Centro Nacional de Pesquisa em Energia e Materiais

LNLS

Project NHEENGATU (NHE)

EPICs support for CompactRIO FPGA and LabVIEW-RT

DAWOOD ALNAJJAR

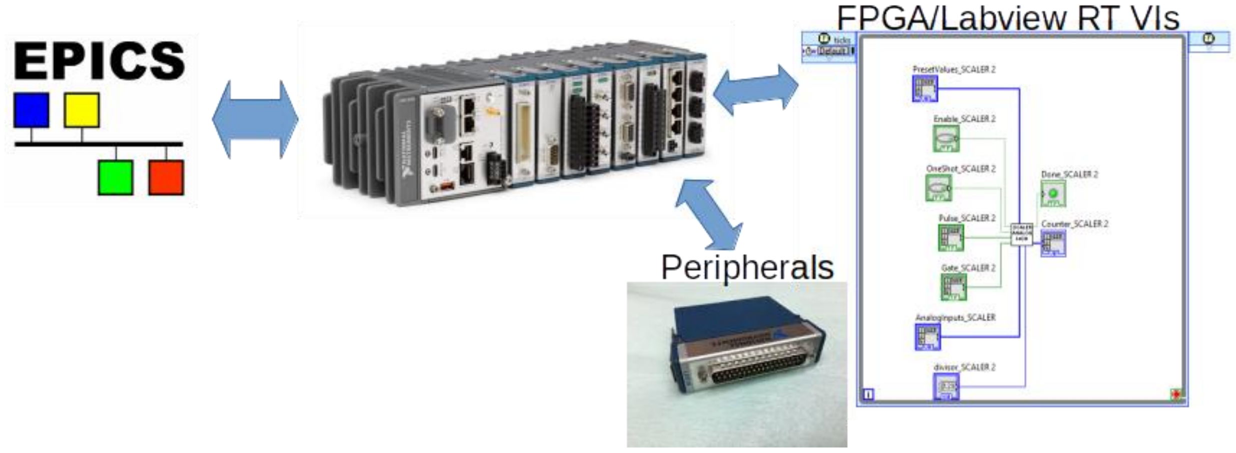
SOL

version 1.0

Introduction

At LNLS, a decision was made to use EPICS[1] to control CompactRIO[2] (CRIO) devices, the real-time embedded industrial controllers by National Instruments (NI). The EPICS platform is an open-source set of libraries that helps building distributed soft real-time control systems for scientific instruments, like the ones that are used in the CNPEM UVX beamlines, and those to be used in Sirius.

CRIO contains a Xilinx FPGA and an Intel processor running an NI version of Linux. It also contains several slots where NI peripherals can be plugged in. The I/O devices can be accessed by the FPGA (hardware) and the LabVIEW-RT (a real-time operational system). Both the FPGA and LabVIEW-RT can be programmed using a visual programming language. In that development environment, the user visually connects building blocks (VIs) to generate the desired circuitry to be implemented on LabVIEW-RT or the FPGA. Whether the LabVIEW-RT or the FPGA or both are chosen, it depends on the system's requirements. The FPGA can be used to perform high-speed processes and high-speed fine-control. LabVIEW-RT is slower than FPGA; however, it provides the ease and power of programming much more complex task without bounds.



EPICS accessing peripherals and VIs deployed in LabVIEW-RT and FPGA

For this purpose, a project called **Nheengatu**[3] (or **NHE**), was initiated with the following objectives in mind:

[1] https://epics.anl.gov/
[2] http://www.ni.com/pt-br/shop/compactrio.html
[3] The chosen project name, *Nheengatu,* is the name of an indigenous language of South America, from the Tupi–Guarani language family. Known also as *Língua Geral* (Portuguese for "general language") nheengatu was the *lingua franca* among the different indigenous peoples, the Portuguese explorers, black slaves and catholic missioners during the Brazilian colonial era. Its speakers, in many regions from the Amazon to São Paulo, were more numerous than the speakers of Portuguese. The language kept an important role in the integration of Brazil until the early 18th century. Then its use was banned by a decree of the Portuguese Empire. Nowadays, nheengatu is still spoken by about 30 thousand people in indigenous tribes through both sides of the Brazil-

- Publish variables from the FPGA and LabVIEW-RT VIs running on CompactRIO through EPICS
- Allow EPICS to alter variables on the FPGA and LabVIEW-RT running on CompactRIO
- Allow EPICS to read/write to peripherals plugged in the CompactRIO platform directly
- Allow EPICS to utilize the FPGA resource to extend functionality such as scaler records or filters
- Use the Linux RT to run the EPICS server (IOC)
- The CRIO deploy process should be as simple as possible
- Integrating the IOC with support software such as synApps should be possible

Colombia and Brazil-Venezuela borders. The language name is derived from the words *nheen* (meaning "tongue" or "to speak") and *katu* (turned *gatu* and meaning "good"), resulting a "good language" or "easy language", which was one of the main objectives during the development of this project.

Version Information

The version information is as follows

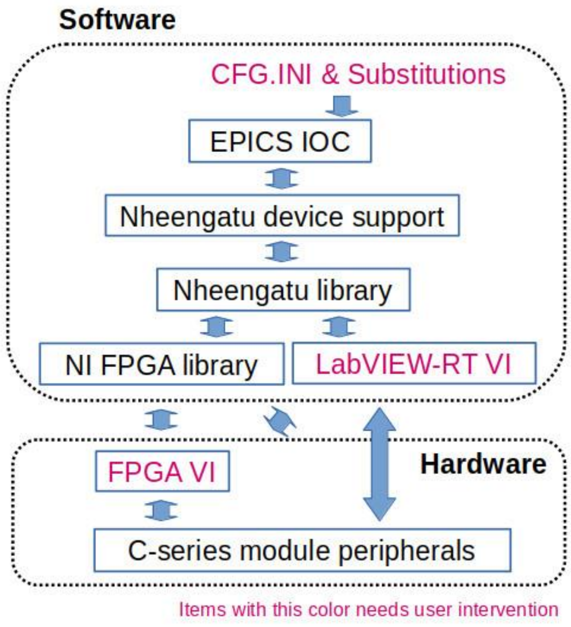
- LabVIEW 2019 32-bit sp1
- VI package manager 2019 f1 (if crashes download the version from https://vipm.jki.net/)
- LabVIEW 2019 FPGA Module Xilinx Compilation Tool for Vivado 2017.2
- FPGA Interface C API 19.0
- LabVIEW 2019 Real-Time Module
- EPICS 3.15.6
- Synapps R6.0
- NI Linux Inter-process communication v.1.5.1.19 (NI package manager)
- CRIO firmware version 6.5.0f0

Proposed Architecture

For simplicity and organization, the architecture was divided into five components as follows:

1. C-series modules: modules with peripherals that will be plugged in the CRIO platform
2. LabVIEW-RT and FPGA VIs: containing all application-specific implementations whether they are control, scalers, direct connections to peripherals, etc.
3. NI FPGA library: library provided by National Instruments to access FPGA controls and indicators
4. NHE library: library to handle all LabVIEW-RT and FPGA variable reading and writing
5. NHE EPICS device support: acting as an interface between the CRIO library and the IOC; also, it provides some functions to configure the CRIO library
6. EPICS IOC: piece of software that will run in the CRIO host and that will export all FPGA and LabVIEW RT available variables to the network though EPICS
7. INI file automatic generation script

An illustration of how these components communicate between them is shown in the following figure:



Communication between Nheengatu project components

Each of the above components will be explained in the following subsections.

C-series module Peripherals

These are the modules that will be plugged in the CRIO platform. To the present date, we identified that the modules to be used in Sirius are as follows:

- NI 9215: 4 AI, 24-bit,100KS/s
- NI 9207: 16 AI, 24-bit, 500S/s
- NI 9263: 4 AO, 16-bit, 100KS/s
- NI 9264: 16 AO, 16-bit, 25KS/s
- NI 9260: 2 AO, 24-bits, 51 KS/s
- NI 9403: 32 DIO, 7us
- NI 9401: 8 DIO, 100ns
- NI 9425: 32 DI, 7 us
- NI 9474: 8 DO, 1us
- NI 9226: Temperature sensor
- NI 9210: Temperature sensor 4 AI, 14 S/s
- NI 9216: Temperature sensor 400 S/s

As can be seen above, all modules can be classified into one of four categories and they will be treated as such:

1. Analog input
2. Analog output
3. Binary input
4. Binary output

Other functionalities include:

- MBBI
- MBBO
- Scaler
- Waveform

FPGA VI

The FPGA VI will have all FPGA-specific logic. For the upper layers to read from and write to FPGA variables, these variables must be defined in the FPGA VI in the form of scalars or arrays of Controls and Indicators. Please note that the indicator/control name is very important. Some keyword must be used to identify the name to distinguish it from other variables that should not be passed to EPICS (e.g. for AI, the keyword could be *ai.* No matter what the keyword is, it needs to be provided to the automation scripts). In the following subsections, it will be demonstrated how to interact with the input and output scalar and array variables.

