GPStudio Tutorial v1.21:
2. How to create and use a simple node project in command line mode?

Sebastien CAUX

March 3, 2017

1 Introduction

This tutorial is done to show how GPStudio is simple to use when you are using only IP in library on a supported board. The objective is to understand the basics of GPStudio tools and especially the command line tools. For this example, we use the famous Dreamcam platform, the first platform supported by GPStudio.

In this tutorial, we create a project for a specific platform and configure it. After that, we add some process in the project, setting up and connect it to image sensor and communication. Finally, we configure the smart camera with the compiled project and check results of processing on the viewer interface.

In a second time, we will create a custom process block externally configurable and test it on a real camera.

At the end of this tutorial, we will be able to use GPStudio with all the command line tools.

2 Driving the flow from image sensor to USB

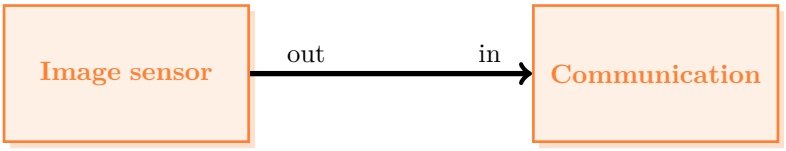

Figure 1: Simple connection from image sensor to communication block, first step

2.1 Create the project and configure platform

At the very beginning, check if your tools are well installed by simply launch:

```
> gpnode --version
# gpnode command line tool to manage a gpstudio node (v1.21)
```

If you do not have this result, you probably need to set your PATH system variable or to call the setenv script. Read the first tutorial to do it.

Firstly, you need to create the project in an empty directory:

```
> mkdir tuto1
> cd tuto1
> gpnode newproject -n tuto1
```

After that, you should have a file named *tuto1.node* in the current directory. This file is the project file and contain the definition of the node. A node in GPStudio is a physical node, it can be a smart camera or a sensor. You can have only one project file per directory and gpnode always works on the project in the current directory. The directory name can be different that the project name, but you should not modify the name of the project file.

The next thing to do is to specify the platform that you want to use for this project. You can do it with:

```
> gpnode setboard -n dreamcam_c3
```

'dreamcam_c3' corresponds to a DreamCam platform with Cyclone III FPGA. The board support package for the 'dreamcam_c3' camera is located at:
<gps-root>/support/board/dreamcam_c3/dreamcam_c3.dev
DreamCam is now set as a target platform. You can check that with the command commandshowboard:

```
> gpnode showboard

dreamcam_c3
```

To have the complete list of supported board, use the gplib tool with the command **listboard**:

```
> gplib listboard

arrow_sockit de0nano dreamcam_c3 stratixcam_s4
```

2.2 Add IO support that you need

Until now, we only specify the Dreamcam as platform, but it is a modular one, you can use different types of image sensor and communication. We need to define witch one we want to use by adding the support of theses peripherals.

You can use this command to know the available devices for the platform that you have specified before:

```
> gpnode listavailabledevice

led mt9 e2v ethernet usb
```

For image sensor, you have two possibilities, mt9 from Aptina or e2v. For this example, we choose mt9:

```
> gpnode adddevice -n mt9
```

By adding mt9 IO, gpnode fetches the name of the driver to use with this IO and copies the hardware driver files implementation. Like that, it allows you to use this image sensor in order to acquire pictures.

