<< endl << endl;
        }
        else {
                cout << endl << " -> Failed to get system version
↪" << endl << endl;
        }

        // Loop to extract HTDC parameters
        int v = 0;
        pch = secure_strtok(version, ":", &next_pch);
        while (pch != NULL) {
                snprintf((char*)&versionParam[v][0], 32, "%s",␣
↪pch);
                pch = secure_strtok(NULL, ":", &next_pch);
                v++;
        }
        if (pch != 0) { snprintf((char*)&versionParam[v][0], 32,
↪"%s", pch); }

        // Display HTDC information
        memcpy(systemName, (char*)&versionParam[2][0] + 3, 3);
        cout << "       AT System       : " << systemName <<␣
↪endl;
        cout << "       Serial number   : " << versionParam[0] <
↪< endl;
        cout << "       Product number  : " << versionParam[1] <
↪< endl;
        cout << "       Firmware version: " << versionParam[2] <
↪< endl;
        cout << endl;

        // Wait some time
        delay(2000);

        /*      CloseDevice function    */
        // Close initial device opened: MANDATORY AFTER EACH END␣
↪OF SYSTEM COMMUNICATION.
        if (HTDC_closeDevice(iDev) == 0) cout << "   ->␣
↪Communication closed" << endl;
        else cout << "  -> Failed to close communication" <<␣
↪endl;

        return 0;
```

(continues on next page)

(continued from previous page)

```
}
```

3.1.2 Python program

```
from ctypes import *
import time

# Import HTDC wrapper file
import HTDC_wrapper as ChronoXea

# Application main
def main():
        key = ''
        iDev = c_short(0)
        nDev = c_short()
        devList = []

        # Scan and open selected device
        devList,nDev=ChronoXea.listDevices()
        if nDev==0:   # if no device detected, wait
                print ("No device connected, waiting...")
                while nDev==0:
                devList,nDev=ChronoXea.listDevices()
                time.sleep(1)
        elif nDev>1:  # if more 1 device detected, select target
                print("Found " + str(nDev) + " device(s) :")
                for i in range(nDev):
                print (" -"+str(i)+": " + devList[i])
                iDev=int(input("Select device to open (0 to n):
↪"))

        # Open device
        if ChronoXea.openDevice(iDev)<0:
                input(" -> Failed to open device, press enter to␣
↪quit !")
                return 0
        print("Device correctly opened")

        # Recover system version
        ret,version = ChronoXea.getSystemVersion(iDev)
        if ret<0: print(" -> failed\n")
        else:print("System version = {} \n".format(version))

        # Wait some time
        time.sleep(2)
```

(continues on next page)

(continued from previous page)

```
        # Close device communication
        ChronoXea.closeDevice(iDev)

# Python main entry point
if __name__ == "__main__":
        main()
```

3.2 Single Channel Measurement

The next example shows how to use ChronoXea to make a measurement on a single channel.

3.2.1 C++ program

```
#include <iostream>
using namespace std;

#include "HTDC.h"

////////////////////////////////////////
// Target channel
#define TARGET_CH       CH_1

// Number sample time to recover
#define N_SAMPLE        100000
////////////////////////////////////////

// Input edge
#define RISING_EDGE     0
#define FALLING_EDGE    1

// input state
#define ON      1
#define OFF     0

// Input level
#define TTL_CMOS        0
#define NIM                     1

// Sync source
#define EXTERNAL_SYNC   0
#define INTERNAL_SYNC   1
```

(continues on next page)

(continued from previous page)

```
// Measurement mode
#define CONTINUOUS_MODE 0
#define ONESHOTE_MODE   1

int main(int argc, const char* argv[]) {
        char* devicesList[10];
        short numberDevices, iDev;
        unsigned long long nData = 0;
        unsigned long n;
        unsigned long nSampleToRecover, nSampleRecovered = 0;
        int status;
        short ret;
        short system_chNumber, system_integrationMode;

        // Memory allocation to recover data from HTDC
        // Offset of 65535 mandatory for exceeding data
        unsigned long long* data;
        data = (unsigned long long*)malloc(N_SAMPLE + 65536);

        if (data == NULL) {
                printf(" --> Failed to assign memory ! \n");
                system("pause");
                return -1;
        }

        // Loop to list devices until at least one is founded
        ret = HTDC_listDevices(devicesList, &numberDevices);
        if (ret == 0) {
                if (numberDevices == 0) {
                        cout << endl << "Please connect AT␣
↪device !" << endl << endl;
                        do {
                                delay(500);
                                ret = HTDC_
↪listDevices(devicesList, &numberDevices);
                        } while (numberDevices == 0);
                }
        }

        // If multiple HTDC devices are detected, select one␣
↪else open it
        if (numberDevices > 1) {
                for (int i = 0; i < numberDevices; i++) {
                        printf(" -%u: %s\n", i, devicesList[i]);
                }
                cout << endl << "Select device to drive: ";
                cin >> iDev;
```

(continues on next page)

(continued from previous page)

```
                if (HTDC_openDevice(iDev) == 0) {
                        cout << "HTDC " << iDev << " correctly␣
↪open !";
                }
                else cout << "Failed to open HTDC" << endl;
        }
        else {
                iDev = 0;
                if (HTDC_openDevice(iDev) == 0) {
                        cout << "HTDC " << iDev << " correctly␣
↪open !";
                }
                else cout << "Failed to open HTDC" << endl;
        }

        // Get system features
        //--------------------

        // Get the number of channels available
        HTDC_getSystemFeature(iDev, 0, &system_chNumber);

        // Get the system integration mode
        HTDC_getSystemFeature(iDev, 1, &system_integrationMode);

        // Adjust Sync divider to 1
        cout << "Set Sync divider" << endl;
        if (HTDC_setSyncDivider(iDev, 1) != 0)
                cout << " -> Failed " << endl;
        else
                cout << " -> Done" << endl;

        // Configure Sync input : enable, rising edge and TTL-
↪CMOS level
        cout << "Set input Sync" << endl;
        if (HTDC_setSyncInputConfig(iDev, ON, RISING_EDGE, TTL_
↪CMOS) != 0)
                cout << " -> Failed " << endl;
        else
                cout << " -> Done" << endl;

        // Apply a 5.5ns delay on target channel
        cout << "Set channel delay" << endl;
        if (HTDC_setChannelDelay(iDev, TARGET_CH, 5.5) != 0)
                cout << " -> Failed " << endl;
        else
                cout << " -> Done" << endl;
```

(continues on next page)

(continued from previous page)

```
        // Activate target channel input: power ON and rising␣
↪edge
        cout << "Channel activation" << endl;
        if (HTDC_setChannelConfig(iDev, TARGET_CH, ON, RISING_
↪EDGE, TTL_CMOS) != 0)
                cout << " -> Failed " << endl;
        else
                cout << " -> Done" << endl;

        // Set channel measurement mode to Continuous
        cout << "Set channel measurement mode" << endl;
        if (HTDC_setMeasMode(iDev, TARGET_CH, CONTINUOUS_MODE) !
↪= 0)
                cout << " -> Failed" << endl;
        else
                cout << " -> Done" << endl;

        // Arm target channel: need of 10000 data during␣
↪infinite time
        cout << "Arm channel" << endl;
        nSampleToRecover = 10000;
        if (HTDC_armChannel(iDev, TARGET_CH, nSampleToRecover, -
↪1) != 0)
                cout << " -> Failed" << endl;
        else
                cout << " -> Done" << endl;

        // Waiting for Sync signal stabilization...
        cout << "Waiting for sync signal stabilization..." <<␣
↪endl;
        delay(3000);

        // Measurement
        //------------

        // Start target channel
        cout << "Start channel" << endl;
        if (HTDC_startChannel(iDev, TARGET_CH) != 0)
                cout << " -> Failed" << endl;
        else
                cout << " -> Done" << endl;

        // Recover data
        cout << "Recover target channel data... " << endl;
        while (nSampleRecovered < nSampleToRecover) {
                // Recover target channel state, to known how␣
↪much data are available
```

(continues on next page)

(continued from previous page)

```
                HTDC_getChannelState(iDev, TARGET_CH, &status, &
↪nSampleToRecover, &nSampleRecovered);

                // Store sample data
                switch (TARGET_CH) {
                        //case CH_1: if (HTDC_getCh1Data(iDev,␣
↪data, &n) != 0) { cout << "Channel 1 : Failed to recover data !
↪" << endl; } break;
                case CH_1: if (HTDC_getCh1Data(iDev, data +␣
↪nData, &n) == 0) { nData += n; } break;
                case CH_2: if (HTDC_getCh2Data(iDev, data +␣
↪nData, &n) == 0) { nData += n; } break;
                case CH_3: if (HTDC_getCh3Data(iDev, data +␣
↪nData, &n) == 0) { nData += n; } break;
                case CH_4: if (HTDC_getCh4Data(iDev, data +␣
↪nData, &n) == 0) { nData += n; } break;
                }

                delay(100);
                printf("\r %lu/%lu data recovered",␣
↪nSampleRecovered, nSampleToRecover);
        }

