IRIO LIBRARY-V1.2.0 USER's MANUAL

GETTING STARTED

INSTRUMENTATION AND APPLIED ACOUSTICS RESEARCH GROUP

GRUPO DE INVESTIGACIÓN EN
INSTRUMENTACIÓN Y
ACÚSTICA APLICADA

Worldwide Technical Support and Product Information

Web: www.i2a2.upm.es
Support: irio@i2a2.upm.es

I2A2 Research Group – Technical University of Madrid

UPM Campus Sur,
Carretera de Valencia, km 7, 28031 Madrid
Phone: +34 91 3364696
Fax: +34 91 3364696

Table of Contents

1 INTRODUCTION

1.1 Purpose and Scope

The IRIO library User's Manual contains information about using the IRIO library API for developing applications to support NI RIO devices, these are: cRIO and FlexRIO. This document presents a description about the possibilities that this C API interface layer permits, for managing these devices. This document does not cover technical details about how the device driver is implemented, and neither description about how NI RIO devices have to be "programmed". All the main details that the IRIO library user must know before starts using the library will be explained in the section 1.2. This information will help the user to understand correctly the following chapters, like the installation and verification procedures, and the use of the C API library provided.

Resource Description	Version	Public Release
IRIO Library	1.2.0	https://github.com/irio-i2a2/iriolib/releases

1.2 Overview

IRIO software library has been designed with the goal of simplifying the interface with RIO devices (compactRIO and FlexRIO). These devices are based on an FPGA which implies an infinite number of possible implementations depending on the FPGA project developer requirements. This means that the IRIO library user, hereinafter along this document "**the user"**, must know about how the RIO device has been configured, in order to implement correctly the software application. The user, will receive the FPGA *bitfile*, the file to be used by IRIO library to program the FPGA (RIO device), and the FPGA *header* file that contains the mapping addresses of the FPGA resources. Both files correspond to a specific RIO configuration.

In subsequent sections it will be explained how the device is configured and how the user will be able to know how to use the library according to it.

The first thing that the user should know is in which software layer, related to the hardware, is located this library (see Fig. 1).

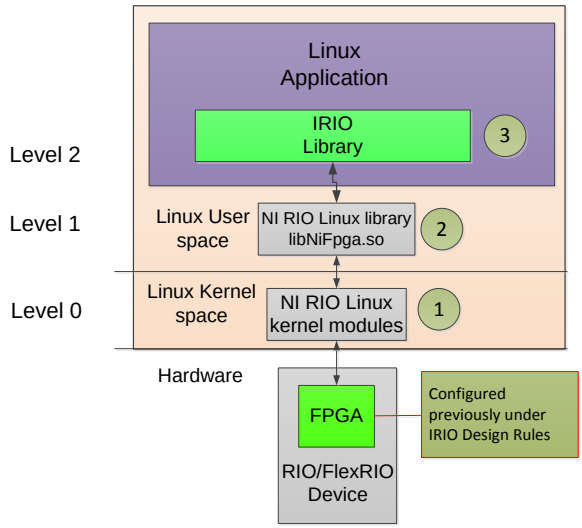

Fig. 1: Software layers used by IRIO Library

As usual Linux device driver implementations, NI RIO Linux device driver (Level 0), requires its corresponding user space library (Level 1) that permits the access to the RIO device. Finally IRIO library (Level 2) uses the NI RIO library to interface the user application with the hardware. The reasons because IRIO library has been added to the upper part of the software layer are:

- RIO devices are reconfigurable devices. This means that their hardware resources (I/O registers and data streams, DMAs), and the logic implemented depends on the FPGA project developer implementation. Every time there is a new FPGA configuration for a different purpose, hardware resources have different map addressing. IRIO library obtains from the FPGA *header* file all the resources mapping, creating a data structure to manage them dynamically, and loads the FPGA *bitfile* into the RIO device, configuring the FPGA in run time. **In the case of the NI RIO EPICS Device driver that uses this library, IRIO library avoids the recompilation of the NI RIO EPICS Device driver to support every new RIO configuration obtained following the IRIO design rules [RD2]**.
- RIO configuration can be oriented towards different purposes, with different methods or approaches for implementing data acquisition, real-time pre-processing etc. These type of "behaviours" have been defined as profiles. IRIO design rules document defines different types of profiles, according to support cRIO implementations, for different I/O modules, and different set of FlexRIO bundles, including data acquisition, image acquisition, with data transfer Device-To-Host, and Device-To-GPU. IRIO library provides a set of functions for achieving all these actions, and other ones, avoiding to the user to know how the RIO device has been configured. The user only requires the FPGA configuration, to know which FPGA resources can be used.

Fig. 2 depicts the use of IRIO library. The first step consists of the FPGA configuration using LabVIEW tools in a Windows platform. This is described in the IRIO Design Rules for LabVIEW-FPGA and is performed by the FPGA project developer attending to the fast

controller specifications. The results of this task are a FPGA *bitfile* and a FPGA *header* file. IRIO library requires these files to configure the FPGA in run time and to address the hardware resources respectively. Once the FPGA is configured IRIO library performs the different actions with the RIO devices.

IRIO library performs the initialization of the RIO device, as explained later, calling to the *irio_initDriver* function. This function prints out all the resources found in the FPGA according to the design rules.

The FPGA project developer has to provide a table (one example is shown in Table 10) containing all the resources used. These resources include the analog and digital I/O elements, the FIFO elements for moving data, and the list of all FPGA registers including the user defined auxiliary registers. The piece of LabVIEW code depicted in Fig. 2 exemplifies I/O operations through FPGA auxiliary registers. In this example FPGA developer requires a logic that acquires single analog input data values(ADC), and in case of overcoming a static threshold, an auxiliary Boolean output register will be latched to ON. The data acquisition period could be configured through an auxiliary input register, with a scalar value containing the microseconds to wait for the next iteration. After configuring the FPGA (step 1), to control and monitor this part of the FPGA the user , can program the auxiliary output register number 1 (defined as auxAO1) with the value, for instance, 1250 –this means a period of 1250µs (step 2). The user does not require knowing neither the mapping address nor the name of the LabVIEW controls and indicators (I/O registers) because IRIO library is doing this translation. The step 3 represents the user reading from the auxiliary digital input register 0 (auxDI0). In Fig. 2, green arrows represent direct (software-hardware) communication with the FPGA through NI-RIO Linux Device Driver. Red discontinuous arrows represent the relationship between the IRIO library calls and the terminals implemented with LabVIEW for FPGA.

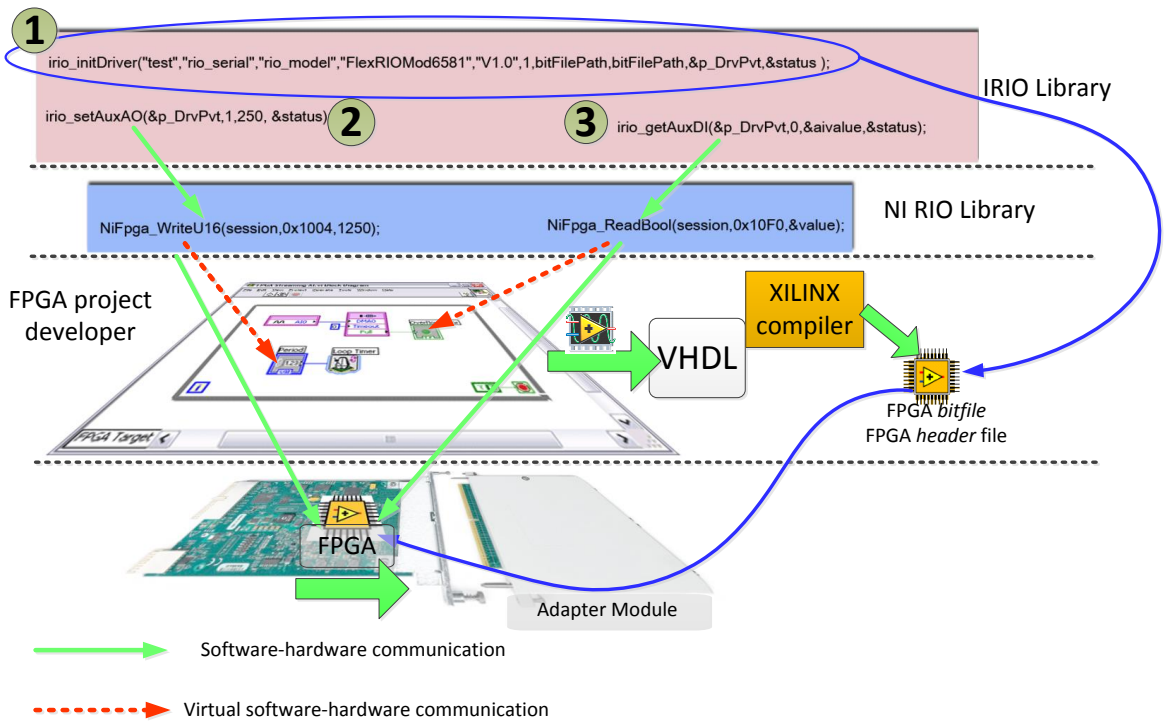

Fig. 2 Overview of the relationship between IRIO library calls, NI-RIO Linux device driver API and the implementation in the RIO device using LabVIEW tools

Fig. 3 represents the implementation of the aforementioned threshold detector example.