Now, to enable a communication, you have the choice between Ethernet or USB. Choose USB support:

```
> gpnode adddevice -n usb
```

You can view the list of ios supports with the command **showio**:

```
> gpnode showdevice

ios :
  + mt9 [mt9]
  + usb [usb_cypress_CY7C68014A]
```

2.3 Connect block flow

To obtain a simple image sent from the image sensor to USB communication, we need a direct connection between 'mt9' output image flow and 'usb' input flow. To see flows that could be connected of a block, use the command **showblock** with a filter for looking only flows:

```
> gpnode showblock -n mt9 -f flows

flows :
    -------------
    |    mt9    |  out (8)
    |           |---------->
    -------------
```

And for usb block:

```
> gpnode showblock -n usb -f flows

flows :
            -------------
  in0 (8)   |           |  out0 (8)
---------->|           |----------->
  in1 (8)   |           |  out1 (8)
---------->|           |----------->
  in2 (8)   |    usb    |
---------->|           |
  in3 (8)   |           |
---------->|           |
            -------------
```

So we want to have this graph of connection:

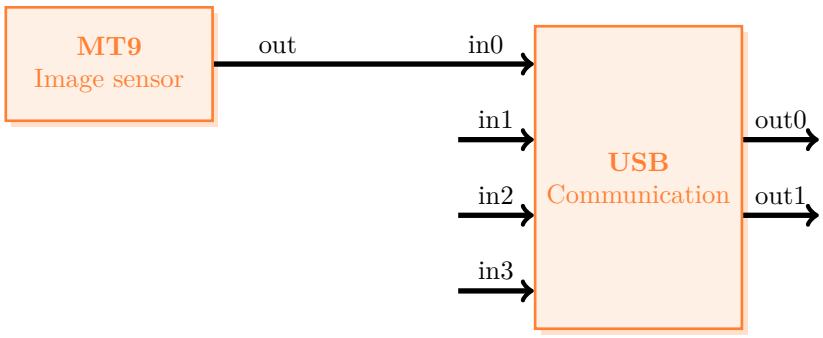

Figure 2: Details on flow connexion

To do it, you need to add a connection with the command **connect**:

```
> gpnode connect -f mt9.out -t usb.in0
```

To check effective connections, it is possible to print the list of connection with **showconnect**:

```
> gpnode showconnect

connects:
  + mt9.out -> usb.in0 (msb)
```

Your project is now fully configured. The next step is to generate code dedicated to the platform. We do it in a 'build' subdirectory:

```
> gpnode generate -o build
```

After that, a subdirectory 'build' is created in your project directory with the following files:

```
tuto1/
├── tuto1.node ..... project file
├── Makefile ..... Helper Makefile to call the built Makefile
└── build/ ..... output directory
    ├── top.vhd ..... generated top level
    ├── ci.vhd ..... generated CI block
    ├── pi.vhd ..... generated PI block
    ├── fi.vhd ..... generated FI block
    ├── node_generated.xml ..... definition of the internal node
    ├── Makefile ..... Makefile
    ├── Makefile.local ..... paths for Makefile
    ├── params.h ..... addresses of each internal registers
    └── IP/ ..... local copy of used IPs
        ├── IP1
        ├── IP2
        └── IP...
```

Figure 3: Files tree of the project and generated files

The produced Makefile offers you several commands:

- **generate**: regenerate the project if you made some modification
- **compile**: launch the compiler for your platform and produce a bitstream for the FPGA
- **send**: configure the connected camera with the bitstream
- **view**: launch the viewer

To launch the compilation process, call make with the rule 'compile'

```
> make compile
```

After few minutes, your project is now compile and ready to send to the camera.

First of all, powered up the DreamCam. Then, connect an USB cable from your computer to the USB communication board at the rear of the camera. Connect a second one to the internal JTAG near the power board.

To configure the FPGA with the generated bitstream, use the 'send' rule of the Makefile

```
> make send
```

This command needs only few seconds to be done. If no errors appear, camera is ready and you can launch the debugger viewer to test your project, **gpviewer**. The `make view` command launch gpviewer with the `node_generated.xml` file as argument. This file configure the interface. Let is try

```
> make view
```

gpviewer ask you which camera he need to open because it supports many camera with different types of connection at the same time.