        // Stop target channel
        cout << "Stop channel" << endl;
        if (HTDC_stopChannel(iDev, TARGET_CH) != 0)
                cout << " -> Failed" << endl;
        else
                cout << " -> Done" << endl;

        // Display sample result
        cout << "\nSample time recovered:" << endl;
        for (int i = 0; i<int(nData); i++) {
                printf(" sample[%i]= %1.3fns \n", i, data[i] *␣
↪HTDC_RES);
                if (i > 10) break;
        }

        // Wait some time
        delay(2000);

        // Close device communication
        HTDC_closeDevice(iDev);
        return 0;
}
```

3.2.2 Python program

```
from ctypes import *
import time

# Import HTDC wrapper file
import HTDC_wrapper as ChronoXea

# Application main
def main():
        iDev=int(0)
        nDev=c_short()
        devList=[]
        nSampleRecovered=int(0)
        nSampleToRecover=int(0)
        sampleList=[]

        # List and display avaliable devices
        devList,nDev=ChronoXea.listDevices()
        if nDev==0:   # if no device detected, wait
                print ("No device connected, waiting...")
                while nDev==0:
                devList,nDev=ChronoXea.listDevices()
                time.sleep(1)
        elif nDev>1:  # if more 1 device detected, select target
                print("Found " + str(nDev) + " device(s) :")
                for i in range(nDev):
                print (" -"+str(i)+": " + devList[i])
                iDev=int(input("Select device to open (0 to n):
↪"))

        # Open device
        if ChronoXea.openDevice(iDev)<0:
                input(" -> Failed to open device, press enter to␣
↪quit !")
                return 0
        print("Device correctly opened")

        # Set sync source: in internal
        print("Sync source")
        ret = ChronoXea.setSyncSource(iDev, 1)
        if ret == 0:
                print("\nSync Source Set\n")
        else:
                print("\nset Sync Source: error\n")

        # Set internal sync frequency to 10kHz
        print("Internal Sync frequency")
```

(continues on next page)

(continued from previous page)

```
        ret = ChronoXea.setInternalSyncFrequency(iDev, 10000)
        if ret == 0:
                print("\nInternal Sync frequency Set\n")
        else:
                print("\nset Internal Sync frequency: error\n")

        # Set sync divider to 1
        print("Sync divier")
        ret = ChronoXea.setSyncDivider(iDev, 1)
        if ret == 0:
                print("\nSync Divider Set\n")
        else:
                print("\nset Sync Divider: error\n")

        # Set sync input configuration: Enable, rising edge and␣
↪TTL-CMOS level
        print("Set input Sync")
        ret = ChronoXea.setSyncInputConfig(iDev,1,0,0)
        if ret == 0:
                print("\nSync Input Config Set\n")
        else:
                print("\nset Sync Input Config: error\n")

        # Set target channel delay to 5.5ns
        print("Sync channel delay")
        ret = ChronoXea.setChannelDelay(iDev, TARGET_CH, 5.5)
        if ret == 0:
                print("\nChannel Delay Set\n")
        else:
                print("\nset Channel delay: error\n")

        # Set channel(s) configuration: Power ON, rising edge␣
↪and TTL-CMOS level
        print("Channel configuration")
        ret = ChronoXea.setChannelConfig(iDev, TARGET_CH, 1, 0,␣
↪0)
        if ret == 0:
                print("\nChannel config Set\n")
        else:
                print("\nset Channel config: error\n")

        # Arm channel(s): arm target channel to recover N_SAMPLE
        print("Arm channel")
        ret = ChronoXea.armChannel(iDev,TARGET_CH,N_SAMPLE)
        if ret == 0:
                print("\nChannel Armed\n")
        else:
```

(continues on next page)

(continued from previous page)

```
                print("\nArm Channel: error\n")
        nSampleToRecover=N_SAMPLE

        print("Waiting stable sync signal... ")
        time.sleep(5)

        # Start channel(s): start target channel
        print("Start channel")
        ret = ChronoXea.startChannel(iDev, TARGET_CH)
        if ret == 0:
                print("\nChannel Started\n")
        else:
                print("\nStart Channel: error\n")

        # Recover data
        print("Recover target channel data... ")
        while nSampleRecovered<nSampleToRecover:

                # Recover target channel state, to known how␣
↪much data are available
                ret, state,nSampleToRecover,nSample = ChronoXea.
↪getChannelState(iDev, TARGET_CH)
                if(ret == 0):
                    nSampleRecovered += nSample

                    # Get channel data
                    ret,n,sample = ChronoXea.getChannelData(iDev,
↪ TARGET_CH)
                    if ret==0:
                        # Store result if data available
                        if n>0: sampleList+=sample

                        # Wait and display progression
                        time.sleep(0.5)
                        print("\r State: {} | {}/{} data␣
↪recovered".format(state,nSampleRecovered,nSampleToRecover))
                    else:
                        print("\nGet Channel Data: error\n")
                else:
                    print("\nchannel State: error\n")

        # Display part of data recovered
        print("\nSample time recovered:")
        for i in range (10):
                print(" sample[{}]={}ns".format(i,
↪round(sampleList[i]*ChronoXea.HTDC_RES,3)))
```

(continues on next page)

(continued from previous page)

```
        print("\nEnd of program")

        # Close device
        ChronoXea.closeDevice(iDev)

# Python main entry point
if __name__ == "__main__":
        main()
```

Note: All function information is available in section *All Functions*.

3.3 OneShot Measurement

The next example shows how to use ChronoXea to make a One Shot Measurement

3.3.1 C++ program

```
#include <iostream>
using namespace std;

#include "HTDC.h"

////////////////////////////////////////
// Target channel
#define TARGET_CH       CH_1

// Number shot to apply
#define N_SHOT  10

// Measurement time in ms (between 100 to 100ms)
#define MEAS_TIME       200

// Deadtime betwwen 2 shot in ms
#define MEAS_DEADTIME   100

// Number sample time to recover
#define N_SAMPLE        100000
////////////////////////////////////////

// Input edge
#define RISING_EDGE     0
#define FALLING_EDGE    1
```

(continues on next page)

(continued from previous page)

```
// input state
#define ON      1
#define OFF     0

// Input level
#define TTL_CMOS        0
#define NIM                     1

// Sync source
#define EXTERNAL_SYNC   0
#define INTERNAL_SYNC   1

// Measurement mode
#define CONTINUOUS_MODE 0
#define ONESHOTE_MODE   1

int main(int argc, const char* argv[]) {
        char* devicesList[10];
        short numberDevices, iDev;
        long n;
        double bw;
        short ret;
        short system_chNumber, system_integrationMode;
        double* pHistoX[N_SHOT];
        double* pHistoY[N_SHOT];

        /*      listDevices function    */
        // List Aurea Technology devices: MANDATORY BEFORE EACH␣
↪OTHER ACTION ON THE SYSTEM
        if (HTDC_listDevices(devicesList, &numberDevices) == 0) {
                if (numberDevices == 0) {
                        cout << endl << "       Please connect␣
↪device !" << endl << endl;
                        do {
                                delay(500);
                                HTDC_listDevices(devicesList, &
↪numberDevices);
                        } while (numberDevices == 0);
                }
        }

        // If multiple HTDC devices are detected, select one␣
↪else open it
        if (numberDevices > 1) {
                for (int i = 0; i < numberDevices; i++) {
                        printf(" -%u: %s\n", i, devicesList[i]);
```

(continues on next page)

(continued from previous page)

```
                }
                cout << endl << "Select device to drive: ";
                cin >> iDev;

                if (HTDC_openDevice(iDev) == 0) {
                        cout << "HTDC " << iDev << " correctly␣
↪open !";
                }
                else cout << "Failed to open HTDC" << endl;
        }
        else {
                iDev = 0;
                if (HTDC_openDevice(iDev) == 0) {
                        cout << "HTDC " << iDev << " correctly␣
↪open !";
                }
                else cout << "Failed to open HTDC" << endl;
        }

        // Get system features
        //--------------------

        // Get the number of channels available
        HTDC_getSystemFeature(iDev, 0, &system_chNumber);

        // Get the system integration mode
        HTDC_getSystemFeature(iDev, 1, &system_integrationMode);

        // Adjust Sync divider to 1
        cout << "Set Sync divider" << endl;
        if (HTDC_setSyncDivider(iDev, 1) != 0)
                cout << " -> Failed " << endl;
        else
                cout << " -> Done" << endl;

        // Configure Sync input : enable, rising edge and TTL-
↪CMOS level
        cout << "Set input Sync" << endl;
        if (HTDC_setSyncInputConfig(iDev, ON, RISING_EDGE, TTL_
↪CMOS) != 0)
                cout << " -> Failed " << endl;
        else
                cout << " -> Done" << endl;