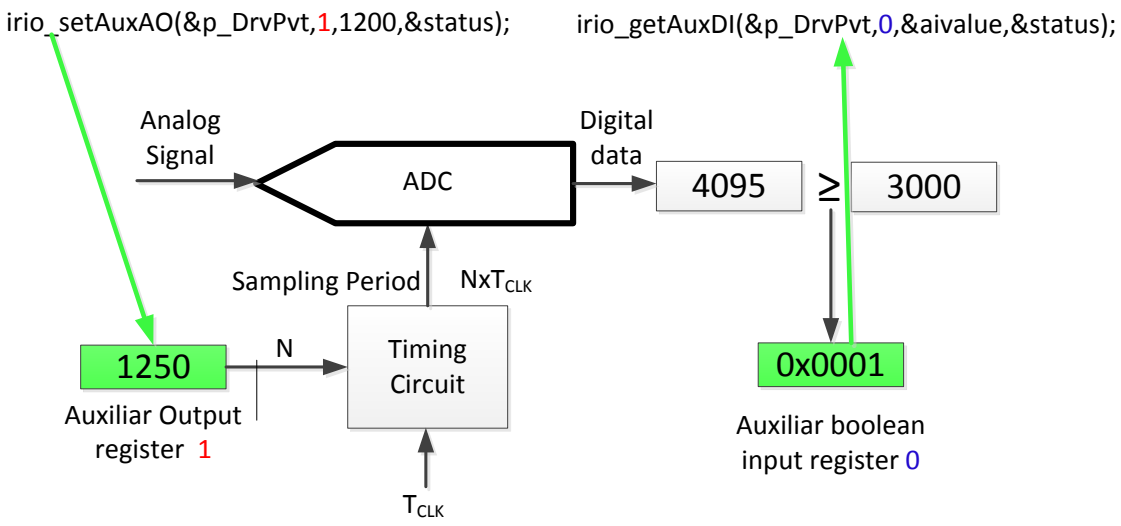

Fig. 3 Threshold detector implementation accessed by IRIO Library

It is absolutely necessary that the user has the information about the profile implemented in the RIO device, the list of all resources implemented, as explained in the design rules.

One of the examples of use of IRIO library is the NI-RIO EPICS device driver available in CCS that permits to create EPICS IOC applications. The different implementations using RIO devices do not require modifying the EPICS device support, only records database and st.cmd file configuration would be required. The same philosophy is applied for the implementation of a Nominal Device Support application using the IRIO library. These examples are shown in Fig. 4.

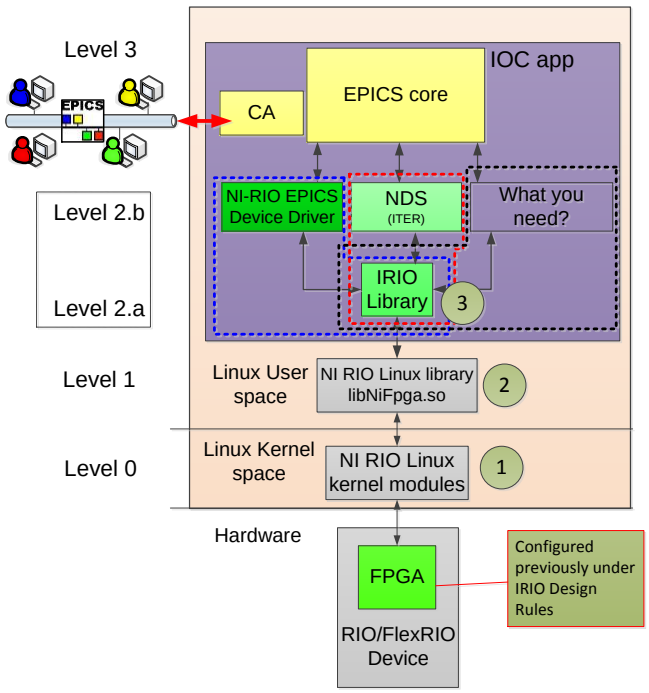

Fig. 4 IRIO library used for implementing an EPICS device support, with asyn, NDS, etc.

It is important to note that IRIO library is absolutely dependant on the NIRIO library.

1.3 Organization of the manual

This manual provides the basic information to use the IRIO Library API. The use of this library requires a basic knowledge of RIO devices. Section 2. Summarizes the RIO devices supported by the IRIO library. Section 3 Explains the development work flow required to complete a RIO application. Section 4 describes the installation process. Section 5 presents the key concepts of IRIO Library and finally section 6 describes some examples using the API.

1.4 Assumptions

Scientific Linux version 6.8 or higher or Red Hat Linux 6.5, both 64 bit architecture is assumed.

1.5 References

[RD1] IRIO EPICS Device Driver Traceability Matrix (TBC)

[RD2] IRIO Design Rules for LabVIEW for FPGA (TBC)

[RD3] NI LabVIEW High-Performance FPGA Developer's Guide (labview_high-perf_fpga_v1.1)

[RD4] IRIO Library API Reference (TBC)

1.6 Acronyms

ASIC	Application Specific Integrated Circuit
CLB	Configurable Logic Blocks
FAM	FlexRIO Adapter Module
FPGA	Field Programmable Gate Array
FlexRIO	Flexible Reconfigurable Input/Output
HDL	Hardware Description Language
HMI	Human-Machine Interface
IOC	Input / Output Controller
LUT	Look-Up Tables
OTP	OneTime Programmable
RAM	Random Access Memory
RIO	Reconfigurable Input/Output
cRIO	Compact Reconfigurable Input/Output
SRAM	Static RAM
TCP	Transmission Control Protocol
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

2 FLEXRIO AND CRIO PLATFORMS OVERVIEW

2.1 FlexRIO devices

FlexRIO devices consist of a FPGA and a set of adapter modules that provide high-performance analog and digital I/O. The adapter modules are interchangeable and define the I/O nodes in the LabVIEW FPGA programming environment.

NI FlexRIO FPGA modules feature, as seen in Table 1 **Error! Reference source not found.**, Xilinx Virtex-5 and Kintex-7 FPGAs, on-board dynamic RAM memory (DRAM), and the hardware interface to NI FlexRIO adapter modules that provide I/O to the FPGA. The adapter module interface consists of 132 lines of general-purpose digital I/O directly connected to FPGA pins, in addition to the power, clocking, and supplementary circuitry necessary to define the interface. Adapter modules are instantiated as a part of the LabVIEW project in a Component-Level Intellectual Property (CLIP) and the I/O interaction is provided by LabVIEW interfaces. Table 2**Error! Reference source not found.** shows the full range of adapter modules provided by NI.

Table 1: FlexRIO devices

Model	Bus	FPGA	FPGA Slices	FPGA DSP Slices	FPGA Memory (Block	Onboard Memory	Supported by IRIO Library V1.2.0
PXIe-7975R	PXIe	Kintex-7 XC7K410T	63,550	1,540	28,620 kbits	512 MB	✓
PXIe-7966R	PXIe	Virtex-5 SX95T -2	14,720	640	8,784 kbits	512 MB	✓
PXIe-7965R	PXIe	Virtex-5 SX95T	14,720	640	8,784 kbits	512 MB	✓
PXIe-7962R	PXIe	Virtex-5 SX50T	8,160	288	4,752 kbits	512 MB	✓
PXIe-7961R	PXIe	Virtex-5 SX50T	8,160	288	4,752 kbits	0 MB	✓
PXI-7954R	PXI	Virtex-5 LX110	17,280	64	4,608 kbits	128 MB	✓
PXI-7953R	PXI	Virtex-5 LX85	12,960	48	3,456 kbits	128 MB	✓
PXI-7952R	PXI	Virtex-5 LX50	7,200	48	1,728 kbits	128 MB	✓
PXI-7951R	PXI	Virtex-5 LX30	4,800	32	1,152 kbits	0 MB	✓

Table 2: FlexRIO adapter modules

Adapter module	Supported by IRIO Library V1.2.0
NI 5791 100 MHz Bandwidth RF Transceiver	Supported on demand
NI 5792 200 MHz Bandwidth RF Receiver	Supported on demand
NI 5793 200 MHz Bandwidth RF Transmitter	Supported on demand

Adapter module	Supported by IRIO Library V1.2.0
NI 5781 100 MS/s Baseband Transceiver	Supported on demand
NI 5782 250 MS/s IF Transceiver	Supported on demand
NI 5731 12-Bit, 40 MS/s, 2 Channel Digitizer NI 5732 14-Bit, 80 MS/s, 2 Channel Digitizer	Supported on demand
NI 5733 16-Bit, 120 MS/s, 2 Channel Digitizer NI 5734 16-Bit, 120 MS/s, 4 Channel Digitizer	Supported on demand
NI 5751 14-Bit, 50 MS/s,16 Channel Digitizer	Supported on demand
NI 5752 12-Bit, 50 MS/s, 32 Channel Digitizer	Supported on demand
NI 5761 14-bit, 250 MS/s, 4 Channel Digitizer	✓
NI 5762 16-Bit, 250 MS/s, 2 Channel Digitizer	Supported on demand
NI 5771 8-Bit, 3GS/s, 2 Channel Digitizer	Supported on demand
NI 5772 12-Bit, 1.6GS/s, 2-Channel Digitizer	Supported on demand
NI 5781 14-Bit, 100 MS/s,2 Channel Baseband Transceiver	✓
AT-1120 14-Bit, 2GS/s, 1-Channel Signal Generator	Supported on demand
AT-1212 14-Bit, 1.2GS/s, 2-Channel Signal Generator	Supported on demand
NI 6581 200 Mbit/s, 54 Channel, Single Ended Digital I/O	✓
NI 6583 300 Mbit/s, 32 SE and 16 LVDS Channel Digital I/O NI 6584 16 Mbit/s, 16 Ch, RS-422/RS-485 Digital I/O	Supported on demand
NI 6585 200 Mbit/s, 32 Channel, LVDS Digital I/O NI 6587 1 Gbit/s, 20 Channel, LVDS Digital I/O	Supported on demand
NI 1483 Full Configuration Camera Link	✓

IRIO Library provides functions to manage the following FlexRIO and adapter module setups. IRIO Library defines the set of a FlexRIO plus an adapter module as a bundle. Next paragraphs describe examples of these different bundles.