Analog input (AI)

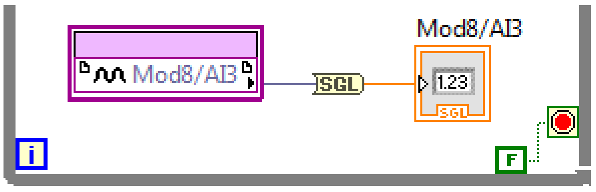
Two types of Analog Input were developed

- Single precision floating point analog input
- Fixed-point analog input

Each has its own advantages and will be described in following sections

Single Precision Floating point Analog Input

The AI must be of type indicator and must be defined as a Single precision floating-point indicator (SGL). This type of variable is limited to 23-bits of precision; however, it covers a large range of data (1.175494351 E -38 ~ 3.402823466 E +38). An illustration is as follows:

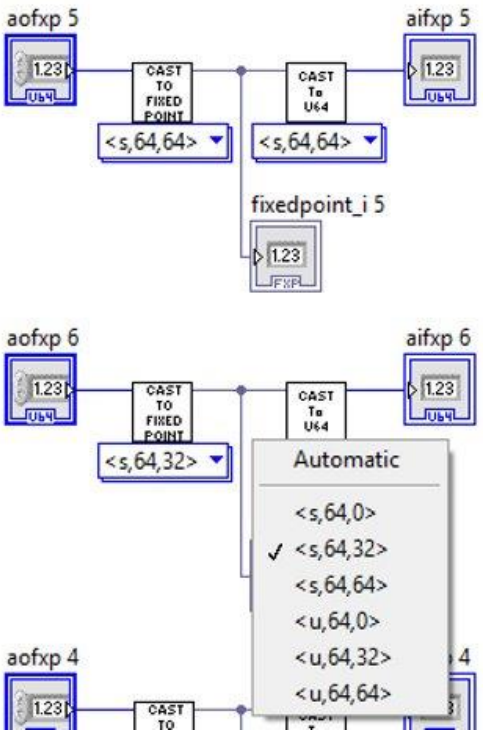


Analog input conversion to single precision floating point

If the FPGA VI is generating an analog variable, it must also be converted to a single precision floating point indicator as exhibited above in the same format. **Note that the single precision floating point is not supported in a single cycle timed loop.**

Fixed-point Analog input

Fixed point analog input was developed to deliver high precision variables to Nheengatu (up to 52 bits limited by EPICS double precision floating point conversion). An IP was developed in the FPGA to cast variables from fixed-point to unsigned 64-bit variables. This value is passed to Nheengatu, and the unsigned 64-bits are converted back to a double precision floating point. Given that the variable in the FPGA does not exceed 52-bits, it can be reconstructed using a double precision floating point without using a single bit of precision. Th IP was encapsulated by a polymorphic VI. An example of the instantiation of the polymorphic VI is shown as follows



The main polymorphic VI is named "Cast to U64.VI"

It has 6 types as follows

- signed, word length 64, integer length 0
- signed, word length 64, integer length 32
- signed, word length 64, integer length 64
- unsigned, word length 64, integer length 0
- unsigned, word length 64, integer length 32
- unsigned, word length 64, integer length 64

Obviously, the inputs will not be aligned with these options, so a "to fixed-point" vi will be used to convert to these formats. **Note that the Fixed point precision is supported in a single cycle timed loop.** The CAST TO FIXED POINT and CAST TO U64 VIs are **ONLY** supported in the single cycle timed loop.

e.g. :

You have a variable formatted as <s,52,30>, which is a signed 52-bit word with 30-bits representing an integer. By calculation, the fraction part is 22 bits (52 - 30 = 22). That means we have 30 bits of integer and 22 bits of fraction that we would like to maintain. The safest option would be the <s,64,32> since it has 32-bits of fraction and 32-bits of integer. It is important to remember that for numbers higher that 52-bits, precision will be lost due to EPICS using doubles to move around data.

Analog output (AO)

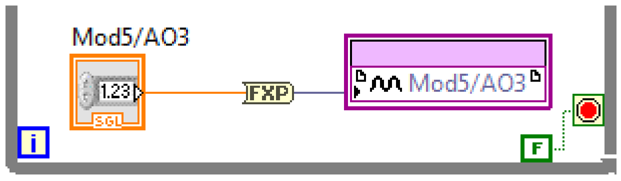
Two types of Analog Outputs were developed

- Single precision floating point analog output
- Fixed-point analog output

Each has its own advantages and will be described in following sections

Single Precision Floating point Analog Output

The Single precision AO must be defined also as a single precision floating-point control (SGL). Then it can be handled as desired. An illustration is as follows:



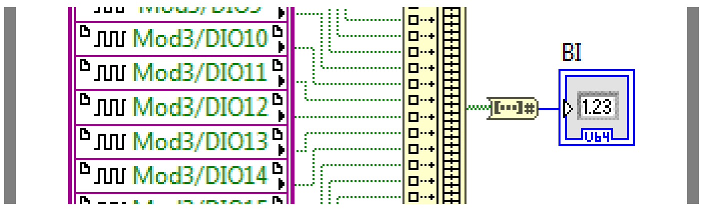
Analog output control of type SGL

Fixed-point Analog output

Fixed point analog output was developed to deliver high precision variables from Nheengatu (up to 52 bits limited by EPICS double precision floating point conversion). An IP was developed in the FPGA to cast variables from unsigned 64-bit variables to Fixed-point. This value is converted by Nheengatu to unsigned 64-bits are converted back to a double precision floating point in the FPGA. Given that the variable in the FPGA does not exceed 52-bits, it can be reconstructed using a double precision floating point without using a single bit of precision. Th IP was encapsulated by a polymorphic VI, and is the same as the VI of the Fixed-point Analog input, however inverted.

Binary input (BI)

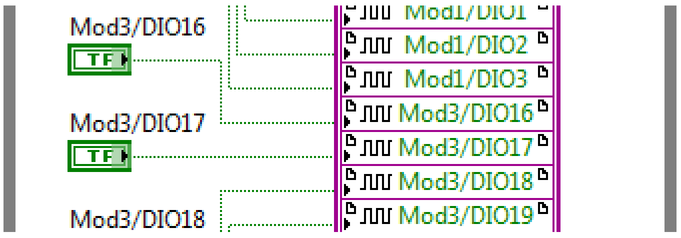
The BI can be defined as an unsigned 64-bit (U64) indicator or a boolean indicator. In the case of the U64 indicator, all BI are joined to one single indicator that supports up to 64 BIs using "Build array" and "Boolean array to number" VIs of LabVIEW. This implementation may come in handy upon having too many BIs. An example is illustrated as follows:



BI conversion to U64 indicator

Binary output (BO)

The BO must be of type control and must be defined as a Boolean. An example connecting the peripherals is illustrated as follows:



BO control directly connected to a peripheral

Scaler

The 64-counter scaler was developed in VHDL, and is imported as an IP in LabVIEW. The benefits of doing this is that we can perform easier validation in VHDL. In addition, exploiting the generate function of VHDL, this kind of logic requires less effort to build. The building blocks of the scaler IP are as follows