        // Activate target channel input: power ON and rising␣
↪edge
        cout << "Channel activation" << endl;
```

(continues on next page)

(continued from previous page)

```
        if (HTDC_setChannelConfig(iDev, TARGET_CH, ON, RISING_
↪EDGE, TTL_CMOS) != 0)
                cout << " -> Failed " << endl;
        else
                cout << " -> Done" << endl;

        // Set channel measurement mode to Continuous
        cout << "Set channel measurement mode" << endl;
        if (HTDC_setMeasMode(iDev, TARGET_CH, ONESHOTE_MODE) !=␣
↪0)
                cout << " -> Failed" << endl;
        else
                cout << " -> Done" << endl;

        // Arm target channel: need of 10000 data during␣
↪infinite time
        cout << "Arm channel" << endl;
        if (HTDC_armChannel(iDev, TARGET_CH, -1, MEAS_TIME) != 0)
                cout << " -> Failed" << endl;
        else
                cout << " -> Done" << endl;

        // Waiting for Sync signal stabilization...
        cout << "Waiting for sync signal stabilization..." <<␣
↪endl;
        delay(3000);


        // Measurement
        //------------

        // Start target channel
        cout << "Start channel" << endl;
        if (HTDC_startChannel(iDev, TARGET_CH) != 0)
                cout << " -> Failed" << endl;
        else
                cout << " -> Done" << endl;

        // Recover the N shots and store data
        for (int i = 0; i < N_SHOT; i++)
        {
                cout << "One shot " << i << endl;
                if (TARGET_CH & 1) if (HTDC_
↪getCh1OneShotMeasurement(iDev, 0, 0, pHistoX[i], pHistoY[i], &
↪bw, &n) != 0) { cout << " Channel 1 : Failed to recover␣
↪OneShoot Data" << endl; }
                if (TARGET_CH & 2) if (HTDC_
↪getCh2OneShotMeasurement(iDev, 0, 0, pHistoX[i], pHistoY[i], &
↪bw, &n) != 0) { cout << " Channel 2 : Failed to recover␣
↪OneShoot Data" << endl; }
```

(continues on next page)

(continued from previous page)

```
                if (TARGET_CH & 4) if (HTDC_
↪getCh3OneShotMeasurement(iDev, 0, 0, pHistoX[i], pHistoY[i], &
↪bw, &n) != 0) { cout << " Channel 3 : Failed to recover␣
↪OneShoot Data" << endl; }
                if (TARGET_CH & 8) if (HTDC_
↪getCh4OneShotMeasurement(iDev, 0, 0, pHistoX[i], pHistoY[i], &
↪bw, &n) != 0) { cout << " Channel 4 : Failed to recover␣
↪OneShoot Data" << endl; }
                cout << " -> data recovered = " << n << endl;
                delay(MEAS_DEADTIME);
        }

        // Stop target channel
        cout << "Stop channel" << endl;
        if (HTDC_stopChannel(iDev, TARGET_CH) != 0)
                cout << " -> Failed" << endl;
        else
                cout << " -> Done" << endl;

        // Wait some time
        delay(2000);

        /*      CloseDevice function    */
        // Close initial device opened: MANDATORY AFTER EACH END␣
↪OF SYSTEM COMMUNICATION.
        if (HTDC_closeDevice(iDev) == 0) cout << "   ->␣
↪Communication closed" << endl;
        else cout << "  -> Failed to close communication" <<␣
↪endl;

        return 0;
}
```

CHAPTER 4

C++ Wrapper

4.1 Wrapper Advantage

A C++ wrapper has been created for several reasons.

- To make functions easy to use.
- To allow multiple HTDC device control in the same application and at the same time.
- To link Dynamic Library inside C++ code and not in the project configuration.

Note: Except OpenDevice function, you do not need to specify iDev when using HTDC wrapper function.

For example function `HTDC_getChannelState(short iDev, unsigned char iCh, int *status, unsigned long *nSampleToRecover, unsigned long *nSampleRecovered)` can be replace by `ObjectName.GetChannelState(unsigned char iCh, int *status, unsigned long *nSampleToRecover, unsigned long *nSampleRecovered)`

4.2 C++ code

Here is an example of how to use this wrapper to recover data from 2 HTDC :

```
#include <iostream>
using namespace std;

#include "HTDC_wrapper.h"
#include "HTDC.h"

// Select shared library compatible to current operating system
#ifdef _WIN32
#define DLL_PATH L"HTDC.dll"
#elif __unix
#define DLL_PATH "libHTDC.so"
#else
#define DLL_PATH "libHTDC.dylib"
#endif

int main(int argc, const char* argv[]) {
        short iDev = 0;
        short ret;
        char* devicesList[10];
        short numberDevices;
        unsigned short arg1[4], arg2[4];
        unsigned long ul_arg1[4], ul_arg2[4];

        HTDC_wrapper HTDC0(DLL_PATH);
        HTDC_wrapper HTDC1(DLL_PATH);

        /*    ListDevices function    */
        // List Aurea Technology devices: MANDATORY BEFORE EACH␣
↪OTHER ACTION ON THE SYSTEM
        if (HTDC0.ListDevices(devicesList, &numberDevices) == 0)
↪{
                if (numberDevices == 0) {
                        cout << endl << "    Please connect AT␣
↪device !" << endl << endl;
                        do {
                                delay(500);
                                HTDC0.ListDevices(devicesList, &
↪numberDevices);
                        } while (numberDevices == 0);
                }
        }

        // Open communication with device 0
        printf(" -%u: %s\n", 0, devicesList[0]);
```

(continues on next page)

(continued from previous page)

```
        HTDC0.OpenDevice(0);
        printf("\n HTDC %d-> Communication Open\n\n", 0);

        // Open communication with device 1
        printf(" -%u: %s\n", 1, devicesList[1]);
        HTDC1.OpenDevice(1);
        printf("\n HTDC %d-> Communication Open\n\n", 1);

        // Recover Channel 1 state from both HTDC
        HTDC0.GetChannelState(1, (int*)&arg1[0], &ul_arg1[0], &
↪ul_arg1[1]);
        printf("HTDC 1 -> Ch[1]: \n");
        printf("    + State   = %u \n", arg1[0]);
        printf("    + Consign = %lu \n", ul_arg1[0]);
        printf("    + Monitor = %lu \n", ul_arg1[1]);

        HTDC1.GetChannelState(1, (int*)&arg2[0], &ul_arg2[0], &
↪ul_arg2[1]);
        printf("HTDC 2 -> Ch[1]: \n");
        printf("    + State   = %u \n", arg2[0]);
        printf("    + Consign = %lu \n", ul_arg2[0]);
        printf("    + Monitor = %lu \n", ul_arg2[1]);

        // Wait some time
        delay(2000);

        /*    CloseDevice function    */
        // Close initial device opened: MANDATORY AFTER EACH END␣
↪OF SYSTEM COMMUNICATION.
        if (HTDC0.CloseDevice() == 0) cout << "   ->␣
↪Communication closed" << endl;
        else cout << "  -> Failed to close communication" <<␣
↪endl;
        if (HTDC1.CloseDevice() == 0) cout << "   ->␣
↪Communication closed" << endl;
        else cout << "  -> Failed to close communication" <<␣
↪endl;

        // Call class destructor
        HTDC0.~HTDC_wrapper();
        HTDC1.~HTDC_wrapper();