2.1.1 FlexRIO PXIe-7966R/NI5761

This bundle provides high speed analog data acquisition applications (see Fig. 5). Typical applications implementation for this device consist of:

- DMA channel containing samples acquired from the 4 analog inputs.
- Hardware logic implementing any kind of application using the 8 digital I/O

Apart from these features, depending on the features implemented in the FPGA, additional hardware processing algorithms can be integrated to analyse the data acquired in real time.

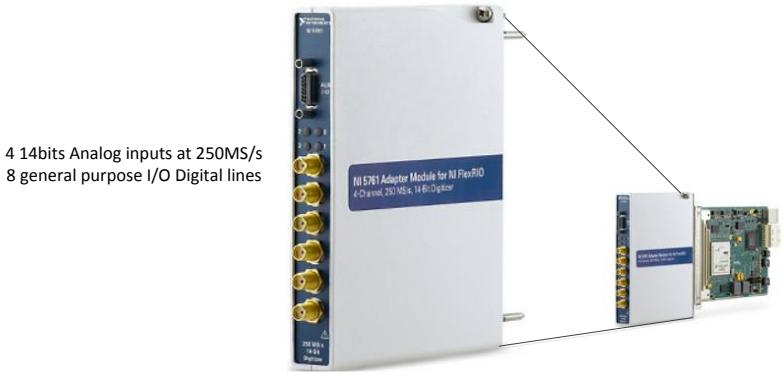

Fig. 5: Bundle PXIe7966R/NI5761

2.1.2 FlexRIO PXIe-7961R/NI6581

This bundle (see Fig. 6) provides the implementation of fast digital data acquisition and digital pattern generation applications.

Fig. 6: Bundle PXIe7961R/NI6581

Typical configuration for this device consist of:

- Fast hardware logic for digital I/O including control applications
- DMA channel containing group of digital samples acquired.

Apart from these features, depending on the features implemented in the FPGA, additional hardware processing algorithms can be integrated to analyse the data acquired in real time.

2.1.3 FlexRIO PXIe-7966R/NI1483

This bundle provides fast image data acquisition applications from cameralink cameras.

Fig. 7: Bundle PXIe7966R/NI1483

Typical implementations for this device consist of:

- Image acquisition system based in the standard CameraLink
- Hardware logic implementing any kind of application using the digital I/O.

Apart from these features, depending on the features implemented in the FPGA, additional hardware processing algorithms can be integrated to analyse the data acquired in real time.

2.2 Compact RIO

CompactRIO is a small, rugged RIO system for embedded and prototyping applications. It is configurable with four-and eight-slot backplanes. It contains two main components: a reconfigurable FPGA in a chassis, and the interchangeable industrial I/O modules. The CompactRIO system can be connected to a computer using a PCIe link. The embedded chassis contains the reconfigurable I/O FPGA chip directly connected to I/O modules that deliver diverse high-performance I/O capabilities.

For cRIO devices the IRIO library supports:

Table 3 cRIO devices supported by IRIO library

HW ID	Description
cRIO 9159	CompactRIO Chassis with Virtex 5 FPGA
cRIO module NI9205	Analog input Module
cRIO module NI9264	Analog output Module
cRIO module 9401	TTL Digital I/O
cRIO module 9477	Digital Outputs 60V sinking
cRIO module 9476	Digital Outputs 24V sourcing
cRIO module 9425	Digital Input 24V sinking

HW ID	Description
cRIO module 9426	Digital Input 24V sourcing

Fig. 8: 9159 chassis and I/O modules

3 DEVELOPMENT WORKFLOW WITH IRIO LIBRARY

3.1 Development cycle

The development cycle to be followed in order to build applications using IRIO Library requires two major steps. The former is the hardware implementation in the FPGA and the later the development of the software using IRIO Library. Fig. 9 summarizes all the steps needed to program a RIO device with the binary image supporting the FPGA functionality (bitfile). Previous to implement any application for RIO devices using the IRIO library in ScientificLinux, this aforementioned **bitfile** and **header** file obtained using LabVIEW tools are needed.

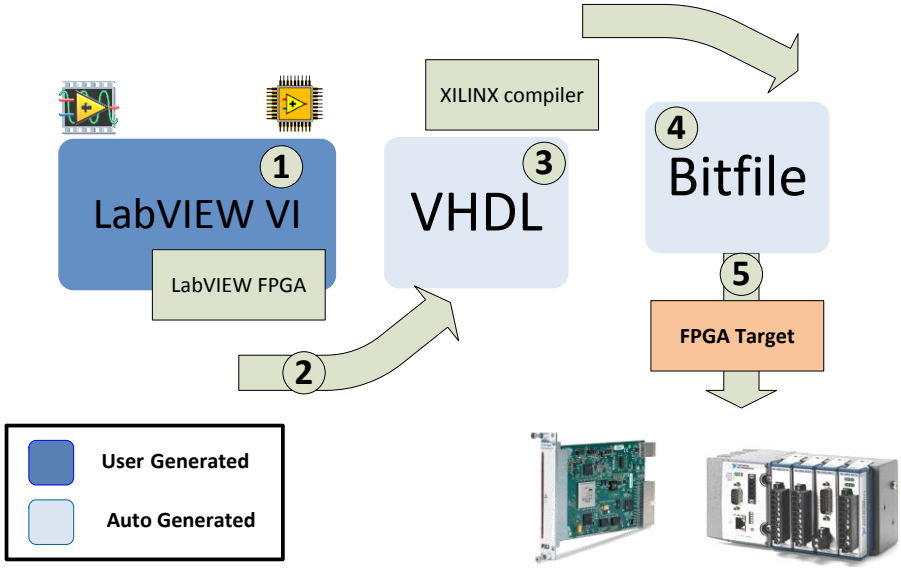

Fig. 9 Developing applications with LabVIEW FPGA for RIO

The complete development workflow to be followed in order to implement applications for RIO devices using IRIO Library and developed with LabVIEW for FPGA is summarized in Fig. 10. This graphic presents the steps to be done using LabVIEW for FPGA in a development environment using a Windows computer with all the software tools installed. Once the FPGA design matches with the expected design, the corresponding bitfile and header files required by the IRIO library are generated. The FPGA project developer should additionally report a table with all the FPGA resources used in order to provide the final user enough information to use the RIO device with IRIO Library. The implementation of software applications using IRIO Library are explained in detail in section 5 and 6.

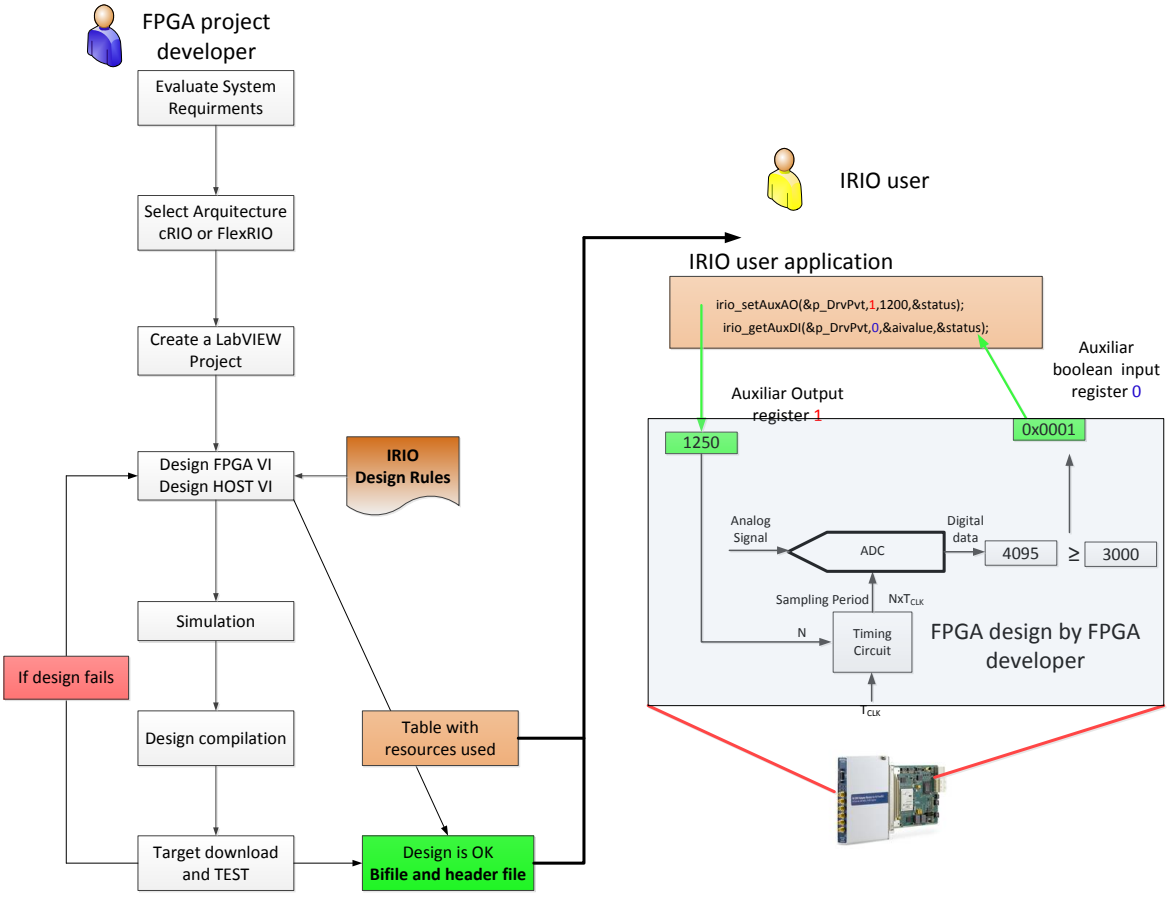

Fig. 10: Development cycle for RIO applications

4 IRIO LIBRARY INSTALLATION

This chapter explains: the IRIO Library software release information and history, the Software/Hardware environment, how to install the required software, how to verify that the software is installed correctly and how to uninstall it.