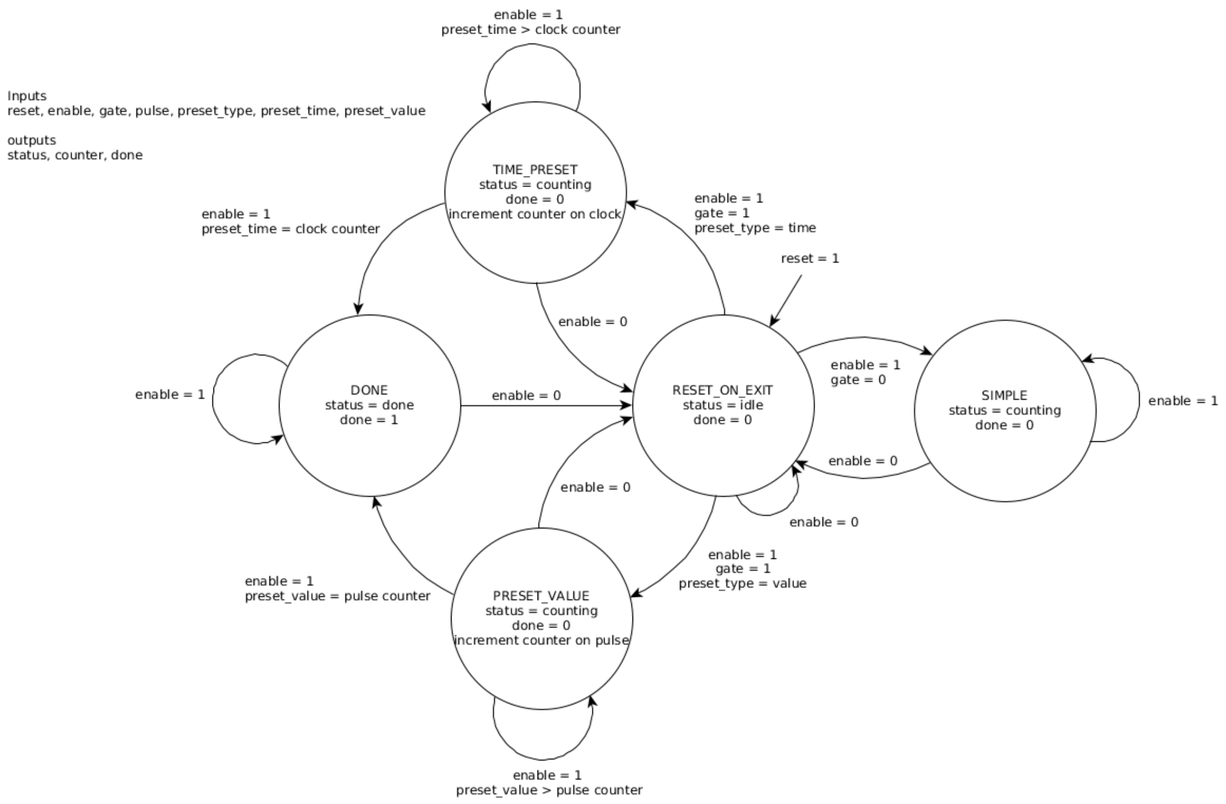
- `scaler_64_lv_wrapper.vhd`: wrapper TOP to make it accessible to LabVIEW

- **`scaler_64.vhd`**: TOP without wrapping
- **`scaler_ctrl.vhd`**: Scaler state machines and control logic
- **`scaler_s1.vhd`**: Scaler counter S1 implementation
- **`scaler_sx.vhd`**: Scaler counters S2-S64 implementations
- **`Utils.vhd`**: containing type definitions

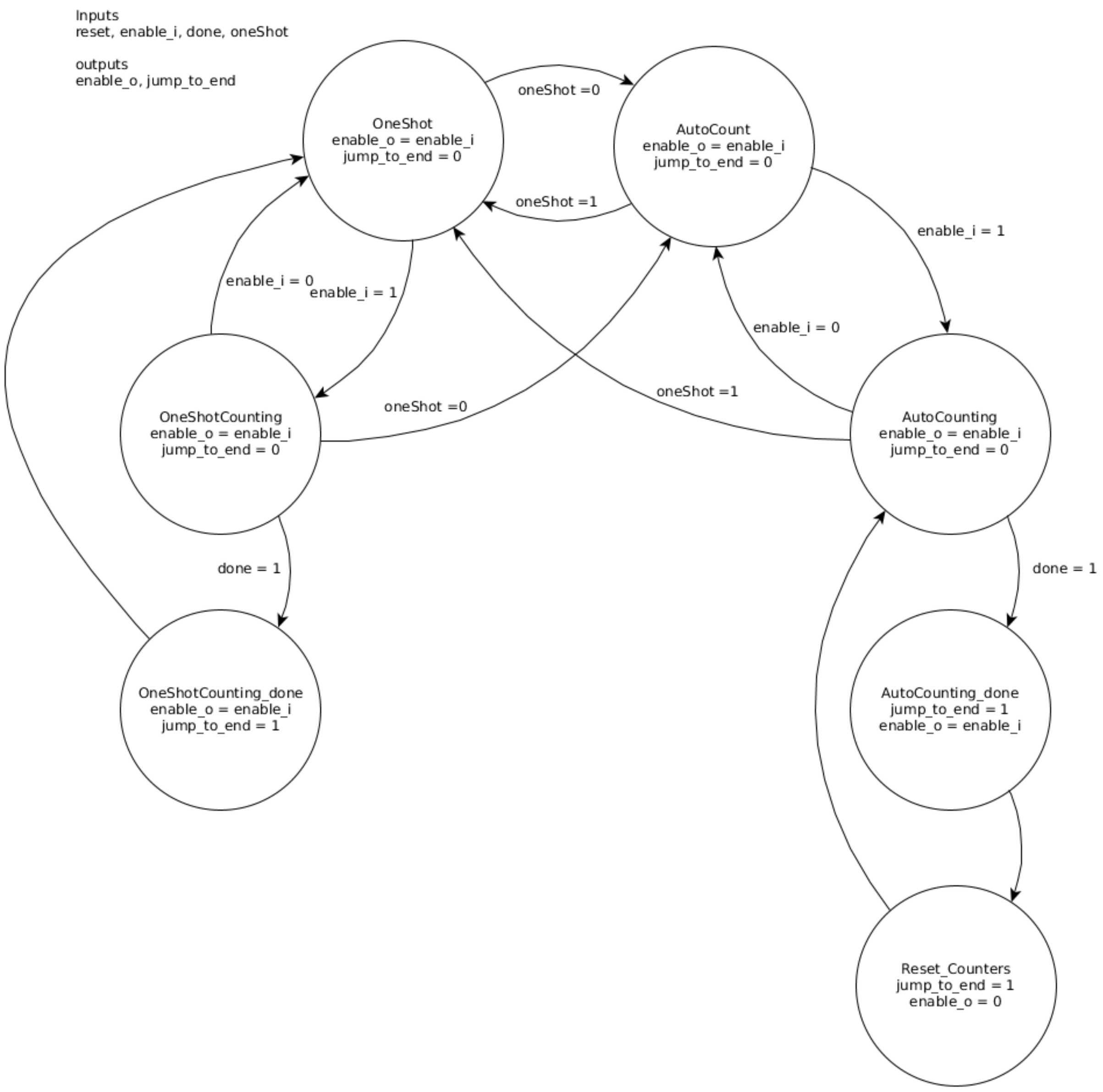
The wrapper file **`scaler_64_lv_wrapper.vhd`** was developed as the top that will be imported and parsed in LabVIEW. At the time this scaler was developed, LabVIEW did not support arrays as an input or an output. The real top is scaler_64.vhd.

The scaler IP supports autoCount and is a standalone module (it does not need EPICS). However, EPICS implements autoCount. So, when this block is used with EPICS, the autoCount feature will be disabled by the Nheengatu library.

The single counter block state machine is as follows



The general scaler 64 state machine is as follows:

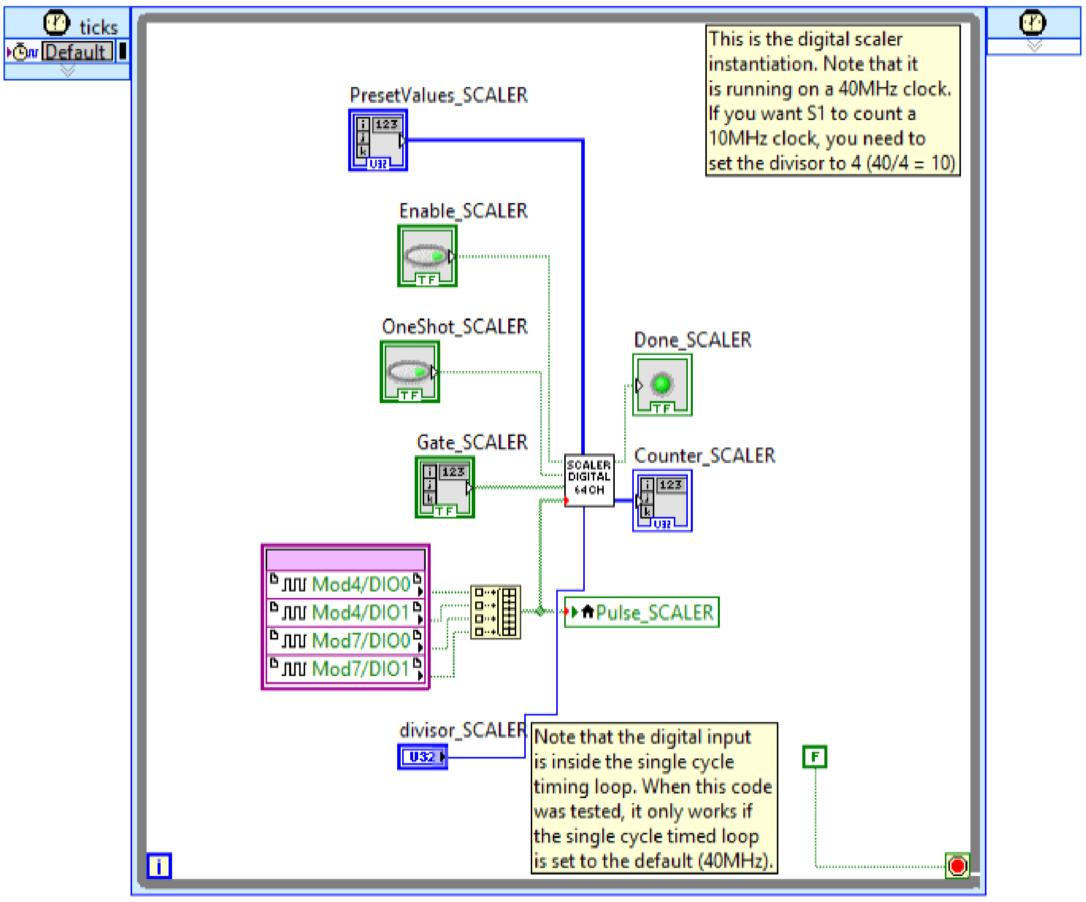


These state machines give a general idea on how the VHDL works. For easier importing of scaler, a llb file was created in the nheengatu LabVIEW reference project. This library contains 2 upper-level modules, and one wrapper for the IP. The wrapper does not need to be used directly; however, the two modules should be used when necessary. The two modules are:

1. Analog Scaler, with 64 counters
2. Digital Scaler, with 64 counters

Both modules use the same wrapper; however, the analog is made to sum analog input periodically, and the digital is made to count single-bit pulses.

An example of the digital scaler instantiation is as follows:



This is the digital scaler instantiation. Note that it is running on a 40MHz clock. If you want S1 to count a 10MHz clock, you need to set the divisor to 4 (40/4 = 10)

Note that the digital input is inside the single cycle timing loop. When this code was tested, it only works if the single cycle timed loop is set to the default (40MHz).

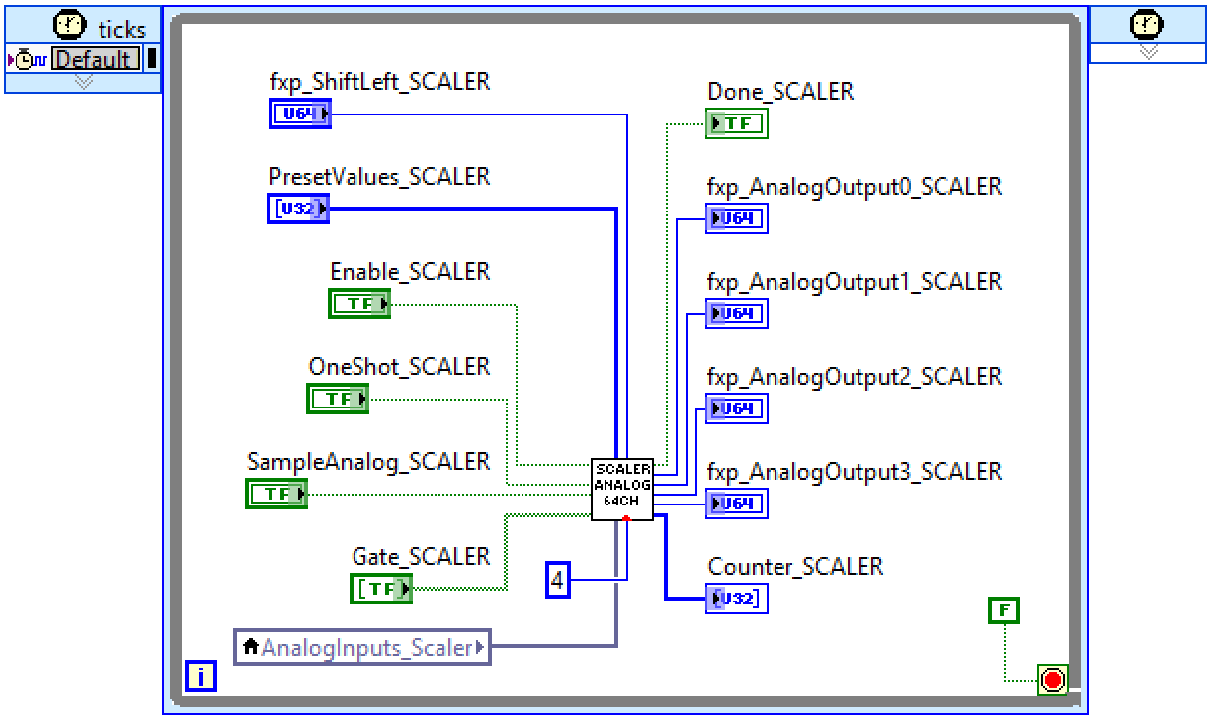
Digital scaler instantiation in FPGA VI

Note that the single-cycle timed loop is running on the default clock (40MHz).

It has the following ports:

1. PresetValues: input, and is used to set the counter preset in case
2. Enable: input, starts the counters whether in preset mode or simple mode
3. OneShot: input, hardware supported oneShot or autoCount. With EPICS, this will be automatically disabled.
4. Gate: input, upon setting, it will enable preset mode
5. Pulse: input, the pulses which will be counted
6. Divisor: input, Counter S1 counts one every Divisor cycles (eventual count will be 40MHz/divisor). This is extremely useful when we have long experiments so the counter dos not saturate, and i the same time we can catch pulses that come on high frequency!
7. Done: output, in case of preset mode, this indicates that the scaler has reached one of its presets
8. Counter: outputs of the 64 counters

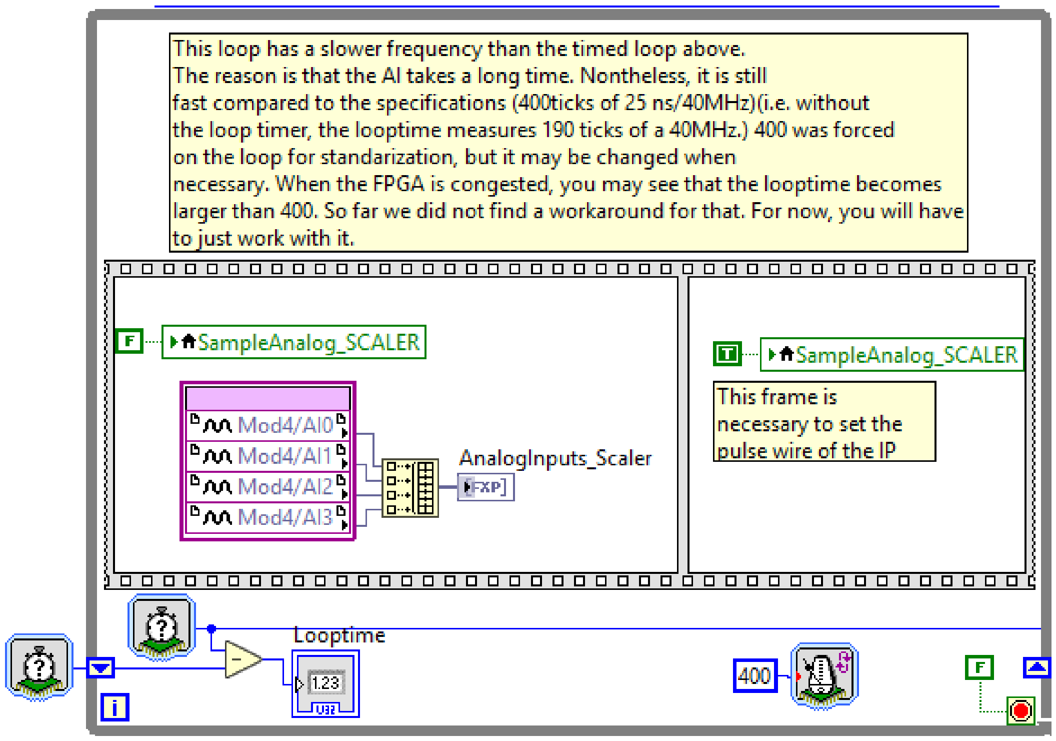
Following is an example of the Analog scaler instantiation:



Analog scaler instantiation in FPGA VI

There is several extra ports when compared to the digital implementation. That is as follows:

- AnalogInputs: The input to be summed on the positive edge of the SampleAnalog port. A simple way of moving the data to this loop can be exhibited as follows
- fxp_AnalogOutputX_SCALER: The data sampled at the analog input module



This loop has a slower frequency than the timed loop above.
The reason is that the AI takes a long time. Nontheless, it is still
fast compared to the specifications (400ticks of 25 ns/40MHz)(i.e. without
the loop timer, the looptime measures 190 ticks of a 40MHz.) 400 was forced
on the loop for standarization, but it may be changed when
necessary. When the FPGA is congested, you may see that the looptime becomes
larger than 400. So far we did not find a workaround for that. For now, you will have
to just work with it.

This frame is
necessary to set the
pulse wire of the IP

One way to obtain analog data every 100us

Waveform

The waveform was implemented to move arrays from the FPGA to EPICS using the Waveform record. The waveform arrays must be indicator arrays of the following variable types:

- I64 : Signed integer 64-bits
- I32 : Signed integer 32-bits
- I16 : Signed integer 16-bits
- I8 : Signed integer 8-bits
- U64 : Unsigned integer 64-bits
- U32 : Unsigned integer 32-bits
- U16 : Unsigned integer 16-bits
- U8 : Unsigned integer 8-bits
- SGL : Single precision floating point

There is no support for Fixed-point type and Double (double type is not supported by the FPGA).

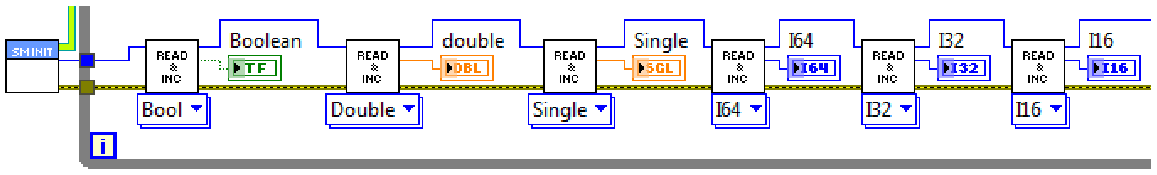
Due to compilation limitations, only small arrays were tested on the FPGA (up to 64 elements). Larger arrays take a long time to synthesis, and accordingly have not been tested. Nonetheless, the infrastructure is available and functional, and newer versions of LabVIEW may have this issue resolved. In the meantime, if a relatively large array becomes necessary, a FIFO can be used between the FPGA and LabVIEW-RT, and the array can be generated there and then passed to EPICS (described in following sections).

LabVIEW-RT VI

In order to read and write variables to and from the LabVIEW-RT VI and EPICS, 4 VIs were developed, and are as follows

1. **`IOC Shared memory initialize.vi`**: initializes shared memory. All input can be left as default. A simple instantiation is more than enough.
2. **`IOC Shared memory de-initialize.vi`**: uninitializes shared memory.
3. **`SM read and increment.vi`** (polymorphic VI): reads the variable from EPICS
4. **`SM write and increment.vi`** (polymorphic VI): writes the variable to EPICS

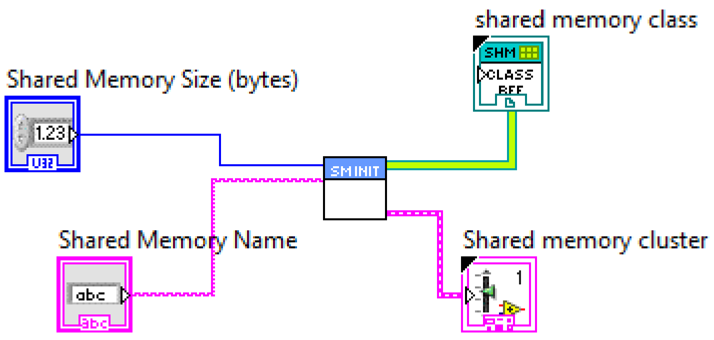
Usage is extremely simple. The ptr_in and ptr_out of all blocks must be chained in sequence. The ptr is generated by the Shared memory initialize.vi, and is referenced by all the other 3 VIs. An illustration is as follows:



Chain pointer inputs and outputs together

Shared memory initialize.vi / uninitialize.vi

A diagram of the inputs and outputs is depicted as follows:



Shared memory initializes inputs and outputs

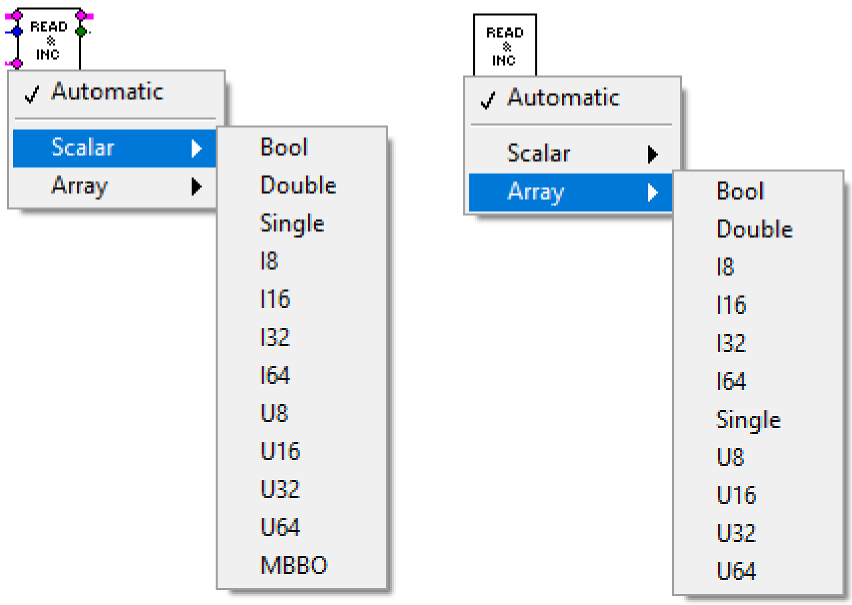
The iputs and the outputs are as follows:

- Shared memory size: default is 1 page size (4096). If changed, the new value must also be informed in the cfg-ini that is given to the nheengatu library. (optional)
- Shared memory name: default name is labview_linux_sm. This file is automatically created in the CRIO in **`/dev/shm/labview_linux_sm`**. If this is to be changed, must also be changed in the cfg.ini file that is given to the nheengatu library. (optional)
- Shared memory class: is a structure that has all shared memory specific data.
- Shared memory cluster: A bundle that will be chained from one shared memory block to another.

Shared memory de-initialize contains only error-in and shared memory-in terminals and is totally passive.