        return 0;
}
```

CHAPTER 5

All Functions

This section provides the prototypes and descriptions of all functions integrated into HTDC library.

Warning: More or less functions are available according to the device type. The compatibility depends on the device part number recovered by the "HTDC_getSystemVersion" function. Please see notes functions to check the compatibility with your device: -> Device compatibility: PN_HTDC_Mx_x

5.1 Library information

5.1.1 HTDC_getLibVersion

short **HTDC_getLibVersion**(unsigned short *value)

Get the librarie version.

Return the version librarie in format 0x0000MMmm

with: MM=major version

mm=minor version

Parameters ***value** – return lib version by pointer

Format: 0xMMmm

with: MM: major version

mm: minor version

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

5.2 Connection Functions

5.2.1 HTDC_listDevices

short **HTDC_listDevices**(char **devices, short *number)

List all the devices connected.

List all ChronoXea devices connected to the computer

Note: Mandatory to do before any other actions on the device.

Note: Device compatibility: all

Parameters

- ***devices** – pointer to the table buffer which contain list of devices connected

  Output format: “deviceName - serialNumber”

  Example:

  devices[0]=”ChronoXea - SN_12345678910”

  devices[1]=”ChronoXea - SN_10987654321”

- ***number** – pointer to the number of devices connected

Returns

0 : Function success

-1 : Function failed

5.2.2 HTDC_openDevice

short **HTDC_openDevice**(short iDev)
Open and initialize target device.

Open and initialize the target ChronoXea device

Note: Mandatory to used after "HTDC_listDevices" and before any other actions on the device.

Note: Device compatibility: all

Parameters **iDev** – Device index indicate by "HTDC_listDevices" function

Between 0 to n (indicated from "HTDC_listDevices" function)

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

5.2.3 HTDC_closeDevice

short **HTDC_closeDevice**(short iDev)
Close device.

Close ChronoXea device previously opened.

Note: Mandatory to do at the end of system control.

Note: Device compatibility: all

Parameters **iDev** – Device index indicate by "HTDC_listDevices" function

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.3 Device Information

5.3.1 HTDC_getSystemVersion

short **HTDC_getSystemVersion**(short iDev, char *version)

Get system version.

Get system version: Serial number, product number and firmware version

---

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" functions
- **version** – Pointer to the buffer which receive the system version.

  String format: "SN_xxxxxxxxxxx:PN_HTDC_Mx_xx:FN_x.xx"

  The receive buffer size must be of 64 octets min.

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.3.2 HTDC_getSystemFeature

short **HTDC_getSystemFeature**(short iDev, short iFeature, short *value)

Get system feature.

Read system memory to recover one system feature

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- **iFeature** – index feature to get

  0: channel number. Return: 0 (1 channel) to 3 (4 channels)

  1: integration mode. Return: 0 (standalone) or 1 (OEM)

- ***value** – value of the target feature

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.4 Set and Get HTDC Configuration

5.4.1 HTDC_setSyncSource

short **HTDC_setSyncSource**(short iDev, unsigned short mode)

Configure the Sync signal source.

Configure the HTDC sync clock source (start of HTDC) between internal or external

Note: Device compatibility: PN_HTDC_Mx_S

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- **mode** – source mode: 0=external, 1=internal

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.4.2 HTDC_getSyncSource

short **HTDC_getSyncSource**(short iDev, unsigned short *mode)

Configure the Sync signal source.

Configure the HTDC sync clock source (start of HTDC) between internal or external

Note: Device compatibility: PN_HTDC_Mx_S

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" functions
- ***mode** – return by pointer the source mode: 0=external, 1=internal

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.4.3 HTDC_setInternalSyncFrequency

short **HTDC_setInternalSyncFrequency**(short iDev, unsigned int value)

Set the internal sync frequency.

Adjust frequency of sync signal generated in internal from a synthesizer.

Note: This internal frequency is only applyed on HTDC sync input if sync source set in internal mode.

Note: Device compatibility: PN_HTDC_Mx_S

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- **value** – frequency value adjusting between 1Hz to 4MHz (by step of 4ns)

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.4.4 HTDC_getInternalSyncFrequency

short **HTDC_getInternalSyncFrequency**(short iDev, unsigned int *value)
Get the internal sync frequency.

Recover the current frequency of sync signal generated in internal from a synthesizer.

Note: Device compatibility: PN_HTDC_Mx_S

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- ***value** – return by pointer the frequency value. Between 1Hz to 4MHz (by step of 4ns)

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.4.5 HTDC_setSyncDivider

short **HTDC_setSyncDivider**(short iDev, unsigned short value)
Set the sync divider.

Adjust the divider applyed on the sync signal. As HTDC start input do

not exceed 4MHz, clock divider function is available to decrease the sync frequency.

Note: The divider is active for both the external and internal sync signal.

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- **value** – divider value between 1 to 1024 by step of 1.

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.4.6 HTDC_getSyncDivider

short **HTDC_getSyncDivider**(short iDev, unsigned short *value)
Get the sync divider.

Recover the current divider apply on the sync signal. As HTDC start input

do not exceed 4MHz, clock divider function is available to decrease the sync frequency.

Note: The divider is active for both the external and internal sync signal.

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- ***value** – return by pointer the divider value. Between 1 to 1024 by step of 1.

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.4.7 HTDC_setSyncInputConfig

short **HTDC_setSyncInputConfig**(short iDev, unsigned short enable, unsigned short edge, unsigned short level)

Configure Sync input.

Configure state (ON or OFF), trigger edge and level (TTL-CMOS or NIM) of sync input

---

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- **enable** – enable sync input: 0=OFF, 1=ON
- **edge** – edge of threshold signal: 0=rising, 1=falling
- **level** – level of input signal: 0=TTL-CMOS, 1=NIM

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.4.8 HTDC_getSyncInputConfig

short **HTDC_getSyncInputConfig**(short iDev, unsigned short *enable, unsigned short *edge, unsigned short *level)

Get Sync input configuration.

Get state (ON or OFF), trigger edge and level (TTL-CMOS or NIM) of sync input

---

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- ***enable** – return by pointer the sync input state. Return values: 0=OFF, 1=ON
- ***edge** – return by pointer the edge of threshold signal. Return values 0=rising, 1=falling

- ***level** – return by pointer the level of input. Return values: 0=TTL-CMOS, 1=NIM

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.4.9 HTDC_setResultFormat

short **HTDC_setResultFormat**(short iDev, unsigned char iCh, unsigned short mode)
Set result format.

Set target channel(s) raw data result in the prefered format mode.

User can set the data format recovered from "HTDC_getChXData" functions

either with HTDC raw data, raw and time tag or only with time tagging.

Depending on the format the features are different:

- "HTDC raw data" : 39bits of data time (~7s of measurement time range)
- "HTDC raw data + time tagging": 32bits of data time (55ms of range) + 32bits of tag data
- "time tagging" : 40bits of tag data

Note: By default the format is in HTDC raw data.

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- **iCh** – index of channel(s): CH_1, CH_2, CH3, CH4 (depends of channels number available)

  Parameter ORING with others channels id. (ex: HTDC_setResultFormat(CH_1|CH_2);)
- **mode** – format result:

  0: HTDC raw data

  1: HTDC raw data + time tagging

2: time tagging

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.4.10 HTDC_getResultFormat

short **HTDC_getResultFormat**(short iDev, unsigned char iCh, unsigned short *mode)

Get result format.

Get target channel raw data result in the prefered format mode.

Get the result data format return from "HTDC_getChXData" functions,

either with HTDC raw data, raw and time tag or only with time tagging.

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- **iCh** – index of channel(s): CH_1, CH_2, CH3, CH4 (depends of channels number available)
- **mode** – return by value the format result:

  0: HTDC raw data

  1: HTDC raw data + time tagging

  2: time tagging

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.5 Set and Get Channel Configuration

5.5.1 HTDC_setChannelDelay

short **HTDC_setChannelDelay**(short iDev, unsigned char iCh, float delay)
Set channel(s) delay.

Allows to control delay on specific(s) channel(s). This is a additional delay between 0 to 10.0ns by step of 10ps relative to the Sync input.

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- **iCh** – index of channel(s) to set: CH_1, CH_2, CH3, CH4 (depends of channels number available)

  Parameter ORING with others channels id. (ex: HTDC_setChannelDelay(CH_1|CH_2);)
- **delay** – delay in ns (between 0.0 to 10.0ns by step of 0.01ns)

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.5.2 HTDC_getChannelDelay