4.1 IRIO Library Release vs. Linux Release

Table 4: IRIO Library vs. Linux release

IRIO Library Version	Release Date	IRIO Library user manual version	NI-RIO Linux Device Driver	Linux Version supported
1.2.0	Jan-2017	1.2.0	15.0.0	ScientificLinux6.8 RHEL 6.5

4.2 IRIO library Release Notes

Table 5: IRIO Library release description

Version	Release Description
1.2.0	• IRIO Library supports its installation over ScientificLinux 6.8 and RHEL 6.5
1.1.2	• Added missing “extern C”. • Improvements in error and initialization messages. • Fixed DMA GPU errors due to the search of unnecessary resources. • Data and image acquisition methods are now structured in the same way. • Typo corrected in cRIO_DAQDMA.c. • Problem with signal generator solved in FlexRIO_mod5761-7966.c
1.1.1	• Added conversion factors in irioDriver and identification of the module. • Minor modifications in 5761 c test to adapt it to 5761 AC coupling. • Minor modifications in Imaq-GPU c test.
1.1.0	• Examples improved supporting input parameters • Example for performance measurement. • New API function to retrieve driver version

Version	Release Description
1.0.0	• Support for cRIO and FlexRIO • The API provides the following functionalities: • Download the bitfile to the FPGA • Identification of the resources found in the FPGA if they are defined as described in the design rules document. • Profiles identification and data structure initialization according to these profiles. • Setters and getters for performing input/output operations in the FPGA

4.3 Software and hardware environment

The table below identifies the hardware, software, and other resources needed to install and run the software.

Table 6: IRIO Library Software and Hardware environment

Type	Description	Manufacture	Version
Software	Scientific Linux	Fermilab/CERN/DESY/ETHZ.	6.8
Software	EPICS Base	Open Source. Many contributors	3.15.4
Software	AsynDriver	APS/CARS: Mark Rivers	R4-30
Software	Busy	APS/AOD/BCDA: Tim Mooney	R1.6.1
Software	Sscan	APS/AOD/BCDA: Tim Mooney	R2-10
Software	Autosave	APS/AOD/BCDA: Tim Mooney	R5-7-1
Software	Calc	APS/AES/BCDA: Tim Mooney	R3-6-1
Software	IRIO library	I2A2-UPM/ITER	1.2.0
Software	NI-RIO Linux Device Driver	National Instruments	15.0.0
Hardware	PC (HDD > 10G, RAM > 2G)	N/A	N/A
Hardware	FlexRIO card with adapter module or cRIO chassis with I/O Modules	National Instruments	N/A

4.4 Installing the IRIO library

This section describes how to install IRIO library and how to check its correct installation.

NOTE: NI-RIO Linux Driver 15.0 + EPICS Base + SynApps must be previously installed. Click here for more information.

4.4.1 Obtaining the source code

IRIO Library software is hosted under GitHub, a web-based Git repository hosting service.

Download IRIO Library software from: https://github.com/irio-i2a2/iriolib/releases

4.4.2 Installing IRIO Library

1. Untar *iriolib-v1.2.0 in /opt/epics/support*
2. Navigate to */opt/epics/support/iriolib-v1.2.0* and execute:

```
$ sudo make
$ sudo make install
```

4.5 Verification of the IRIO library installation

The IRIO library is installed under /usr/local folder.

Verify that next folders exist and include the following content:

- /usr/local/include/iriolib-v1.2.0

```
├── irioDataTypes.h
├── irioDriver.h
├── irioError.h
├── irioHandlerAnalog.h
├── irioHandlerDigital.h
├── irioHandlerDMAGPU.h
├── irioHandlerDMA.h
├── irioHandlerImage.h
├── irioHandlerSG.h
└── irioResourceFinder.h
```

- /usr/local/src/iriolib-v1.2.0

```
├── irioDriver.c
├── irioError.c
├── irioHandlerAnalog.c
├── irioHandlerDigital.c
├── irioHandlerDMA.c
├── irioHandlerDMAGPU.c
├── irioHandlerImage.c
├── irioHandlerSG.c
├── irioResourceFinder.c
```

- /usr/local/lib/iriolib-v1.2.0

```
├── libirioCore.a
├── libirioCore.so
└── obj
    ├── irioDriver.o
    ├── irioError.o
    ├── irioHandlerAnalog.o
    ├── irioHandlerDigital.o
    ├── irioHandlerDMA.o
    ├── irioHandlerImage.o
    ├── irioHandlerSG.o
    └── irioResourceFinder.o
```

- /usr/local/irio_examples

```
├── cRIO_DAQDMA.c
├── cRIO_IO.c
├── FlexRIO_mod1483-Image-CPU.c
├── FlexRIO_mod1483-Image-GPU.c
├── FlexRIO_mod1483-uart.c
├── FlexRIO_mod5734.c
├── FlexRIO_mod5761.c
├── FlexRIO_mod5761-perf.c
├── FlexRIO_mod6581.c
├── FlexRIO_noModule.c
├── FlexRIO_onlyResources.c
├── Makefile
├── resourceFail
│   ├── 7961
│   │   ├── NiFpga_FlexRIOonlyResources_7961.h
│   │   └── NiFpga_FlexRIOonlyResources_7961.lvbitx
│   ├── 7965
│   │   ├── NiFpga_FlexRIOonlyResources_7965.h
│   │   └── NiFpga_FlexRIOonlyResources_7965.lvbitx
│   └── 7966
│       ├── NiFpga_FlexRIOonlyResources_7966.h
│       └── NiFpga_FlexRIOonlyResources_7966.lvbitx
└── resourceTest
    ├── 7952
    │   ├── NiFpga_FlexRIO_CPUDAQ.h
    │   ├── NiFpga_FlexRIO_CPUDAQ.lvbitx
    │   ├── NiFpga_FlexRIO_CPUIMAQ.h
    │   ├── NiFpga_FlexRIO_CPUIMAQ.lvbitx
    │   ├── NiFpga_FlexRIO_GPUDAQ.h
    │   ├── NiFpga_FlexRIO_GPUDAQ.lvbitx
    │   ├── NiFpga_FlexRIO_GPUIMAQ.h
    │   └── NiFpga_FlexRIO_GPUIMAQ.lvbitx
    ├── 7961
    │   ├── NiFpga_FlexRIO_CPUDAQ_7961.h
    │   ├── NiFpga_FlexRIO_CPUDAQ_7961.lvbitx
    │   ├── NiFpga_FlexRIO_CPUIMAQ_7961.h
    │   ├── NiFpga_FlexRIO_CPUIMAQ_7961.lvbitx
    │   ├── NiFpga_FlexRIO_GPUDAQ_7961.h
    │   ├── NiFpga_FlexRIO_GPUDAQ_7961.lvbitx
    │   ├── NiFpga_FlexRIO_GPUIMAQ_7961.h
    │   ├── NiFpga_FlexRIO_GPUIMAQ_7961.lvbitx
    │   ├── NiFpga_FlexRIOMod6581_7961.h
    │   ├── NiFpga_FlexRIOMod6581_7961.lvbitx
    │   ├── NiFpga_FlexRIOnoModule_7961.h
    │   ├── NiFpga_FlexRIOnoModule_7961.lvbitx
```

```
    │   ├── NiFpga_FlexRIOonlyResources_7961.h
    │   └── NiFpga_FlexRIOonlyResources_7961.lvbitx
    ├── 7965
    │   ├── NiFpga_FlexRIO_CPUDAQ_7965.h
    │   ├── NiFpga_FlexRIO_CPUDAQ_7965.lvbitx
    │   ├── NiFpga_FlexRIO_CPUDAQ.h
    │   ├── NiFpga_FlexRIO_CPUDAQ.lvbitx
    │   ├── NiFpga_FlexRIO_CPUIMAQ_7965.h
    │   ├── NiFpga_FlexRIO_CPUIMAQ_7965.lvbitx
    │   ├── NiFpga_FlexRIO_CPUIMAQ.h
    │   ├── NiFpga_FlexRIO_CPUIMAQ.lvbitx
    │   ├── NiFpga_FlexRIO_GPUDAQ_7965.h
    │   ├── NiFpga_FlexRIO_GPUDAQ_7965.lvbitx
    │   ├── NiFpga_FlexRIO_GPUDAQ.h
    │   ├── NiFpga_FlexRIO_GPUDAQ.lvbitx
    │   ├── NiFpga_FlexRIO_GPUIMAQ_7965.h
    │   ├── NiFpga_FlexRIO_GPUIMAQ_7965.lvbitx
    │   ├── NiFpga_FlexRIO_GPUIMAQ.h
    │   ├── NiFpga_FlexRIO_GPUIMAQ.lvbitx
    │   ├── NiFpga_FlexRIOMod1483_7965.h
    │   ├── NiFpga_FlexRIOMod1483_7965.lvbitx
    │   ├── NiFpga_FlexRIOMod5761_7965.h
    │   ├── NiFpga_FlexRIOMod5761_7965.lvbitx
    │   ├── NiFpga_FlexRIOMod6581_7965.h
    │   ├── NiFpga_FlexRIOMod6581_7965.lvbitx
    │   ├── NiFpga_FlexRIOnoModule_7965.h
    │   ├── NiFpga_FlexRIOnoModule_7965.lvbitx
    │   ├── NiFpga_FlexRIOonlyResources_7965.h
    │   └── NiFpga_FlexRIOonlyResources_7965.lvbitx
    ├── 7966
    │   ├── NiFpga_FlexRIO_CPUDAQ_7966.h
    │   ├── NiFpga_FlexRIO_CPUDAQ_7966.lvbitx
    │   ├── NiFpga_FlexRIO_CPUIMAQ_7966.h
    │   ├── NiFpga_FlexRIO_CPUIMAQ_7966.lvbitx
    │   ├── NiFpga_FlexRIO_GPUDAQ_7966.h
    │   ├── NiFpga_FlexRIO_GPUDAQ_7966.lvbitx
    │   ├── NiFpga_FlexRIO_GPUIMAQ_7966.h
    │   ├── NiFpga_FlexRIO_GPUIMAQ_7966.lvbitx
    │   ├── NiFpga_FlexRIOMod1483_7966.h
    │   ├── NiFpga_FlexRIOMod1483_7966.lvbitx
    │   ├── NiFpga_FlexRIOMod5734_7966.h
    │   ├── NiFpga_FlexRIOMod5734_7966.lvbitx
    │   ├── NiFpga_FlexRIOMod5761_7966.h
    │   ├── NiFpga_FlexRIOMod5761_7966.lvbitx
    │   ├── NiFpga_FlexRIOnoModule_7966.h
    │   ├── NiFpga_FlexRIOnoModule_7966.lvbitx
    │   ├── NiFpga_FlexRIOonlyResources_7966.h
    │   ├── NiFpga_FlexRIOonlyResources_7966.lvbitx
    │   ├── NiFpga_FlexRIO_perf_7966.h
    │   ├── NiFpga_FlexRIO_perf_7966.lvbitx
    │   ├── NiFpga_FPGA1483_8tap8GPUV1_0.h
    │   └── NiFpga_FPGA1483_8tap8GPUV1_0.lvbitx
    └── 9159
        ├── NiFpga_cRIODAQDMA_9159.h
        ├── NiFpga_cRIODAQDMA_9159.lvbitx
        ├── NiFpga_cRIOIO_9159.h
        └── NiFpga_cRIOIO_9159.lvbitx
```