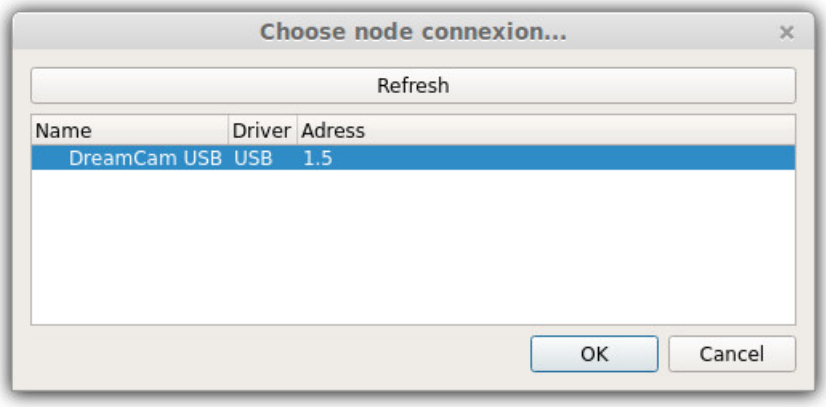

Figure 4: First windows of gpviewer to choose your camera

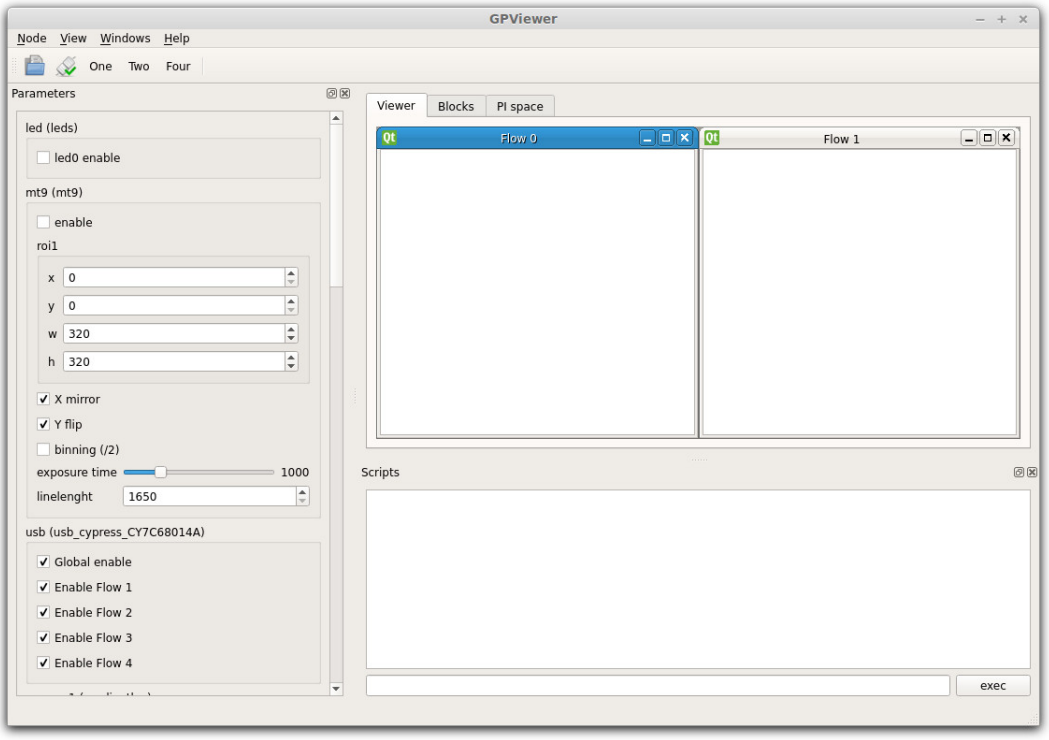

Figure 5: GPViewer main interface with blocks parameters on left and flow viewers on right

To enable the video just click on enable into mt9 block section. The video will appear. You can play with image sensor parameters like exposure time, the region of interest that modify resolution of the image or the direction of the picture.