short **HTDC_getChannelDelay**(short iDev, unsigned char iCh, float *delay)
Get channel delay.

Allows to get the current delay applyed on the target channel.

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function

- **iCh** – index of channel(s) to set: CH_1, CH_2, CH3, CH4 (depends of channels number available)
- ***delay** – return by pointer the delay in ns (between 0.00 to 10.00ns by step of 0.01ns)

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.5.3 HTDC_setChannelConfig

short **HTDC_setChannelConfig**(short iDev, unsigned char iCh, unsigned short power, unsigned short edge, unsigned short level)

Set channel(s) configuration.

Allows to configurate specific(s) channel(s) to activate power, threshold edge and input level

---

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- **iCh** – index of channel(s) to set: CH_1, CH_2, CH3, CH4 (depends of channels number available)

  Parameter ORING with others channels id. (ex: HTDC_setChannelConfig(CH_1|CH_2);)
- **power** – channel power supply: 0=OFF or 1=ON
- **edge** – input signal edge sensibility: 0=rising or 1=falling
- **level** – level of input signal: 0=TTL-CMOS, 1=NIM

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.5.4 HTDC_getChannelConfig

short **HTDC_getChannelConfig**(short iDev, unsigned char iCh, unsigned short *power, unsigned short *edge, unsigned short *level)

Get channel configuration.

Allows to get the configuration of the target channel as the power state, edge and level input.

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- **iCh** – index of channel to set: CH_1, CH_2, CH3, CH4 (depends of channels number available)
- ***power** – return by pointer the actual state of the channel power supply: 0=OFF or 1=ON
- ***edge** – return by pointer the input signal edge sensibility: 0=rising or 1=falling
- ***level** – return by pointer the level of input. Return values: 0=TTL-CMOS, 1=NIM

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.5.5 HTDC_setDataRate

short **HTDC_setDataRate**(short iDev, unsigned char iCh, unsigned short value)

Set data rate of target channel(s) measurement.

Set target channel(s) data rate between 100 to 1000ms.

Allows to fix the MAX time before to recover data (kind of timeout). Means, if a big event flow is present on a HTDC input the data transfer between device and computer is done continuously but if the flow is lower as the buffer can stored data are sent to host at this data rate. By setting the rate, the user can thus recover the data more or less quickly using "HTDC_getChXData" or "HTDC_getChXHistogram" functions.

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- **iCh** – index of channel(s): CH_1, CH_2, CH3, CH4 (depends of channels number available)

  Parameter ORING with others channels id. (ex: HTDC_setDataRate(CH_1|CH_2);)
- **value** – data rate value in ms (between 100 to 1000ms, by step of 100ms)

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.5.6 HTDC_getDataRate

short **HTDC_getDataRate**(short iDev, unsigned char iCh, unsigned short *value)

Get data rate of a target channel.

Get current target channel data rate (between 100 to 1000ms). This value corresponds to the MAX time before to recover data (kind of timeout).

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- **iCh** – index of channel(s): CH_1, CH_2, CH3, CH4 (depends of channels number available)
- **value** – return by pointer the data rate value (between 100 to 1000ms, by step of 100ms)

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.5.7 HTDC_setMeasMode

short **HTDC_setMeasMode**(short iDev, unsigned char iCh, unsigned short mode)

Set the measurement mode of target channel(s)

Adjust target channel(s) measurement either in "Continuous" or "OneShot" mode. A channel in "Continuous" mode allows to run in free running as soon as the start command is applyed. A channel in "OneShot" mode allows to get one sample of measurement after the start command is send AND ONLY at each "HTDC_getChXOneShotMeasurement" function sending. By default, all channels measurement mode are in "Continuous" mode.

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- **iCh** – index of channel(s): CH_1, CH_2, CH3, CH4 (depends of channels number available)

  Parameter ORING with others channels id. (ex: HTDC_setMeasMode(CH_1|CH_2);)
- **mode** – measurement mode: 0=Continuous or 1=OneShot

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.5.8 HTDC_getMeasMode

short **HTDC_getMeasMode**(short iDev, unsigned char iCh, unsigned short *mode)

Get measurement mode of a target channel.

Get current target channel measurement mode (between "Continuous" or "OneShot" mode).

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- **iCh** – index of channel(s): CH_1, CH_2, CH3, CH4 (depends of channels number available)
- **mode** – return by pointer the curent measurement mode (0=Continuous or 1=OneShot)

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.6 Channel State

5.6.1 HTDC_armChannel

short **HTDC_armChannel**(short iDev, unsigned char iCh, long nSample, long time)

Arm target channel(s)

Arm target channel(s) to recover data either on a sample number or on a fix time.

Note: For a correct behavior, be careful to set ONLY ONE arming condition (the other parameter must be set to -1 (infinite))

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- **iCh** – index of channel to set: CH_1, CH_2, CH3, CH4 (depends of channels number available)

  Parameter ORING with others channels id. (ex: HTDC_armChannel(CH_1|CH_2);)
- **nSample** – number of sample time to recover.

  Values to set:

  -1=no condition (infinite)

x=between 1 to 2^31 by step of 1

- **time** – time of the acquisition in ms.

  Values to set:

  -1=no time condition (infinite)

  x=between 100 to 2^31 by step of 100ms

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.6.2 HTDC_startChannel

short **HTDC_startChannel**(short iDev, unsigned char iCh)

Start target channel(s)

Start the target channel(s) if it was previously armed.

Note: Be careful to arm the target channel(s) before to use this function

---

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- **iCh** – index of channel(s): CH_1, CH_2, CH3, CH4 (depends of channels number available)

  Parameter ORING with others channels id. (ex: HTDC_startChannel(CH_1|CH_2);)

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.6.3 HTDC_stopChannel

short **HTDC_stopChannel**(short iDev, unsigned char iCh)

Stop target channel(s)

Stop the target channel(s).

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- **iCh** – index of channel(s): CH_1, CH_2, CH3, CH4 (depends of channels number available)

  Parameter ORING with others channels id. (ex: HTDC_stopChannel(CH_1|CH_2);)

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.7 Monitoring Functions

5.7.1 HTDC_getChannelState

short **HTDC_getChannelState**(short iDev, unsigned char iCh, int *status, unsigned long *nSampleToRecover, unsigned long *nSampleRecovered)

Get channel state.

Return the state of the target channel: status, number of sample to recover and consign

---

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function

- **iCh** – index of channel to set: CH_1, CH_2, CH3, CH4 (depends of channels number available)
- ***status** – pointer to status of channel: 0=stopped, 1=armed or 2=running
- ***nSampleToRecover** – pointer to the number of sample to recover
- ***nSampleRecovered** – pointer to the number of sample recovered

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.7.2 HTDC_getEventsCounts

short **HTDC_getEventsCounts**(short iDev, int *nEvents, unsigned long count[])
Get events counts.

Allows to get events counts on each inputs. Every seconds values are upated.

This function can be polled to evaluate the current events apply on HTDC inputs.

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- **nEvents** – pointer to the events number(inputs)
- **count** – pointer to table which contains counting values of each inputs

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.7.3 HTDC_getCh1Data

short **HTDC_getCh1Data**(short iDev, unsigned long long *data, unsigned long *count)

Get channel 1 data.

Get channel 1 data in several format according to the mode set by the "HTDC_setResultFormat" function.

Thus the user can get only HTDC's raw data, time tagging or both.

Note: This function should be polled until the conditions, previously set from "HTDC_armChannel" function, are reached. The maximum pooling rate of this function depends of the data rate value set by "HTDC_setDataRate" function.

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- ***data** – pointer to the target buffer (in 64bits) to store data.

  Note: target buffer must be sized according to the number of data to recover.

  According to the result format mode set by HTDC_setResultFormat function the user can recover data with different formats:

  –

  in HTDC raw data (mode 0):

  Only HTDC's raw data are provided and given in multiple of HTDC_RES (0.013ns).

  Format in buffer:

  data[0]=0x000000RRRRRRRRRR

  data[1]=0x000000RRRRRRRRRR

  ...

  with:

  RRRRRRRRRR: 5 bytes of HTDC's raw data. Represent the time value in multiple of 0.013ns: time=data[x]*0.013

–

in HTDC raw data + time tagging (mode 1):

Both tds's raw data and time tagging.

Format in buffer:

data[0]=0xTTTTTTTTRRRRRRRR

data[1]=0xTTTTTTTTRRRRRRRR

...

with:

TTTTTTTT: 4 bytes (MSB) of time tagging value. Represent the number of Sync period until a ch1 event.

RRRRRRRR: 4 bytes (LSB) of HTDC's raw data. Represent the time value between the last Sync and the ch1 event. In multiple of 0.013ns: time=data[x]*0.013

–

in time tagging mode (mode 3):

Only time tagging value is returned.

Format in buffer:

data[0]=0x000000TTTTTTTTTT

data[1]=0x000000TTTTTTTTTT