After the library is installed, execute the following command to list the RIO devices available.

```
$ lsni -v
```

The output will be similar to this depending on the configuration of your PC.

```
[ebernal@localhost ~]$ lsni -v
Scanning localhost for devices...

System Configuration API Experts found:
NI-RIO 15.0.0 (ni-rio)
NI Network Browser 15.0 (network)
NIFLEXRIO 15.0.0 (niflexrio)
NI System Configuration 15.0 (nisyscfg)

System Configuration API resources found:
RIO0
--Primary Expert:       NI-RIO 15.0.0
--Model Name:           NI 9159
--Serial Number:        019ED079
--Bus/Dev/Func:         6/8/0

RIO1
--Primary Expert:       NI-RIO 15.0.0
--Model Name:           NI PXIe-7966R
--Serial Number:        01A34CC7
--Bus/Dev/Func:         18/0/0

RIO2
--Primary Expert:       NI-RIO 15.0.0
--Model Name:           NI PXIe-7966R
--Serial Number:        0177A2AD
--Bus/Dev/Func:         19/0/0
```

4.6 Uninstalling the IRIO library

The user needs super user permission.

1. Uninstall the software executing next commands:

```
$ cd /opt/epics/support/IRIOLIB-1.2.0
$ sudo make clean uninstall
```

2. Verify that next folders do not exist:
   - /usr/local/include/iriolib-v1.2.0
   - /usr/local/src/iriolib-v1.2.0
   - /usr/local/lib/iriolib-v1.2.0
   - /usr/local/irio_examples

5 IRIO LIBRARY FUNCTIONALITY

This chapter contains information about the library key concepts. It is important to understand how the FPGA resources can be used by other applications using this library.

The API provides the following functionalities:

- Download the bitfile to the FPGA
- Identification of the resources found in the FPGA if they are defined as described in the design rules document [RD2].
- Profiles identification and data structure initialization according to these profiles.
- Setters and getters for performing input/output operations in the FPGA

5.1.1 FPGA resources.

The implementation of FPGA code has to meet the design rules described in [RD2]. That document describes the different profiles available to be implemented for the user in cRIO and FlexRIO technology. Once the design has been compiled and the bitfile have been tested using LabVIEW, the FPGA application is ready to be moved to Linux environment to be used in a standalone C/C++ application or in an EPICS application.

The header file and the bitfile obtained with C API generator have to be copied to the corresponding folder depending of the application type.

Depending on the profile implemented resources are organized in:

1. Common resources
2. Specific profile resources
3. Optional Resources.

The resources in the FPGA are identified using the specific labels described in [RD2].

5.1.2 IRIO library basic use.

The use of IRIO library requires following a sequence of specific steps [RD4]. These steps are depicted in Fig. 11 and can be described as:

- Initialization of IRIO driver. This is accomplished with the irio_initDriver function. This function performs all the library initialization. The main operations implemented in the function are: download the bitfile to the FPGA, resources identification depending on the profile implemented in the FPGA and data structure initialization for later use by other API calls. This function returns a structure of type irioDrv_t that contains the mapping of all resources found in the FPGA.
- Use of getters and setters. The applications using the IRIO library can access to read/write the terminal available in the FPGA in order to know the initialization value or to set this. All getters/setters use a similar prototype with these parameters in common:
    - irioDrv_t* p_DrvPvt, this is data structure for the RIO device
    - TStatus* status, this is data structure containing the status and errors obtained once the function has been executed.
- If the application is using DMA (for image acquisition profiles or DAQ acquisition profile) the user needs to configure the DMA using irio_setupDMAstoHost.
- Execution of the hardware implemented in the FPGA. In the initialization the bitfile is downloaded but it is not executed. This means that the FPGA is waiting until the software send a run command. This run command is executed with the irio_setFPGAstart.

- Once the FPGA is running the application the user can interact with the FPGA terminals using the setters/getters and can trigger data acquisition retrieving the data using the irio_getDMATtoHost. There are many more functions explained in the Library API part of this document.
- For closing the driver and release the resources, irio_closeDriver is required to be invoked.

The relationship between the IRIO library calls and NI-RIO Linux Device Driver API is described in the document **Error! Reference source not found.**

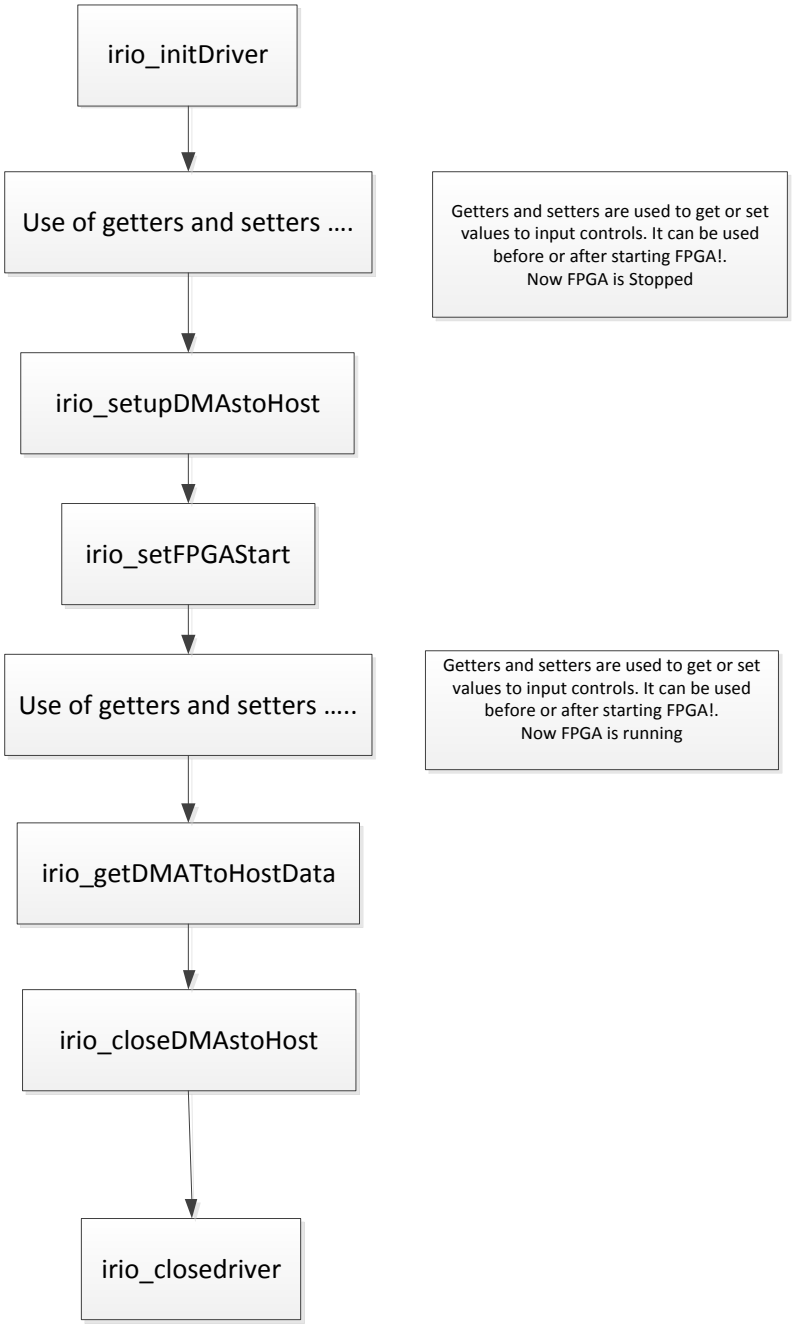

Fig. 11: Basic steps to use IRIO library

6 USE EXAMPLES OF IRIO LIBRARY

The software unit contains an examples folder with different examples to understand the use of IRIO library. These examples are listed in Table 7. The column with "LabVIEW template used" allows the user to check the LabVIEW template used for the implementation. The examples are located in **/examples**.