3 Adding a process from GPStudio library

In this second part, we want to add some processing between the image sensor and the communication to obtain a smart camera and not only a 'webcam'.

Figure 6: Details on flow connexion

3.1 Add process that you need

Check process that could be use

```
> gplib listprocess

conv gradienthw histogramhw hog lbp normhw roi slidevm
```

For example, we add a gradienthw as process block and we call it 'process1'

```
> gpnode addprocess -n process1 -d gradienthw
```

To have the list of process in the project, just type

```
> gpnode showprocess

process :
  + process1 [gradienthw]
```

3.2 Connect process flow

Take a look to the flow name of the process

```
> gpnode showblock -n process1 -f flows

flows :
           ------------------
  in (16)  |                |  magnitude (16)
---------->|                |----------------->
           |    process1    |    angle (16)
           |                |----------------->
           ------------------
```

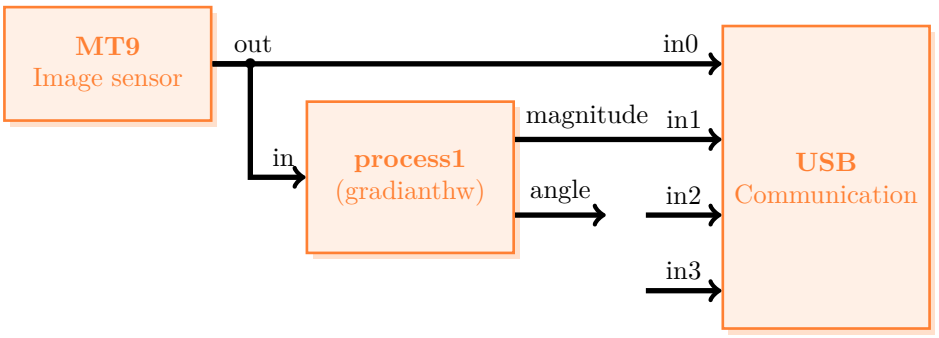

Figure 7: Details on flow connexion

```
> gpnode connect -f mt9.out -t process1.in
> gpnode connect -f process1.magnitude -t usb.in1
```

```
> gpnode showconnects

connects :
  + mt9.out -> usb.in0 (msb)
  + mt9.out -> process1.in (msb)
  + process1.magnitude -> usb.in1 (msb)
```

4 Creating simple custom process

A tool is done to create custom block process, **gpproc**. It allows to describe interfaces of your process and provide a template to fill with your code.

As example, we will create a simple image threshold process with an input image and an output result.

Figure 8: Details on flow connexion

4.1 Creating the process and defining the structure

In your node directory, create a new directory with the process name and add a process project

```
> mkdir mythreshold
> cd mythreshold
> gpproc new -n mythreshold
```

A new file with the *.proc* will appear in the current directory. This file represents the block process definition.

Now, we can add flow interfaces input and output respectively named 'in' and 'out'

```
> gpproc addflow -n in -d in -s 8
> gpproc addflow -n out -d out -s 8
```

You can check that with **showblock** command

```
> gpproc showblock

          ----------------------
  in (8)  |    mythreshold     |  out (8)
--------->|                    |---------->
          ----------------------
```

Generate template in the directory hdl

```
> gpproc generate -o hdl
mythreshold.vhd generated
mythreshold_process.vhd generated
```

Two files are provided by the block template generator.

`mythreshold.vhd` content:

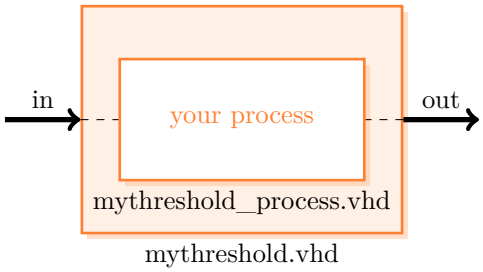

Figure 9: Details on flow connexion

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
library std;

entity threshold is
    generic (
        CLK_PROC_FREQ : integer;
        IN_SIZE       : integer;
        OUT_SIZE      : integer
    );
    port (
        clk_proc : in std_logic;