...

with:

TTTTTTTTTT: 5 bytes of time tigging value. Represent the number of Sync period until a ch1 event

- **`*count`** – return by pointer the number of data recovered.

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.7.4 HTDC_getCh2Data

short **HTDC_getCh2Data**(short iDev, unsigned long long *data, unsigned long *count)

Get channel 2 data.

Get channel 2 data in several format according to the mode set by the "HTDC_setResultFormat" function.

Thus the user can get only HTDC's raw data, time tagging or both.

Note: This function should be polled until the conditions, previously set from "HTDC_armChannel" function, are reached. The maximum pooling rate of this function depends of the data rate value set by "HTDC_setDataRate" function.

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- ***data** – pointer to the target buffer (in 64bits) to store data.

  Note: target buffer must be sized according to the number of data to recover.

  According to the result format mode set by HTDC_setResultFormat function the user can recover data with different formats:

  –

  in HTDC raw data (mode 0):

  Only HTDC's raw data are provided and given in multiple of HTDC_RES (0.013ns).

  Format in buffer:

  data[0]=0x000000RRRRRRRRRR

  data[1]=0x000000RRRRRRRRRR

  ...

  with:

  RRRRRRRRRR: 5 bytes of HTDC's raw data. Represent the time value in multiple of 0.013ns: time=data[x]*0.013

–

in HTDC raw data + time tagging (mode 1):

Both tds's raw data and time tagging.

Format in buffer:

data[0]=0xTTTTTTTTRRRRRRRR

data[1]=0xTTTTTTTTRRRRRRRR

...

with:

TTTTTTTT: 4 bytes (MSB) of time tagging value. Represent the number of Sync period until a ch2 event.

RRRRRRRR: 4 bytes (LSB) of HTDC's raw data. Represent the time value between the last Sync and the ch2 event. In multiple of 0.013ns: time=data[x]*0.013

–

in time tagging mode (mode 3):

Only time tagging value is returned.

Format in buffer:

data[0]=0x000000TTTTTTTTTT

data[1]=0x000000TTTTTTTTTT

...

with:

TTTTTTTTTT: 5 bytes of time tigging value. Represent the number of Sync period until a ch2 event

- **`*count`** – return by pointer the number of data recovered.

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.7.5 HTDC_getCh3Data

short **HTDC_getCh3Data**(short iDev, unsigned long long *data, unsigned long *count)

Get channel 3 data.

Get channel 3 data in several format according to the mode set by the "HTDC_setResultFormat" function.

Thus the user can get only HTDC's raw data, time tagging or both.

Note: This function should be polled until the conditions, previously set from "HTDC_armChannel" function, are reached. The maximum pooling rate of this function depends of the data rate value set by "HTDC_setDataRate" function.

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- ***data** – pointer to the target buffer (in 64bits) to store data.

  Note: target buffer must be sized according to the number of data to recover.

  According to the result format mode set by HTDC_setResultFormat function the user can recover data with different formats:

  –

  in HTDC raw data (mode 0):

  Only HTDC's raw data are provided and given in multiple of HTDC_RES (0.013ns).

  Format in buffer:

  data[0]=0x000000RRRRRRRRRR

  data[1]=0x000000RRRRRRRRRR

  ...

  with:

  RRRRRRRRRR: 5 bytes of HTDC's raw data. Represent the time value in multiple of 0.013ns: time=data[x]*0.013

–

in HTDC raw data + time tagging (mode 1):

Both tds's raw data and time tagging.

Format in buffer:

data[0]=0xTTTTTTTTRRRRRRRR

data[1]=0xTTTTTTTTRRRRRRRR

...

with:

TTTTTTTT: 4 bytes (MSB) of time tagging value. Represent the number of Sync period until a ch3 event.

RRRRRRRR: 4 bytes (LSB) of HTDC's raw data. Represent the time value between the last Sync and the ch3 event. In multiple of 0.013ns: time=data[x]*0.013

–

in time tagging mode (mode 3):

Only time tagging value is returned.

Format in buffer:

data[0]=0x000000TTTTTTTTTT

data[1]=0x000000TTTTTTTTTT

...

with:

TTTTTTTTTT: 5 bytes of time tigging value. Represent the number of Sync period until a ch3 event

- **`*count`** – return by pointer the number of data recovered.

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.7.6 HTDC_getCh4Data

short **HTDC_getCh4Data**(short iDev, unsigned long long *data, unsigned long *count)

Get channel 4 data.

Get channel 4 data in several format according to the mode set by the "HTDC_setResultFormat" function.

Thus the user can get only HTDC's raw data, time tagging or both.

Note: This function should be polled until the conditions, previously set from "HTDC_armChannel" function, are reached. The maximum pooling rate of this function depends of the data rate value set by "HTDC_setDataRate" function.

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- ***data** – pointer to the target buffer (in 64bits) to store data.

  Note: target buffer must be sized according to the number of data to recover.

  According to the result format mode set by HTDC_setResultFormat function the user can recover data with different formats:

  –

  in HTDC raw data (mode 0):

  Only HTDC's raw data are provided and given in multiple of HTDC_RES (0.013ns).

  Format in buffer:

  data[0]=0x000000RRRRRRRRRR

  data[1]=0x000000RRRRRRRRRR

  ...

  with:

  RRRRRRRRRR: 5 bytes of HTDC's raw data. Represent the time value in multiple of 0.013ns: time=data[x]*0.013

–

in HTDC raw data + time tagging (mode 1):

Both tds's raw data and time tagging.

Format in buffer:

data[0]=0xTTTTTTTTRRRRRRRR

data[1]=0xTTTTTTTTRRRRRRRR

...

with:

TTTTTTTT: 4 bytes (MSB) of time tagging value. Represent the number of Sync period until a ch4 event.

RRRRRRRR: 4 bytes (LSB) of HTDC's raw data. Represent the time value between the last Sync and the ch4 event. In multiple of 0.013ns: time=data[x]*0.013

–

in time tagging mode (mode 3):

Only time tagging value is returned.

Format in buffer:

data[0]=0x000000TTTTTTTTTT

data[1]=0x000000TTTTTTTTTT

...

with:

TTTTTTTTTT: 5 bytes of time tigging value. Represent the number of Sync period until a ch4 event

- **`*count`** – return by pointer the number of data recovered.

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.7.7 HTDC_getCh1Histogram

short **HTDC_getCh1Histogram**(short iDev, double Xmin, double Xmax, double *histoX, double *histoY, double *binWidth, long *count)

Get channel 1 histogram.

Get the histogram of the time data measured from channel 1.

User can set the range of the histogram to recover or let the function adjust automatically the best range. As the histogram is represented on two axes of 65536 values, the bin width is automatically adjusted to respect this deep.

Note: This function should be polled until the conditions, previously set from "HTDC_armChannel" function, are reached. The maximum pooling rate of this function depends of the data rate value set by "HTDC_setDataRate" function.

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- **Xmin** – min value in ns of the histogram to recover.

  Range: 0 to 1000000000ns (1s)

  Note: if Xmin=0, the histogram is in auto scale mode.

  Means the function will adjust the min value according to the input signal measured.
- **Xmax** – max value in ns of the histogram to recover.

  Range: 0 to 1000000000ns (1s)

  Note: if Xmax=0, the histogram is in auto scale mode.

  Means the function will adjust the max value according to the input signal measured.
- ***histoX** – pointer to target buffer for X axis histogram values recovery.

  Each row represent the time value of bin.

  Note: the target buffer size must be of 65536.

- ***histoY** – pointer to target buffer for Y axis histogram values recovery.

  Each row represent the number of occurences of bin.

  Note: the target buffer size must be of 65536.

- ***binWidth** – return by pointer the value of binWidth in ns.

  If the range of the measured signal is upper 65536x13ps (851ns)

  the function automatically adjust the bin width to respect the

  Y axis deep of maximum 65536 values.

- ***count** – return by pointer the number of data recovered.

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.7.8 HTDC_getCh2Histogram

short **HTDC_getCh2Histogram**(short iDev, double Xmin, double Xmax, double *histoX, double *histoY, double *binWidth, long *count)

Get channel 2 histogram.

Get the histogram of the time data measured from channel 2.

User can set the range of the histogram to recover or let the function

adjust automatically the best range. As the histogram is represented

on two axes of 65536 values, the bin width is automatically adjusted to

respect this deep.

Note: This function should be polled until the conditions, previously set from "HTDC_armChannel" function, are reached. The maximum pooling rate of this function depends of the data rate value set by "HTDC_setDataRate" function.

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by “HTDC_listDevices” function
- **Xmin** – min value in ns of the histogram to recover.

  Range: 0 to 1000000000ns (1s)

  Note: if Xmin=0, the histogram is in auto scale mode.

  Means the function will adjust the min value according

  to the input signal measured.
- **Xmax** – max value in ns of the histogram to recover.

  Range: 0 to 1000000000ns (1s)

  Note: if Xmax=0, the histogram is in auto scale mode.

  Means the function will adjust the max value according

  to the input signal measured.
- ***histoX** – pointer to target buffer for X axis histogram values recovery.

  Each row represent the time value of bin.

  Note: the target buffer size must be of 65536.
- ***histoY** – pointer to target buffer for Y axis histogram values recovery.

  Each row represent the number of occurences of bin.

  Note: the target buffer size must be of 65536.