Table 7: List of examples provided in the software unit

Name	Utility	LabVIEW template used
cRIO_DAQDMA.c	Example of cRIO DAQ profile in compactRIO platform	cRIO9159_DAQDMA
cRIO_IO.c	Example of cRIO Point by Point profile in compactRIO platform	cRIO9159_IO
FlexRIO_mod1483-Image.c	Example of NI1483 adapter module for FlexRIO platform	FlexRIO1483_8T8
FlexRIO_mod1483-uart.c	Example of NI1483 adapter module for FlexRIO platform. Use of serial line	FlexRIO1483_8T8
FlexRIO_mod5761-Perf.c	Example of NI5761 adapter module for FlexRIO platform. Performance measurement	FlexRIO5761
FlexRIO_mod5761.c	Example of NI5761 adapter module for FlexRIO platform	FlexRIO5761
FlexRIO_mod6581.c	Example of NI6581 adapter module for FlexRIO platform	FlexRIO6581
FlexRIO_noModule.c	Example of FlexRIO card without adapter module	FlexRIOnoModule
FlexRIO_onlyResources.c	Example for testing the resources in the FPGA	FlexRIOonlyResources

Next paragraphs present the details of the different calls to IRIO Library in two examples, one for compactRIO platform and other for FlexRIO.

6.1 cRIO example for data acquisition using point by point profile.

6.1.1 Functionality

The functionality implemented is this example can be summarized in:

- Data acquisition from three analog input channels of the NI9205 (AI0, AI1 and AI2). The sampling rate can range from mHz to 1kHz. If the debug mode is activate the values acquired are 1.25v, 3.125v and -4.75v respectively.
- Analog output on AO0, AO1 and AO2 in NI9264. If AOEnable for the different channels is false the output is 0v.
- Digital output on DO0 (channel 0 in 9401), DO1 (DO0 in 9477), and DO2 (DO0 in 9476).
- Digital input from DI0 (DI04 from NI9401), DI1 (DI0 from NI9477) and DI2 (DI0 from 9425).
- Two auxiliary register auxAO0 and auxAO1 connected to auxAI0 and auxAI1.
- Two auxiliary binary variables auxDO0 and auxDO1 connected to auxDI0 and auxDI1.

This example is using the hardware configuration depicted in Fig. 12. The example uses analog input from NI9205(AI0,AI1,AI2), three analog outputs from NI9264 (AO0,AO1,AO2), two digital lines in NI9401, one digital line in NI9477, NI9476, NI9426 and NI9425. The interconnection among the modules is presented in Table 8

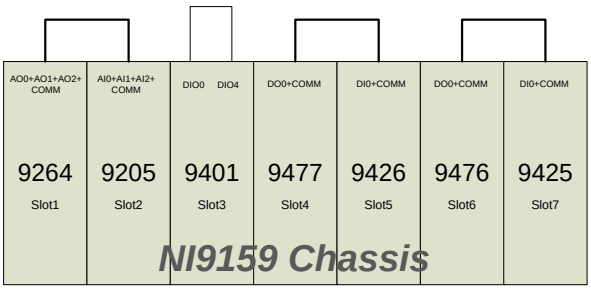

Fig. 12: cRIO modules allocation for PBP example

Table 8: Interconnection of signals for the example

Interconnection of channels from modules of the NI9159 Chassis	
NI9264 AO0	NI9205 AI0
NI9264 AO1	NI9205 AI1
NI9264 AO2	NI9205 AI2
NI9264 COMM	NI9205 COMM
NI9401 DIO0	NI9401 DIO4
NI9477 DO0	NI9426 DI0

Interconnection of channels from modules of the NI9159 Chassis	
NI9477 COMM	Power Supply COMM
NI9426 Vsup	Power Supply Vcc
NI9476 DO0	NI9425 DI0
NI9476 Vsup	Power Supply Vcc
NI9425 COMM	Power Supply COMM

6.1.2 Resources implemented in the FPGA

The profile implemented in this cRIO platform is the PBP (Point by Point). The resources are listed below in Table 9.

Table 9: Resources implemented in the PBP profile

Terminal Name	Data type	Type	Detail	Information	Default Value	Initialized before run?
Platform	U8	Indicator	This terminal defines the form factor used in the FPGA implementation	Mandatory	0- cRIO	YES
Common Terminals for cRIO						
FPGAVIversion	Array U8	Indicator	Contains the VI version, 2 elements. One for MM major version, and the next one mm minor version. MM.mm	Mandatory	FPGAVIVersion[0]=1 FPGAVIVersion[1]=1	YES
InitDone	Boolean	Indicator	This terminal must be set to true when the FPGA is initialized	Mandatory	False=NOK	NO
InsertedOModulesID	Array U16	Indicator	Numeric array of values indicating the c-Modules IDs detected	Mandatory	Defined by NI	NO
cRIOModulesOK	Boolean	Indicator	I/O Modules correctly detected	Mandatory		NO

Terminal Name	Data type	Type	Detail	Information	Default Value	Initialized before run?
Fref	U32	Indicator	Contains the Reference clock of the FPGA for sampling rate	Mandatory	1000	YES
DevQualityStatus	U8	Indicator	This indicator will show the status of the acquisition	Mandatory	0	NO
DevTemp	I16	Indicator	This indicator will show the temperature of the FPGA	Mandatory	0	NO
Devprofile	U8	Indicator	This indicator defines the implementation in the FPGA (DAQ, Image, etc)	Mandatory	1	YES
DebugMode	Boolean	Control	If debug is true the FPGA will simulate the acquired data. Otherwise, physical signals are acquired	Mandatory	False	NO
DAQStartStop	Boolean	Control	This terminal must be set to true to start data acquisition	Mandatory	False	NO
Specific Terminals for Point by Point acquisition profile						
SamplingRate<n>	U16	Control	Integer number obtained as Sampling rate/Fref	Mandatory	n ={0 .. 2}	NO
Optional Resources for Point by Point profile						
AI0 AI1 AI2	I32	Indicator	Digital sample	Optional	0	NO
auxAI0 auxAI1	I32	Indicator	Auxiliary internal FPGA variables	Optional	0	NO
AO0 AO1 AO2	I32	Indicator	Digital Sample	Optional	0	NO

Terminal Name	Data type	Type	Detail	Information	Default Value	Initialized before run?
auxAO0 auxAO1	I32	Control	Auxiliary internal FPGA variables	Optional	0	NO
AOEnable0 AOEnable1 AOEnable2	Boolean	Control	Enable or Disable analog output		False	NO
DO0 DO1 DO2	Boolean	Control	Digital line	Optional	False	NO
auxDO0 auxDO1	Boolean	Control	Digital line	Optional	False	NO
DI0 DI1 DI2	Boolean	Indicator	Digital line	Optional	False	
auxDI0 auxDI1	Boolean	Indicator	Digital line	Optional	False	
SGNo	U8	Control	Number of waveform generators	Optional	0	YES

6.1.3 Source code description

The example using the resources explained before is named cRIO_IO.c. Next an explanation of the use of the IRIO library is presented. A program using the IRIO Library must contain the header files with data types definition and prototypes shown below. The initialization of the library and the IRIO device is done with the call **irio_initDriver.** Before calling this function some variables have to defined and initialized.

```
/*************************************************/
/*              INCLUDES REQUIRED                */
/*************************************************/
#include <irioDriver.h>
#include <irioDataTypes.h>
#include <irioHandlerAnalog.h>
#include <irioHandlerDigital.h>
#include <irioHandlerDMA.h>
#include <irioHandlerImage.h>
#include <irioHandlerSG.h>
#include <irioResourceFinder.h>
/*************************************

/*************************************************/
/*              VARIABLES REQUIRED               */
/*************************************************/

irioDrv_t p_DrvPvt; // Device data structure
TIRIOStatusCode st=IRIO_success; //status
TStatus status; //
status.msg=NULL; //concatenated error messages
status.code=IRIO_success; //status
```

```
st|=irio_initDriver("test","0x019ED079","NI
9159","cRIOIO_9159","V1.1",1,bitFilePath,bitFilePath,&p_DrvPvt,&status );
//INPUT OUTPUT parameters description
//@param[in] appCallID      Name for this irioDriver session.
//@param[in] DeviceSerialNumber S/n of the RIO target device
//@param[in] RIODeviceModel Model number of the RIO target device
//@param[in] projectName    Part of the name of the bitfile to be
downloaded into the FPGA
//@param[in] FPGAversion    Version of the bitfile (e.g. [1,1] for v1.1).
Must match with the value of the FPGA register
//@param[in] verbosity      Indicates whether or not should the driver
print trace messages.
//@param[in] headerDirPath where to search for the header file
corresponding to the bitfile to be downloaded
//@param[in] bitfile  Path where to search for the bitfile to be
downloaded
//@param[out] p_DrvPvt      Pointer to the driver structure. Will be
initialized with the resources found.
//@param[out] status        Warning and error messages produced during
the execution of this call will be added here.
//@return \ref TIRIOStatusCode result of the execution of this call.
```

irio_initDriver downloads the bitfile in the FPGA, identifies all the resources implemented and returns the implemented resources in a data structure (p_DrvPvt) for later access by other API calls. Once the FPGA is configured, it is set to *execution state* calling irio_setFPGAStart with *value* parameter set to 1. The execution of this function implies checking correct location of I/O modules and the completion of the initialization steps defined by the hardware designer (LabVIEW for FPGA designer).

```
st|=irio_setFPGAStart(&p_DrvPvt,1,&status);
//@param[in] p_DrvPvt Pointer to the driver session structure
//@param[in] value    0=Do nothing, 1=Start FPGA if not already started
//@param[out] status  Warning and error messages produced during the
execution of this call will be added here.
//@return \ref TIRIOStatusCode result of the execution of this call.
```