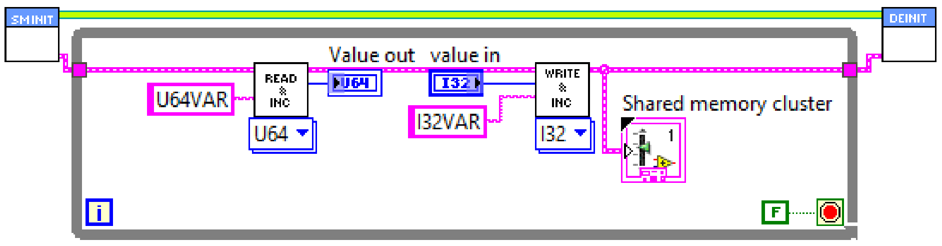
Read/write and increment polymorphic VIs

The polymorphic VIs supports scalar and array of the following data types: Bool, double, Single, I8, I16, I32, I64, U8, U16, U32, U64. Can also be left for LabVIEW to do it automatically. An illustration is as follows



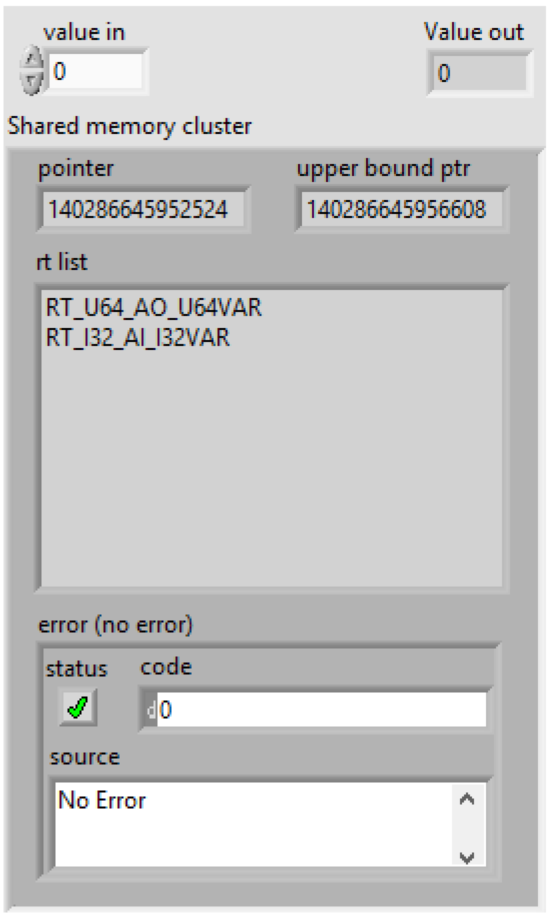
Choose the VI input/output type using polymorphic VIs in LabVIEW

A simple example of opening shared memory, reading a 64-bit unsigned variable from EPICS, writing a 32-bit signed variable to EPICS, and then closing the shared memory is shown in the following illustration



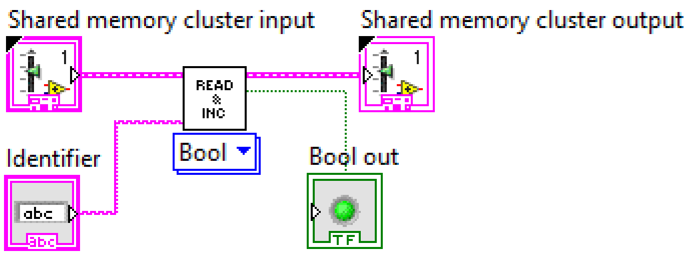
LabVIEW-RT - EPICS shared memory example

The front panel of the previous example is as follows


value in
0
Value out
0
Shared memory cluster
pointer
140286645952524
upper bound ptr
140286645956608
rt list
RT_U64_AO_U64VAR
RT_I32_AI_I32VAR
error (no error)
status
✓
code
0
source
No Error

As can be seen, out-of-range memory errors or other kind of errors are shown in the interface. A list of peripherals is shown in the textbox RT list. This will need to be copied as is to the RT.list file that will be processed by the automatic generation scripts.

Indicator and contol of the Boolean read and increment VI is shown in the following figure.



Read and increment inputs and outputs

The description is as follows

- Shared memory cluster input/output: This is a cluster that needs to be daisy chained from one read and increment block to another.
- Identifier: A string without spaces to identify the variable (will be used for the RT list)
- Bool out: Boolean output read from shared memory

NI FPGA Library

The NI FPGA library is a dynamic library built using the static files that get generated along with the header file and the bitstream of the FPGA VI project. We simply packed them in a dynamic library format.

Nheengatu Library

This is where the core of the Nheengatu solution relies. The Nheengatu library operates based on a single cfg.ini file that needs to be produced by the end user (using the INI automatic generation scripts). The library needs to know all the following information to be able to operate correctly:

- The IP of the CRIO where the FPGA bitfile will be written to
- The Path of the bitfile
- The shared memory path with the LabVIEW-RT
- Whether or not it will use the shared memory with the LabVIEW-RT
- The signature of the FPGA bitfile
- The name of the bitfile
- The shared memory size
- Binary input 64-bit variable address (if available)
- Binary input bit-name mapping (of the 64-bits) (if available)
- Analog output addresses (if available)
- Analog input addresses (if available)
- Binary output addresses (if available)
- List of scalers (if available)
- Addresses for scaler input and output ports (if available)
- Waveforms and their addresses (if available)
- Fixed-point analog inputs and outputs and their addresses (if available)
- Enumerate input (MBBI) (if available)
- Enumerate input (MBBO) (if available)

All this information must be available in the configuration file for the library to operate correctly. Later on, it will be demonstrated how this information will be obtained automatically using an auto-generation scripts. This file will be passed to the Nheengatu library through the software stack starting at the IOC.

The library abstracts interactions with either the FPGA VI, or the LabVIEW-RT VI, and makes it transparent for the upper layers. The upper layers do not need to know anything about the source of the data being acquired, nor the destination of the data being sent. The upper layer will be concerned only with the variable name.

The library uses "Boosts" library Maps to store address data, and, uses property trees' ini parser to parse the ini configuration file.

NHE EPICS device support

The Nheengatu device support includes the following records:

- Binary Input record
- Binary output record
- Analog Input record
- Analog Output record
- Scaler record
- Waveform record
- MBBO record
- MBBI record

It also contains some functions necessary for initialization and un-initialization. The initialization function must be called before anything in the command file of the IOC. The de-initialization function is hooked automatically to the EPICs exit functions, so it does not need to be called anywhere.

These are all the items that need to be imported from the device support. They need to be specified in the dbd file of the IOC:

- devScalerCRIO: Scaler device support
- devBICRIO: BI device support
- devAICRIO: AI device support
- devAOCRIO: AO device support
- devBOCRIO: BO device support
- devWAVEFORMCRIO: WAVEFORM device support
- devMBBICRIO: MBBI device support
- devMBBOCRIO: MBBO device support

Furthermore, there are three items that need to be imported by the IOC:

- debugCRIOSupVar: for debugging. Currently, it has very little or no use.
- CRIODebugReg: for debugging. It prints all accesses to the NHE library to the terminal and to a file in the /etc/autosave/ folder.
- CRIOreg : A hook for the function that will initialize all the CRIO environment. It enables the IOC to call the crioSupSetup function through the IOC command (cmd) file. crioSupSetup receives two parameters:
    - the path to the .ini file
    - The library hash information and version printing parameter: 0; do not print, 1 print version

NHE IOC implementation

The IOC was compiled with some db templates. If a new db is needed, the IOC must be recompiled. If not, the IOC can be executed with the appropriate substitutions file. The db templates have in common the following variables that will need to be filled in by the cmd file using the ***dbLoadTemplate*** command:

- BL: beamline name (e.g. SOL)
- LC: Location (e.g. Cabine A)
- EQ: equipment name (e.g. 9401A:BO0)
- PIN: pin name in the INI file mapped to this record (e.g. RT_DBL_MCOLLECTOR)
- DTYP: device name as defined in the dbd file which maps the device name with the devices in the device support (e.g. CrioBO)

The scaler has one more variable and that is FREQ, i.e. the frequency of the scaler to be specified so that the scaler middleware can estimate the precise time passed using the S1 counter.

The waveform has the FTVL (value type) and the NELM (array size).

There are several more fields that were defined based on necessity.

An example of the substitution files that will passed at runtime through the IOC shell is as follows

```
file "$(TOP)/db/devBOCRIO.db.template"

{

pattern

{BL, LOC, EQ, DTYP, PIN, DESC}

{"SOL", "A", "CRIO5:RT:BOOLRD", "CrioBO", "RT_BOL_BO_boolRead", "This is a description"}

{"SOL", "A", "CRIO5:9403A:bo0", "CrioBO", "", "This is a description"}

}
```

Note that RT variables also need to be included.