- ***binWidth** – return by pointer the value of binWidth in ns.

  If the range of the measured signal is upper 65536x13ps (851ns)

  the function automatically adjust the bin width to respect the

  Y axis deep of maximum 65536 values.
- ***count** – return by pointer the number of data recovered.

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.7.9 HTDC_getCh3Histogram

short **HTDC_getCh3Histogram**(short iDev, double Xmin, double Xmax, double *histoX, double *histoY, double *binWidth, long *count)

Get channel 3 histogram.

Get the histogram of the time data measured from channel 3.

User can set the range of the histogram to recover or let the function adjust automatically the best range. As the histogram is represented on two axes of 65536 values, the bin width is automatically adjusted to respect this deep.

Note: This function should be polled until the conditions, previously set from "HTDC_armChannel" function, are reached. The maximum pooling rate of this function depends of the data rate value set by "HTDC_setDataRate" function.

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- **Xmin** – min value in ns of the histogram to recover.

  Range: 0 to 1000000000ns (1s)

  Note: if Xmin=0, the histogram is in auto scale mode.

  Means the function will adjust the min value according to the input signal measured.
- **Xmax** – max value in ns of the histogram to recover.

  Range: 0 to 1000000000ns (1s)

  Note: if Xmax=0, the histogram is in auto scale mode.

  Means the function will adjust the max value according to the input signal measured.
- ***histoX** – pointer to target buffer for X axis histogram values recovery.

  Each row represent the time value of bin.

  Note: the target buffer size must be of 65536.

- ***histoY** – pointer to target buffer for Y axis histogram values recovery.

  Each row represent the number of occurences of bin.

  Note: the target buffer size must be of 65536.

- ***binWidth** – return by pointer the value of binWidth in ns.

  If the range of the measured signal is upper 65536x13ps (851ns)

  the function automatically adjust the bin width to respect the

  Y axis deep of maximum 65536 values.

- ***count** – return by pointer the number of data recovered.

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.7.10 HTDC_getCh4Histogram

short **HTDC_getCh4Histogram**(short iDev, double Xmin, double Xmax, double *histoX, double *histoY, double *binWidth, long *count)

Get channel 4 histogram.

Get the histogram of the time data measured from channel 4.

User can set the range of the histogram to recover or let the function

adjust automatically the best range. As the histogram is represented

on two axes of 65536 values, the bin width is automatically adjusted to

respect this deep.

Note: This function should be polled until the conditions, previously set from "HTDC_armChannel" function, are reached. The maximum pooling rate of this function depends of the data rate value set by "HTDC_setDataRate" function.

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by “HTDC_listDevices” function
- **Xmin** – min value in ns of the histogram to recover.

  Range: 0 to 1000000000ns (1s)

  Note: if Xmin=0, the histogram is in auto scale mode.

  Means the function will adjust the min value according

  to the input signal measured.
- **Xmax** – max value in ns of the histogram to recover.

  Range: 0 to 1000000000ns (1s)

  Note: if Xmax=0, the histogram is in auto scale mode.

  Means the function will adjust the max value according

  to the input signal measured.
- ***histoX** – pointer to target buffer for X axis histogram values recovery.

  Each row represent the time value of bin.

  Note: the target buffer size must be of 65536.
- ***histoY** – pointer to target buffer for Y axis histogram values recovery.

  Each row represent the number of occurences of bin.

  Note: the target buffer size must be of 65536.
- ***binWidth** – return by pointer the value of binWidth in ns.

  If the range of the measured signal is upper 65536x13ps (851ns)

  the function automatically adjust the bin width to respect the

  Y axis deep of maximum 65536 values.
- ***count** – return by pointer the number of data recovered.

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.7.11 HTDC_getCh1OneShotMeasurement

short **HTDC_getCh1OneShotMeasurement**(short iDev, double Xmin, double Xmax, double *histoX, double *histoY, double *binWidth, long *count)

Get channel 1 one shot histogram.

Get histogram of the data measured from channel 1 during the time condition adjusted with the "HTDC_armChannel" function. User can set the range of the histogram to recover or let the function adjust automatically the best range. As the histogram is represented on two axes of 65536 values, the bin width is automatically adjusted to respect this deep.

Respect the correct initialization and order of the functions:

- 1: Init the target channel
- 2: Set channel measurement mode in OneShot
- 3: Arm the channel with an infinite nSample (-1) and the desired measurement time
- 4: Start channel
- 5: Run one or more OneShot measurement when you want
- 6: Stop channel

Example of use:

- HTDC_setChannelConfig(CH_1, 1, 0, 0); // Init channel: power ON, rising edge, TTL-CMOS
- HTDC_setMeasMode(CH_1, 1); // Set channel measurement mode to OneShot
- HTDC_armChannel(CH_1, -1, 200); // Arm channel to recover data during 200ms, no sample number condition
- HTDC_startChannel(CH_1); // Start channel
- HTDC_getCh1OneShotMeasurement(0, 0, pHistoX, pHistoY, &bw); // Get one shot data sample
- HTDC_startChannel(CH_1); // Stop channel

Note: This function is a blocking function with a timeout of 30s.

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- **Xmin** – min value in ns of the histogram to recover.

  Range: 0 to 1000000000ns (1s)

  Note: if Xmin=0, the histogram is in auto scale mode.

  Means the function will adjust the min value according

  to the input signal measured.
- **Xmax** – max value in ns of the histogram to recover.

  Range: 0 to 1000000000ns (1s)

  Note: if Xmax=0, the histogram is in auto scale mode.

  Means the function will adjust the max value according

  to the input signal measured.
- ***histoX** – pointer to target buffer for X axis histogram values recovery.

  Each row represent the time value of bin.

  Note: the target buffer size must be of 65536.
- ***histoY** – pointer to target buffer for Y axis histogram values recovery.

  Each row represent the number of occurences of bin.

  Note: the target buffer size must be of 65536.
- ***binWidth** – return by pointer the value of binWidth in ns.

  If the range of the measured signal is upper 65536x13ps (851ns)

  the function automatically adjust the bin width to respect the

  Y axis deep of maximum 65536 values.
- ***count** – return by pointer the number of data recovered.

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.7.12 HTDC_getCh2OneShotMeasurement

short **HTDC_getCh2OneShotMeasurement**(short iDev, double Xmin, double Xmax, double *histoX, double *histoY, double *binWidth, long *count)

Get channel 2 one shot histogram.

Get histogram of the data measured from channel 2 during the time condition adjusted with the "HTDC_armChannel" function. User can set the range of the histogram to recover or let the function adjust automatically the best range. As the histogram is represented on two axes of 65536 values, the bin width is automatically adjusted to respect this deep.

Respect the correct initialization and order of the functions:

- 1: Init the target channel
- 2: Set channel measurement mode in OneShot
- 3: Arm the channel with an infinite nSample (-1) and the desired measurement time
- 4: Start channel
- 5: Run one or more OneShot measurement when you want
- 6: Stop channel

Example of use:

- HTDC_setChannelConfig(CH_2, 1, 0, 0); // Init channel: power ON, rising edge, TTL-CMOS
- HTDC_setMeasMode(CH_2, 1); // Set channel measurement mode to OneShot
- HTDC_armChannel(CH_2, -1, 200); // Arm channel to recover data during 200ms, no sample number condition
- HTDC_startChannel(CH_2); // Start channel
- HTDC_getCh2OneShotMeasurement(0, 0, pHistoX, pHistoY, &bw); // Get one shot data sample
- HTDC_startChannel(CH_2); // Stop channel

Note: This function is a blocking function with a timeout of 30s.

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- **Xmin** – min value in ns of the histogram to recover.

  Range: 0 to 1000000000ns (1s)

  Note: if Xmin=0, the histogram is in auto scale mode.

  Means the function will adjust the min value according

  to the input signal measured.
- **Xmax** – max value in ns of the histogram to recover.

  Range: 0 to 1000000000ns (1s)

  Note: if Xmax=0, the histogram is in auto scale mode.

  Means the function will adjust the max value according

  to the input signal measured.
- ***histoX** – pointer to target buffer for X axis histogram values recovery.

  Each row represent the time value of bin.

  Note: the target buffer size must be of 65536.
- ***histoY** – pointer to target buffer for Y axis histogram values recovery.

  Each row represent the number of occurences of bin.

  Note: the target buffer size must be of 65536.
- ***binWidth** – return by pointer the value of binWidth in ns.

  If the range of the measured signal is upper 65536x13ps (851ns)

  the function automatically adjust the bin width to respect the

  Y axis deep of maximum 65536 values.
- ***count** – return by pointer the number of data recovered.

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.7.13 HTDC_getCh3OneShotMeasurement

short **HTDC_getCh3OneShotMeasurement**(short iDev, double Xmin, double Xmax, double *histoX, double *histoY, double *binWidth, long *count)

Get channel 3 one shot histogram.