Once the FPGA is running (the hardware is performing its function as it has been designed), the user can call the different API function in order to use the hardware. In this example the data acquisition is started calling this function:

```
st|=irio_setDAQStartStop(&p_DrvPvt,1,&status);
```

This call checks if data acquisition is started

```
st|=irio_getDAQStartStop(&p_DrvPvt,&aivalue,&status);
```

This function gets the VI version downloaded in the FPGA:

```
st|=irio_getFPGAVIVersion(&p_DrvPvt,&value, 2, &valueLength,&status);
```

This function gets the temperature of the RIO device:

```
st|=irio_getDevTemp(&p_DrvPvt,&aivalue,&status);
```

This function reads the profile implemented in the RIO device:

```
st|=irio_getDevProfifle(&p_DrvPvt,&aivalue,&status);
```

This function enables and disables the debug mode of the RIO FPGA.

```
st|=irio_setDebugMode(&p_DrvPvt,0,&status);
```

This piece of code reads the value of three analog inputs two times. The first reading from AI0, AI1 and AI2, is done with debug mode set to OFF. In this case we are reading the physical voltage acquired by ADC. Remember that AO0,AO1 and AO2 in NI9264 are connected to the three analog input in NI9205. The second time, the reading from AI0, AI1 and AI2 is done with debug mode set to ON. In this case we are reading simulated values as explained before. The values read from RIO devices are binary values. If you want to convert this to voltage you can use the CVADC filed available in p_DrvPvt struct. This floating point value contains the conversion value for this adapter module.

```
// With Debug mode set to OFF
st|=irio_getAI(&p_DrvPvt,0,&aivalue,&status);
printf("Slot 2 9205 AI0 Value %f \n", aivalue*p_DrvPvt.CVADC);
…
st|=irio_getAI(&p_DrvPvt,1,&aivalue,&status);

st|=irio_getAI(&p_DrvPvt,2,&aivalue,&status);

// Now Debug mode is set to ON
st|=irio_setDebugMode(&p_DrvPvt,1,&status);

st|=irio_getAI(&p_DrvPvt,0,&aivalue,&status);
…
st|=irio_getAI(&p_DrvPvt,1,&aivalue,&status);

st|=irio_getAI(&p_DrvPvt,2,&aivalue,&status);
```

The following piece of code shows the use of analog output functions. The value that are written to the FPGA for the analog outputs are binary values. The conversion between floating point values and binary values has to be done using CVDAC constant available in the p_DrvPvt struct. The binary value is obtained multiplying the floating point by this constant.

```
// Debug mode is set to OFF
st|=irio_setDebugMode(&p_DrvPvt,0,&status);

st|=irio_setAO(&p_DrvPvt,0,4000, &status);//AO0 configuration
usleep(15);
//4000/p_DrvPvt.CVDAC
st|=irio_setAOEnable(&p_DrvPvt,0,NiFpga_True, &status);//AO0 activation
usleep(15);

st|=irio_getAI(&p_DrvPvt,0, &aivalue, &status);//AI0 reading
usleep(15);
//The same for the AO1 and AI1

st|=irio_setAO(&p_DrvPvt,1,8000, &status);
usleep(15);
```

```
st|=irio_setAOEnable(&p_DrvPvt,1,NiFpga_True, &status);
usleep(15);

st|=irio_getAI(&p_DrvPvt,1, &aivalue, &status);
usleep(15);
//For reading the configured values use these functions

st|=irio_getAOEnable(&p_DrvPvt,0,&aivalue,&status);
usleep(15);
st|=irio_getAOEnable(&p_DrvPvt,1,&aivalue,&status);

st|=irio_getAO(&p_DrvPvt,0,&aivalue, &status);
usleep(15);
st|=irio_getAO(&p_DrvPvt,1,&aivalue, &status);
```

Finally, the program shows how to use digital I/O functions.

```
st|=irio_setDO(&p_DrvPvt,0,1, &status);
usleep(15);
irio_getDI(&p_DrvPvt,0,&aivalue, &status);
usleep(15);

//For reading value configured
st|=irio_getDO(&p_DrvPvt,0,&aivalue, &status);
usleep(15);

//For reading and writing from/to auxiliar I/O
st|=irio_setAuxAO(&p_DrvPvt,0,5000, &status);
usleep(15);
st|=irio_getAuxAO(&p_DrvPvt,0,&aivalue,&status);
usleep(15);
st|=irio_getAuxAI(&p_DrvPvt,0,&aivalue,&status);
usleep(15);

st|=irio_setAuxDO(&p_DrvPvt,0,1, &status);
usleep(15);
st|=irio_getAuxDO(&p_DrvPvt,0,&aivalue,&status);
usleep(15);
st|=irio_getAuxDI(&p_DrvPvt,0,&aivalue,&status);
usleep(15);

//For stopping the acquisition
st|=irio_setDAQStartStop(&p_DrvPvt,0,&status);
```

The last step for closing the communication with the RIO devices and freeing the resources is done with this call.

```
//for closing session with cRIO device
st|=irio_closeDriver(&p_DrvPvt, &status);
```

NOTE: Every time an IRIO function is used, the status of the function execution can be checked as it is shown below:

```
st|=irio_setDAQStartStop(&p_DrvPvt,0,&status);
if(St!=0)
{
    printf("[ERROR] %s\n", status.msg);fflush(stdout);
```

```
    free(status.msg);status.msg=NULL;status.code=IRIO_success;
    return 0;
}
```

6.2 FlexRIO example for data acquisition using PXIe7966R and NI5761

6.2.1 Functionality

This example is using a FlexRIO PXIe7966R with a NI5761 adapter module. The example is implemented the data acquisition of 4 channels. CH0 and CH1 are analog input channels connected to signals generators and CH2 and CH3 are internally connected in the FPGA to two simulated signals generators. The result of this is two real signals and two simulated signals. The profile implemented in this FlexRIO platform is the data acquisition one. The user can control the data acquisition sampling rate and can interact with the FPGA parameters to configure the simulated signal generators. This implementation also include analog and digital register in the FPGA (auxiliary analog and digital) that can be used for different purposes by the user.

6.2.2 Resources implemented in the FPGA

The table summarizes the resources implemented in the FPGA. Terminals are identified as they are defined in the design rules.

Table 10 FlexRIO PXIe-7966R + NI5761 adapter module configuration

Terminal Name	Data type	Type	Detail	Information	Default Value	Initialize Before Run?
Platform	U8	Indicator	This terminal defines the form factor used in the FPGA implementation	Mandatory	FlexRIO	YES
Common Terminals for FlexRIO						
FPGAVIversion	Array U8	Indicator	Contains the VI version, 2 elements. One for MM major version, and the next one mm minor version. MM.mm	Mandatory	FPGAVIVersion[0]=1 FPGAVIVersion[1]=0	YES
InitDone	Boolean	Indicator	This terminal must be set to true when the FPGA is initialized	Mandatory	False	N/A
RIOAdapterCorrect	Boolean	Indicator	Boolean indicating if the adapter module is the correct for the application	Mandatory	True	NO
InsertedIOModuleID	U32	Indicator	Contains the Module ID of the corresponding mdouel	Mandatory	0	NO

Terminal Name	Data type	Type	Detail	Information	Default Value	Initialize Before Run?
Fref	U32	Indicator	Contains the Reference clock of the FPGA for sampling rate	Mandatory	100M	YES
DevQualityStatus	U8	Indicator	This indicator will show the status of the acquisition	Mandatory	0	NO
DevTemp	I16	Indicator	This indicator will show the temperature of the FPGA	Mandatory	0	NO
Devprofile	U8	Indicator	This indicator defines the implementation in the FPGA (DAQ, Image, etc)	Mandatory	0	NO
DAQStartStop	Boolean	Control	This terminal must be set to true to start data acquisition	Mandatory	False	NO
Specific Terminals for data acquisition profile						
DMATtoHOSTNCh	U16, array	Indicator	Describes the number of DMAs implemented in the FPGA. The array must be initializes with the number of channels available in each DMA.	Mandatory	Array size=1 DMATtoHOSTNCh[0]=4	YES
DMATtoHOSTFrameType	U8, array	Indicator	Describes the frame type used in the DMA frame	Mandatory	Array size=1 DMATtoHOSTFrameType[0]=0	YES
DMATtoHOSTSampleSize	U8, array	Indicator	Size in bytes for the channel sample	Mandatory	Array size=1 DMATtoHOSTSampleSize[0]=2	YES
DMATtoHOSTBlockNWords	U16, array	Indicator	Length of the block used for each DMA		Array size=1 DMATtoHOSTBlockNWords[0]=4096	YES
DebugMode	Boolean	Control	If debug is true the FPGA will simulate the acquired data. Otherwise, physical signals are acquired	Mandatory	False	