.INI file and substitutions automatic generation script

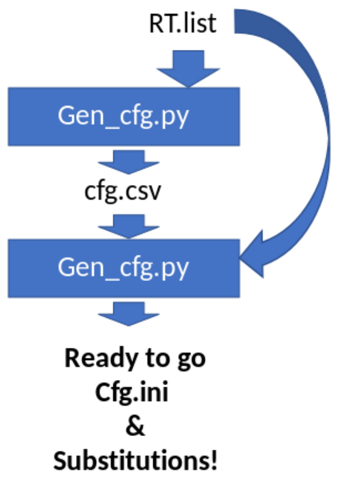
Whenever the content of an FPGA VI is altered, the indicator and the control addresses change. This generates a burden to the deployer everytime he needs to make a new *.ini file. We devised a script to read the header file generated by the LabVIEW FPGA VI and to produce a ready to use cfg.ini. This works given that the deployer provides correct input to the script and abides by the naming/control indicator type during the VI development. The script can be called using all the default parameters, as shown:

```
./gen_cfg.py -s ./gen_cfg_example/ -u --waveformkey waveform_ --binum 24 --crio CRIO4
```

Script parameters are as follows:

```
usage: gen_cfg.py [-h] [-u] [-d DST] [-p PATH] [-s SRC] [--binum BINUM]
          [--extract] [--ip IP] [--smfname SMFNAME] [--smsize SMSIZE]
          [--aikey AIKEY] [--aokey AOKEY] [--bokey BOKEY]
          [--bikey BIKEY] [--mbbikey MBBIKEY] [--mbbokey MBBOKEY]
          [--fxpkey FXPKEY] [--scalerkey SCALERKEY]
          [--waveformkey WAVEFORMKEY] [--beamline BEAMLINE]
          [--bidtyp BIDTYP] [--aidtyp AIDTYP] [--bodtyp BODTYP]
          [--aodtyp AODTYP] [--mbbodtyp MBBODTYP]
          [--mbbidtyp MBBIDTYP] [--wfdtyp WFDTYP] [--crio CRIO]
          [--loc LOC] [--scalerdtyp SCALERDTYP] [--cfgcsv CFGCSV]
```

The script also generates the db substitutions files that will be used by the crio-ioc.cmd file. The ini and the substitution file generation flow is as follows:



EPICS files generation flow

The flow can be divided in the following task:

1. build the RT.list file
2. Run the gen_cfg.py script with appropriate parameters
3. Update the cfg.csv file using some csv viewer (e.g. libre office)
4. run the gen_cfg.py again with the appropriate parameters to generate ready to use cfg.ini and substitutions files

1- RT.list generation

The RT.list is a file that simply contains all RT variables. All variables must follow a defined format for Nheengatu to infer information about them. The syntax is as follows <RT_VARTYPE_VARNAMEX>. VARNAME must be one of the following (except when VARTYPE is MBI/MBO)

- AI, AO, BI, BO, WF

X can be a unique number assigned by the user.

The VARTYPE has to be one of the following

- BOL, DBL, SGL, I64, I32, I16, I08, U64, U32, U16, U08, MBO, MBI

In case of waveforms (WF), the variable name must be followed by the array number of elements.

The order in the list is very important. The order must follow the order in the chain in the LabView-RT VI.

An example of the RT.list is as follows

RT_SGL_WF _IDENTIFIER 5

RT_U32_WF _IDENTIFIER 113

RT_BOL_BO _IDENTIFIER

RT_DBL_AO _IDENTIFIER

RT_DBL_AI _IDENTIFIER

RT_SGL_AI_IDENTIFIER

RT_MBO_IDENTIFIER

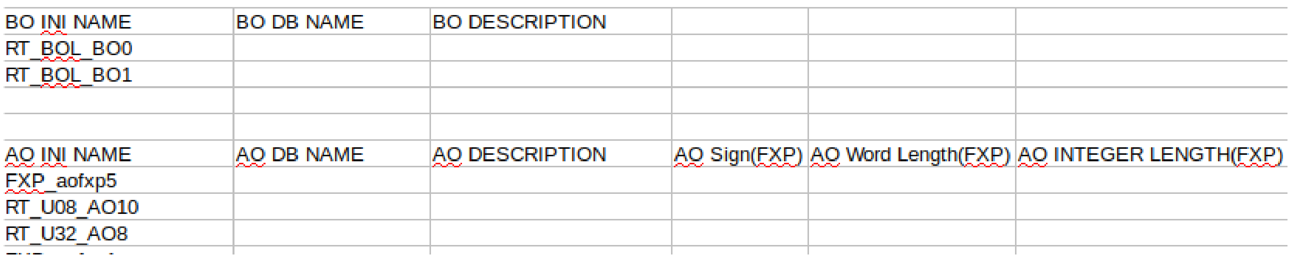
This file shows that LabVIEW-RT has an array of floats of size 5 in the beginning of the chain followed by an array of unsigned integers of size 113, followed by a Boolean output, followed by a analog output of type double, followed by and analog input of type double, followed by an analog input of time float, followed by an MBO.

The cluster in the labview RT project will generate the content of this file. It just needs to be copied manually.

2- First run of the cfg_gen.py script

Run with the --extract input parameter. The objective of this run is to use the data in the RT.list file, and all the data in the header file to generate a cfg.csv template file that needs to be filled by the user.

An example of the generated file is as follows



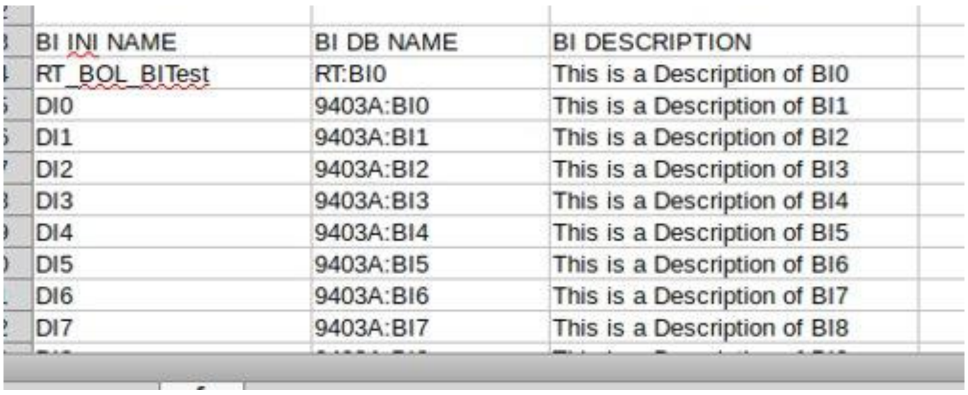
BO INI NAME	BO DB NAME	BO DESCRIPTION			
RT_BOL_BO0					
RT_BOL_BO1					
AO INI NAME	AO DB NAME	AO DESCRIPTION	AO Sign(FXP)	AO Word Length(FXP)	AO INTEGER LENGTH(FXP)
FXP_aofxp5					
RT_U08_AO10					
RT_U32_AO8					

A visualization of an empty template of cfg.csv

It can be noticed that the template is empty, and needs to be filled by the user. Only waveform size information will be filled in the table. Upon having a previously filled cfg.csv file, the flag --refcsv can be used. The common data will be extracted and copied to the new cfg.csv file.

3 - Updating the cfg.csv

The file contains EPICS description information, and other necessary variables.



BI INI NAME	BI DB NAME	BI DESCRIPTION
RT_BOL_BITest	RT:BI0	This is a Description of BI0
DI0	9403A:BI0	This is a Description of BI1
DI1	9403A:BI1	This is a Description of BI2
DI2	9403A:BI2	This is a Description of BI3
DI3	9403A:BI3	This is a Description of BI4
DI4	9403A:BI4	This is a Description of BI5
DI5	9403A:BI5	This is a Description of BI6
DI6	9403A:BI6	This is a Description of BI7
DI7	9403A:BI7	This is a Description of BI8

Visualization of the cfg.csv file using libre office

Items that need to be filled include

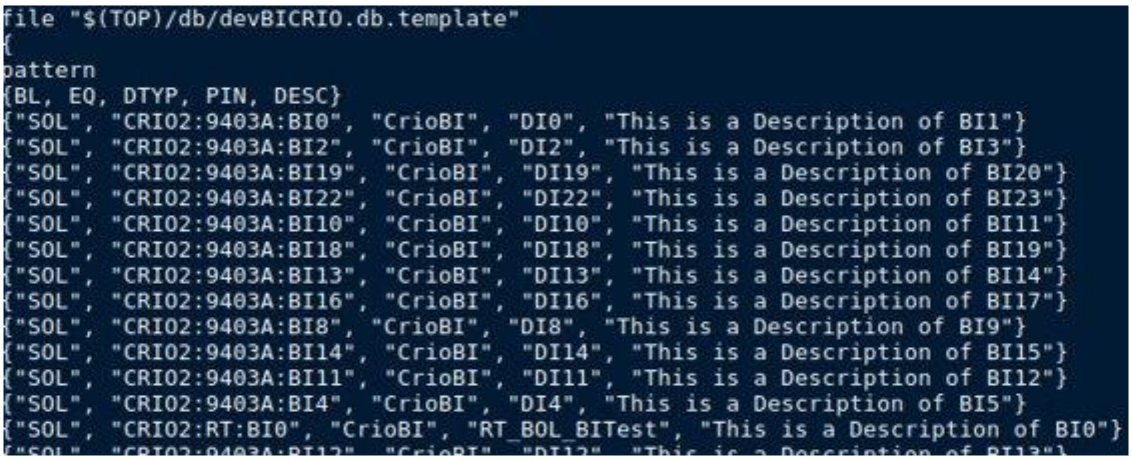
- Record name
- Record description
- If fixed point, requires sign, word length, and integer word length
- Whether to enable autosave or not
- whether to initialize the PV with an initial value, in which case, the initial value needs to be provided