Get histogram of the data measured from channel 3 during the time condition adjusted with the "HTDC_armChannel" function. User can set the range of the histogram to recover or let the function adjust automatically the best range. As the histogram is represented on two axes of 65536 values, the bin width is automatically adjusted to respect this deep.

Respect the correct initialization and order of the functions:

- 1: Init the target channel
- 2: Set channel measurement mode in OneShot
- 3: Arm the channel with an infinite nSample (-1) and the desired measurement time
- 4: Start channel
- 5: Run one or more OneShot measurement when you want
- 6: Stop channel

Example of use:

- HTDC_setChannelConfig(CH_3, 1, 0, 0); // Init channel: power ON, rising edge, TTL-CMOS
- HTDC_setMeasMode(CH_3, 1); // Set channel measurement mode to OneShot
- HTDC_armChannel(CH_3, -1, 200); // Arm channel to recover data during 200ms, no sample number condition
- HTDC_startChannel(CH_3); // Start channel
- HTDC_getCh3OneShotMeasurement(0, 0, pHistoX, pHistoY, &bw); // Get one shot data sample
- HTDC_startChannel(CH_3); // Stop channel

Note: This function is a blocking function with a timeout of 30s.

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- **Xmin** – min value in ns of the histogram to recover.

  Range: 0 to 1000000000ns (1s)

  Note: if Xmin=0, the histogram is in auto scale mode.

  Means the function will adjust the min value according

  to the input signal measured.
- **Xmax** – max value in ns of the histogram to recover.

  Range: 0 to 1000000000ns (1s)

  Note: if Xmax=0, the histogram is in auto scale mode.

  Means the function will adjust the max value according

  to the input signal measured.
- ***histoX** – pointer to target buffer for X axis histogram values recovery.

  Each row represent the time value of bin.

  Note: the target buffer size must be of 65536.
- ***histoY** – pointer to target buffer for Y axis histogram values recovery.

  Each row represent the number of occurences of bin.

  Note: the target buffer size must be of 65536.
- ***binWidth** – return by pointer the value of binWidth in ns.

  If the range of the measured signal is upper 65536x13ps (851ns)

  the function automatically adjust the bin width to respect the

  Y axis deep of maximum 65536 values.
- ***count** – return by pointer the number of data recovered.

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.7.14 HTDC_getCh4OneShotMeasurement

short **HTDC_getCh4OneShotMeasurement**(short iDev, double Xmin, double Xmax, double *histoX, double *histoY, double *binWidth, long *count)

Get channel 4 one shot histogram.

Get histogram of the data measured from channel 4 during the time condition adjusted with the "HTDC_armChannel" function. User can set the range of the histogram to recover or let the function adjust automatically the best range. As the histogram is represented on two axes of 65536 values, the bin width is automatically adjusted to respect this deep.

Respect the correct initialization and order of the functions:

- 1: Init the target channel
- 2: Set channel measurement mode in OneShot
- 3: Arm the channel with an infinite nSample (-1) and the desired measurement time
- 4: Start channel
- 5: Run one or more OneShot measurement when you want
- 6: Stop channel

Example of use:

- HTDC_setChannelConfig(CH_4, 1, 0, 0); // Init channel: power ON, rising edge, TTL-CMOS
- HTDC_setMeasMode(CH_4, 1); // Set channel measurement mode to OneShot
- HTDC_armChannel(CH_4, -1, 200); // Arm channel to recover data during 200ms, no sample number condition
- HTDC_startChannel(CH_4); // Start channel
- HTDC_getCh4OneShotMeasurement(0, 0, pHistoX, pHistoY, &bw); // Get one shot data sample
- HTDC_startChannel(CH_4); // Stop channel

Note: This function is a blocking function with a timeout of 30s.

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- **Xmin** – min value in ns of the histogram to recover.

  Range: 0 to 1000000000ns (1s)

  Note: if Xmin=0, the histogram is in auto scale mode.

  Means the function will adjust the min value according

  to the input signal measured.
- **Xmax** – max value in ns of the histogram to recover.

  Range: 0 to 1000000000ns (1s)

  Note: if Xmax=0, the histogram is in auto scale mode.

  Means the function will adjust the max value according

  to the input signal measured.
- ***histoX** – pointer to target buffer for X axis histogram values recovery.

  Each row represent the time value of bin.

  Note: the target buffer size must be of 65536.
- ***histoY** – pointer to target buffer for Y axis histogram values recovery.

  Each row represent the number of occurences of bin.

  Note: the target buffer size must be of 65536.
- ***binWidth** – return by pointer the value of binWidth in ns.

  If the range of the measured signal is upper 65536x13ps (851ns)

  the function automatically adjust the bin width to respect the

  Y axis deep of maximum 65536 values.
- ***count** – return by pointer the number of data recovered.

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.8 Cross Correlation

5.8.1 HTDC_setCrossCorrelationALU

short **HTDC_setCrossCorrelationALU**(short iDev, unsigned short iMeas)

Set cross correlation ALU.

Set the ALU's measure type to apply the cross correlation.

Note: Be careful to select the correct type to have a positive result.

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by "HTDC_listDevices" function
- **iMeas** – Measure type available:

  0: CH1 - CH2

  1: CH2 - CH1

  2: CH2 - CH3

  3: CH3 - CH2

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

5.8.2 HTDC_getCrossCorrelationData

short **HTDC_getCrossCorrelationData**(short iDev, unsigned long long *data, unsigned long *count)

Get channel cross correlation data.

Get samples time measured from cross correlation.

Allows to recover only cross correlation data. To use this function, the "result format" of both target channels

MUST BE adjusted in “HTDC raw data + time tagging” mode. Respect the correct initialization and order of the functions:

- 1: Set result data format to “HTDC raw data + time tagging”
- 2: Init the 2 target channels
- 3: Set channels measurement mode in Continuous
- 4: Init ALU to calculate time between 2 channels
- 5: Arm target channels
- 6: Arm x cross channel (which do calcul)
- 7: Start target channels + x cross
- 8: Get data of the cross correlation
- 9: Stop channels + x cross

Example of use:

- HTDC_setResultFormat(CH_1|CH_2, 1); // Set result format to “HTDC raw data + time tagging”
- HTDC_setChannelConfig(CH_1|CH_2, 1, 0, 0); // Init channel: power ON, rising edge, TTL-CMOS
- HTDC_setMeasMode(CH_1|CH_2, 1); // Set channel measurement mode to OneShot
- HTDC_setCrossCorrelationALU(1); // Set ALU to calcul CH2 - CH1
- HTDC_armChannel(CH_1|CH_2, -1, -1); // Arm channels to recover in continuous
- HTDC_armChannel(CH_CROSS, 200, -1); // Arm x Cross to recover 200 sample
- HTDC_startChannel(CH_1|CH_2|CH_CROSS); // Start channels
- HTDC_getCrossCorrelationData(data); // Get result data
- HTDC_stopChannel(CH_1|CH_2|CH_CROSS); // Stop channels

Note: This function should be polled until the conditions, previously set from “HTDC_armChannel” function on each target channels, are reached. The maximum pooling rate of this function depends of the data rate value set by “HTDC_setDataRate” function.

Note: Device compatibility: all

Parameters

- **iDev** – Device index indicate by “HTDC_listDevices” function
- ***data** – pointer to the target buffer (in 64bits) to store data.

  Target buffer must be sized according to the number of data to recover.

  Each data are on raw data format and given in multiple of HTDC_RES (0.013ns).

  Format in buffer:

  data[0]=0x000000RRRRRRRRRR

  data[1]=0x000000RRRRRRRRRR

  ...

  with:

  RRRRRRRRRR: 5 bytes of HTDC’s raw data. Represent the time value in multiple of 0.013ns: time=data[x]*0.013
- ***count** – return by pointer the number of data recovered.

Returns

0 : Function success

-1 : Function failed

-2 : Parameter(s) error

-4 : iDev index Out Of Range

CHAPTER 6

Revision History

6.1 v2.0 (16/07/21)

- Add Device index in all functions
- Modify all get function to return value by pointer
- Add `TDC_` at the beginning of all functions
- Add Wrapper
- Handle multiple device application

6.2 v1.6 (15/03/21)

- **Improve functions**
  - TDC_armCh : allows inter-correlation using
- Add comments instruction for 'TDC_getCrossCorrelationData' function

6.3 v1.5 (26/02/21)

- **Improve function :**
  - TDC_getChxData : return correct n data according to consign
- Add comments on functions header

6.4 v1.4 (13/10/20)

- **Add function :**
  - TDC_getSystemVersion
  - TDC_getSystemFeature

6.5 v1.3 (01/06/20)

- **Modify functions :**
  - TDC_armChannel : add time condition
- **Add functions :**
  - TDC_setDataRate
  - TDC_getDataRate
  - TDC_setMeasMode
  - TDC_getMeasMode
  - TDC_getCh1OneShotMeasurement
  - TDC_getCh2OneShotMeasurement
  - TDC_getCh3OneShotMeasurement
  - TDC_getCh4OneShotMeasurement

6.6 v1.2 (24/04/20)

- **Rename functions :**
  - listDevices to TDC_listDevices
  - openDevice to TDC_openDevice
  - closeDevice to TDC_closeDevice
- **Modify functions :**
  - TDC_setSyncInputConfig

    - TDC_getSyncInputConfig
    - TDC_setChannelConfig
    - TDC_getChannelConfig
- **Add functions :**
    - TDC_setSyncSource
    - TDC_getSyncSource
    - TDC_setSyncDivider
    - TDC_getSyncDivider
    - TDC_setChannelDelay
    - TDC_getChannelDelay

6.7 v1.1 (14/11/19)

- **Add functions :**
    - TDC_setResultFormat
    - TDC_getResultFormat
    - TDC_setCrossCorrelationMode
    - TDC_getCrossCorrelationData
    - TDC_getCh1Histogram
    - TDC_getCh2Histogram
    - TDC_getCh3Histogram
    - TDC_getCh4Histogram

6.8 v1.0 (29/04/19)

- First release