Terminal Name	Data type	Type	Detail	Information	Default Value	Initialize Before Run?
DMATtoHOST0	FIFO	DMA to HOST	FIFO memory in the FPGA	Mandatory		N/A
DMATtoHOSTSamplingRat0	U16	Control	Integer number obtained as Sampling rate/Fref	Mandatory	10000	NO
DMATtoHOSTEnable0	Boolean	Control	Enable or disable write to DMA FIFO	Mandatory	False	YES
DMATtoHOSTOverflows	U16	Indicator	Status of the different DMA FIFO	Mandatory	0	YES
Optional Resources						
auxAI0 auxAI1 auxAI2 auxAI3 auxAI4 auxAI5 auxAI9 auxAI10	I32	Indicator	Auxiliary internal FPGA variables	Optional		NO
auxAO0 auxAO1 auxAO2 auxAO3 auxAO4 auxAO5	I16	Control	Auxiliary internal FPGA variables Connected to auxAI<n>	Optional	0	NO
auxDI0 auxDI1 auxDI2 auxDI3 auxDI4 auxDI5	Boolean	Indicator	Internal input line	Optional		NO
auxDO0 auxDO1 auxDO2 auxDO3 auxDO4 auxDO5	Boolean	Control	Internal FPGA lines connected to auxDI<n>	Optional	False	NO
SGNo	U8	Control	Number of waveform generators	Optional	2	YES

Terminal Name	Data type	Type	Detail	Information	Default Value	Initialize Before Run?
SGSignalType0	U8	Control	Signal shape to be generated	Optional	0	YES
SGAmp0	U32	Control	DSS accumulator increment	Optional	0	YES
SGFreq0	U32	Control	Phase control	Optional	100000	YES
SGPhase0	U32	Control	Phase control	Optional	0	YES
SGUpdateRate0	U32	Control	Update rate	Optional	40	YES
SGFref0	U32	Indicator	Reference frequency	Optional	100000000	YES
SGSignalType1	U8	Control	Signal shape to be generated	Optional	0	YES
SGAmp1	U32	Control	DSS accumulator increment	Optional	0	YES
SGFreq1	U32	Control	Phase control	Optional	100000	YES
SGPhase1	U32	Control	Phase control	Optional	0	YES
SGUpdateRate1	U32	Control	Update rate	Optional	40	YES
SGFref1	U32	Indicator	Reference frequency	Optional	100000000	YES

6.2.3 Source code description

The initialization of the library and the IRIO device is done with the call to irio_initDriver. This is the piece of code showing the header files, the variable definition and the call.

```
/***********************************************/
/*              INCLUDES REQUIRED              */
/***********************************************/
#include <irioDriver.h>
#include <irioDataTypes.h>
#include <irioHandlerAnalog.h>
#include <irioHandlerDigital.h>
#include <irioHandlerDMA.h>
#include <irioHandlerImage.h>
#include <irioHandlerSG.h>
#include <irioResourceFinder.h>
/*************************************

/***********************************************/
/*              VARIABLES REQUIRED             */
/***********************************************/

irioDrv_t p_DrvPvt; // Device data structure
TIRIOStatusCode st=IRIO_success; //status
TStatus myStatus; //
status.msg=NULL; //concatenated error messages
```

```
status.code=IRIO_success; //status
```

```
myStatus=irio_initDriver("RIO0","0x0177A2AD","PXIe-7966R",
"FlexRIOMod5761_7966","V1.1",1,headerFilePath,bitFilePath,&p_DrvPvt,&stat
us);
//INPUT OUTPUT parameters description
//@param[in] appCallID      Name for this irioDriver session.
//@param[in] DeviceSerialNumber S/n of the RIO target device
//@param[in] RIODeviceModel Model number of the RIO target device
//@param[in] projectName    Part of the name of the bitfile to be
downloaded into the FPGA
//@param[in] FPGAversion    Version of the bitfile (e.g. [1,1] for v1.1).
Must match with the value of the FPGA register
//@param[in] verbosity      Indicates whether or not should the driver
print trace messages.
//@param[in] headerDirPath where to search for the header file
corresponding to the bitfile to be downloaded
//@param[in] bitfile  Path where to search for the bitfile to be
downloaded
//@param[out] p_DrvPvt      Pointer to the driver structure. Will be
initialized with the resources found.
//@param[out] status        Warning and error messages produced during
the execution of this call will be added here.
//@return \ref TIRIOStatusCode result of the execution of this call.
```

irio_initDriver call downloads the bitfile in the FPGA, identifies all the resources implemented and returns the implemented resources data structure (p_DrvPvt) for later access to the FPGA. The FPGA is set to execution state with when calling irio_SetFPGAStart with value parameter set to 1. The execution of this function implies the detection of correct adapter module by the FPGA and the completion of the initialization steps defined by the hardware designer (LabVIEW for FPGA designer)

```
myStatus=irio_setFPGAStart(&p_DrvPvt,1,&status);
//@param[in] p_DrvPvt Pointer to the driver session structure
//@param[in] value    0=Do nothing, 1=Start FPGA if not already started
//@param[out] status  Warning and error messages produced during the
execution of this call will be added here.
//@return \ref TIRIOStatusCode result of the execution of this call.
```

Once the FPGA is running the user can interact with it using the IRIO API. In this case the next call is to irio_setDebugMode setting debug to OFF.

```
myStatus=irio_setDebugMode(&p_DrvPvt,0,&status); // Debug mode set to OFF
```

This call to irio_setSGSignalType initialises the signal generator 0 to generate a DC signal (this signal generator is implemented in the FPGA hardware).

```
myStatus=irio_setSGSignalType(&p_DrvPvt,0,0,&status); // DC signal
configured, value 0
```

This call to irio_setAO writes to the AO0 control in the FPGA.

```
myStatus=irio_setAO(&p_DrvPvt,0,2048,&status); // Set AO to 2048 digital
value
```

This call to irio_setAOEnable enables the update of AO0.

```
myStatus=irio_setAOEnable(&p_DrvPvt,0,1,&status); // AO enable
```

This application is using DMA to implement the data movement between the FPGA and the host. Before starting the data acquisition it is necessary to configure the DMA. The call irio_setUpDMAsTtoHost configures the DMA buffers, starts all of them and clears them in order to be ready for data acquisition.

```
myStatus=irio_setUpDMAsTtoHost(&p_DrvPvt,&status);
// DMAs are configured
```

This call configures the sampling rate used by the RIO device. The sampling rate programmed in the RIO device is obtained dividing the reference frequency of the design, Fref; and the value in S/s desired by the user. The result is an integer number that have to be used as input parameter in this call.

```
myStatus=irio_setDMATtoHostSamplingRate(&p_DrvPvt,0,1000,&status);
// DMA SamplingRate is configured to Fref/1000-->100kS/s
```

Once DMA and sampling rate are correctly configured the user can activate the data movement in the FPGA using this call. DMAs could use a big amount of the bandwidth available in the system. This call allows the user the activation/deactivation of DMA data movement for debugging purposes.

```
myStatus=irio_setDMATtoHostEnable(&p_DrvPvt,0,1,&status);
//DMA data transfer to Host is activated
```

This call starts data acquisition in the RIO device. Once data acquisition is started the DMA process move data to host.

```
myStatus=irio_setDAQStartStop(&p_DrvPvt,1,&status);
```

```
// Data acquisition is started
```

This call attempts to read one block of data form the buffer. The variable elementsRead returns the number of blocks read. This function can be called until the data acquisition process ends. In the example only one block of data is acquired. Pay attention that dataBuffer variable contains the binary values acquired from the FPGA. The data organization of this buffer is explained in the design rules document [RD2]

```
myStatus=irio_getDMATtoHostData(&p_DrvPvt,1,0,dataBuffer,
&elementsRead,&status);
//1 block of 4096 64 bit words are expected to be acquired
```

The next part of the program configures the internal signal generator and acquires the signal

```
irio_setDebugMode(&p_DrvPvt,0,&status); // Debug mode set to OFF
usleep(100);
irio_setSGSignalType(&p_DrvPvt,0,0,&status); // DC signal configured
myStatus=irio_setSGUpdateRate(&p_DrvPvt,0,10,&status);
// (Fref/SGUpdateRate)=S/s e.g., the value 10 means 10MS/s
…..
accIncr=100*(4294967296.0/10000000.0);
irio_setSGFreq(&p_DrvPvt,0,accIncr,&status);
//SGFreq=(freq_desired*(2to32))/(Fref/SGUpdateRate))
myStatus=irio_setSGAmp(&p_DrvPvt,0,4096,&status);
usleep(100);
irio_setDAQStartStop(&p_DrvPvt,0,&status);
//DMA buffer is full of data
irio_cleanDMAsTtoHost(&p_DrvPvt,&status);
irio_setAOEnable(&p_DrvPvt,0,0,&status);
usleep(100);
irio_setSGSignalType(&p_DrvPvt,0,1,&status);
usleep(100);
irio_setAOEnable(&p_DrvPvt,0,1,&status);
usleep(100);
…..

irio_setDAQStartStop(&p_DrvPvt,1,&status);
do{
  irio_getDMATtoHostData(&p_DrvPvt,1,0,dataBuffer,elementsRead,&status);
….
…..
…
}while (sampleCounter<1);

myStatus=irio_closeDriver(&p_DrvPvt, &status);
…
return 0;
}
```

NOTE: It is necessary that every time an IRIO function is used check it as is showed below:

```
St|=irio_setDAQStartStop(&p_DrvPvt,0,&status);
if(St!=0)
{
    printf("[ERROR] %s\n", status.msg);fflush(stdout);
```

```
    free(status.msg);status.msg=NULL;status.code=IRIO_success;
    return 0;
}
```