Automatically generated items are

- Number of elements incase of waveform
- FXP size in case of scaler

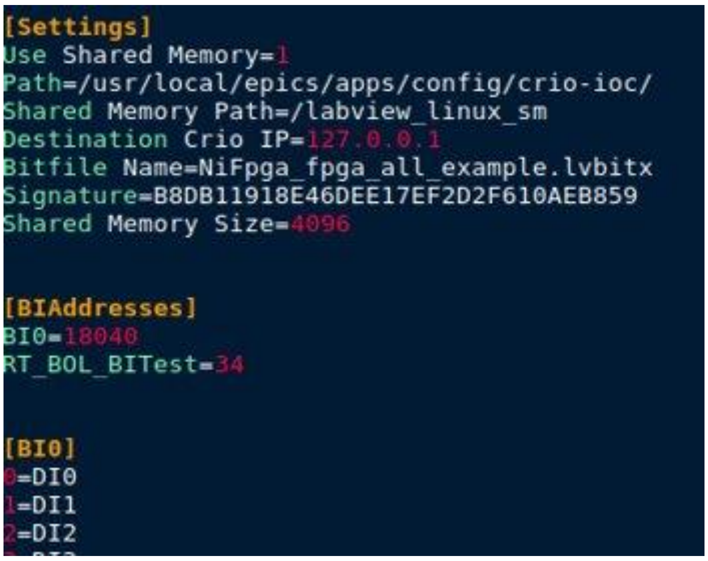
4 - run the gen_cfg.py again with the appropriate parameters to generate ready to use cfg.ini and substitutions files

The gen_cfg.py script must be run again with the same parameters, just without the --extract switch. In sequence, the script will generate all the EPICS substitutions files, cfg.ini for Nheengatu, and a reference folder that has all the information to be able to re-generate the generated data.



```
file "$(TOP)/db/devBICRIO.db.template"
{
pattern
{BL, EQ, DTYP, PIN, DESC}
{"SOL", "CRIO2:9403A:BI0", "CrioBI", "DI0", "This is a Description of BI1"}
{"SOL", "CRIO2:9403A:BI2", "CrioBI", "DI2", "This is a Description of BI3"}
{"SOL", "CRIO2:9403A:BI19", "CrioBI", "DI19", "This is a Description of BI20"}
{"SOL", "CRIO2:9403A:BI22", "CrioBI", "DI22", "This is a Description of BI23"}
{"SOL", "CRIO2:9403A:BI10", "CrioBI", "DI10", "This is a Description of BI11"}
{"SOL", "CRIO2:9403A:BI18", "CrioBI", "DI18", "This is a Description of BI19"}
{"SOL", "CRIO2:9403A:BI13", "CrioBI", "DI13", "This is a Description of BI14"}
{"SOL", "CRIO2:9403A:BI16", "CrioBI", "DI16", "This is a Description of BI17"}
{"SOL", "CRIO2:9403A:BI8", "CrioBI", "DI8", "This is a Description of BI9"}
{"SOL", "CRIO2:9403A:BI14", "CrioBI", "DI14", "This is a Description of BI15"}
{"SOL", "CRIO2:9403A:BI11", "CrioBI", "DI11", "This is a Description of BI12"}
{"SOL", "CRIO2:9403A:BI4", "CrioBI", "DI4", "This is a Description of BI5"}
{"SOL", "CRIO2:RT:BI0", "CrioBI", "RT_BOL_BITest", "This is a Description of BI0"}
{"SOL", "CRIO2:9403A:BI12", "CrioBI", "DI12", "This is a Description of BI13"}
```

Example of the Binary input file



```
[Settings]
Use Shared Memory=1
Path=/usr/local/epics/apps/config/crio-ioc/
Shared Memory Path=/labview_linux_sm
Destination Crio IP=127.0.0.1
Bitfile Name=NiFpga_fpga_all_example.lvbitx
Signature=B8DB11918E46DEE17EF2D2F610AEB859
Shared Memory Size=4096

[BIAddresses]
BI0=18040
RT_BOL_BITest=34

[BI0]
0=DI0
1=DI1
2=DI2
```

Generated cfg.ini example

Setup CRIO IOC Folders on NFS

to setup the folder heirarchy in the NFS, the script ***crioSetupBlFolders.sh*** in the NFS /usr/bin folder can be used.

The usage is as follows

./crioSetupBlFolders.sh <CRIO LOCATION> <CRIO POSTFIX> <CRIO IOC FOLDER NAME>

where the <NFS CRIO IOC to be used> is the folder of the crio ioc that you want to be linked in the folder heirarchy

e.g.

./crioSetupBlFolders.sh A CRIO06 2019_12_12_01

given that the folder heirarchy is as follows

```
[dawood.alnajjar@nfs-epics crio-ioc]$ tree -L 1 /usr/local/epics-nfs/apps/R3.15.6/crio-ioc

/usr/local/epics-nfs/apps/R3.15.6/crio-ioc

├── 2019_09_24_01

└── 2019_10_24_01

└── 2019_12_12_01

 3 directories, 0 files
```

The folder structure should look like this

<BBB>/LOC-CRIO<XX>/

|_ apps/config/crio-ioc/ -> here you will put specific configuration files for your CRIO generated by the gen_cfg.py script

|_ apps/crio-ioc -> link to crio-ioc version used (always use the last version)

|_ base -> link to epics base 3.15.6

After generating the folder structure, move it to /usr/local/setup-bl/<BEAMLINE> on the NFS.

Setup NHE IOCs in a new cRIO

After cRIO modules are in cRIO and the LabView projects are running (in FPGA and LabviewRT), it is necessary to configure cRIO Linux to run the IOCs. There are some steps to deploy NHE IOCs in cRIOL Linux, these steps are described below:

cRIO Name

An important setting, before starting to deploy IOC in cRIO Linux is configure cRIO name to SOL standard. The cRIO name should use the standard:

<BBB>-<LOC>-CRIO<XX>

The nomenclature is defined in this page and is constantly updated.

Where <BBB> is beamline abbreviation (the first 3 consonant), <LOC> is the location, and X is the <cRIO> number from beamline (an incremental number starting on 1).

cRIO setup

We built some scripts to help cRIO setup, to allow IOCs run.

To setup cRIO and creates a new user with sudo, and configure NFS, and several other tasks, uses the script on this repository:

https://gitlab.cnpem.br/SOL/Projetos/crio-first-setup

This script will leave the CompactRIO ready to run any EPICS command file in a matter of seconds. The script will create a user <SOL>, and a password of <solS0L>, which is the common sol password.

Repository

The repository of the Nheengatu project is available at:
https://gitlab.cnpem.br/SOL/Projetos/nheengatu

You can find a lot of resources available there too.

Compiling CRIO files

To compile the CRIO files. there are 2 containers in this repository. one that uses a CRIO chassi and compiles directly on it (for immediate testing) (branch compile_on_crio), and the other compiles without a CRIO chassi (branch master). Feel free to use either one depending on your needs.

Trouble shooting

Once the IOC is up and running, a debug interface was developed, and can be activated using the ioc command prompt. By typing help in the terminal, the list of possible commands should appear, among which is "crioDebug" function. crioDebug function receives a single parameter, 0 or 1, to disable and enable debug respectively. Enabling debug, all AI, AO, BI and BO instantiated reads and writes requested by the CrioLinux library will be printed in the terminal, and in a log file located in </opt/autosave> folder. Once the debug is no longer necessary, you can run crioDebug with the value 0 to disable debugging.

GDB can be used with EPICS. The library needs to be compiled with Debug flag instead of Release flag in order to be able to step in it.

Useful commands

```
LD_LIBRARY_PATH=<PATH TO NHE>
```

```
sshfs -o allow_other -o ssh_command="ssh -o StrictHostKeyChecking=no" <USER>@<IP>:<PATH IN CRIO> <PATH IN LOCAL MACHINE>
```