        ------------------------------ in flow ------------------------------
        in_data  : in std_logic_vector(IN_SIZE-1 downto 0);
        in_fv    : in std_logic;
        in_dv    : in std_logic;

        ------------------------------ out flow -----------------------------
        out_data : out std_logic_vector(OUT_SIZE-1 downto 0);
        out_fv   : out std_logic;
        out_dv   : out std_logic
    );
end threshold;

architecture rtl of threshold is
component threshold_process
    generic (
        CLK_PROC_FREQ : integer;
        IN_SIZE       : integer;
        OUT_SIZE      : integer
    );
    port (
        clk_proc : in std_logic;

        ------------------------------ in flow ------------------------------
        in_data  : in std_logic_vector(IN_SIZE-1 downto 0);
        in_fv    : in std_logic;
        in_dv    : in std_logic;

        ------------------------------ out flow -----------------------------
```

```
      out_data : out std_logic_vector(OUT_SIZE-1 downto 0);
      out_fv   : out std_logic;
      out_dv   : out std_logic
   );
end component;

begin
   threshold_process_inst : threshold_process
    generic map (
      CLK_PROC_FREQ => CLK_PROC_FREQ,
      IN_SIZE       => IN_SIZE,
      OUT_SIZE      => OUT_SIZE
   )
    port map (
      clk_proc => clk_proc,
      in_data  => in_data,
      in_fv    => in_fv,
      in_dv    => in_dv,
      out_data => out_data,
      out_fv   => out_fv,
      out_dv   => out_dv
   );

end rtl;
```

`threshold_process.vhd` content :

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
library std;

entity threshold_process is
   generic (
      CLK_PROC_FREQ : integer;
      IN_SIZE       : integer;
      OUT_SIZE      : integer
   );
   port (
      clk_proc : in std_logic;

      ------------------------------ in flow ------------------------------
      in_data  : in std_logic_vector(IN_SIZE-1 downto 0);
      in_fv    : in std_logic;
      in_dv    : in std_logic;

      ------------------------------ out flow -----------------------------
      out_data : out std_logic_vector(OUT_SIZE-1 downto 0);
      out_fv   : out std_logic;
      out_dv   : out std_logic
   );
end threshold_process;

architecture rtl of threshold_process is
```

```
begin
    data_process : process (clk_proc, reset_n)
    begin
        if(reset_n='0') then
            out_data <= (others => '0');
            out_dv <= '0';
            out_fv <= '0';
        elsif(rising_edge(clk_proc)) then
            out_dv <= in_dv;
            out_fv <= in_fv;

            if(in_dv='1' and in_fv='1') then
                out_data <= in_data;
            end if;
        end if;
    end process;
end rtl;
```

By default, the generated process file is a by pass. Replace the copy of the input pixel value to the output by a conditional statement to binarise the picture in `threshold_process.vhd`:

```
    if(in_dv='1' and in_fv='1') then
        if(in_data<std_logic_vector(to_unsigned(127, IN_SIZE))) then
            out_data <= (others => '0');
        else
            out_data <= (others => '1');
        end if;
    end if;
```

Add generated files to the project

```
> gpproc addfile -p hdl/mythreshold.vhd -t vhdl -g hdl
> gpproc addfile -p hdl/mythreshold_process.vhd -t vhdl -g hdl
```

Return to the node project and add your process

```
> cd ..
> gpnode addprocess -n mythreshold1 -d mythreshold/mythreshold.proc
```

Connect the input flow to the image sensor and the output to a free input flow of the USB communication block

```
> gpnode connect -f mt9.out -t mythreshold1.in
> gpnode connect -f mythreshold1.out -t usb.in2
```

Like that, you obtain
Test it

```
> make generate compile send view
```

4.2 Process with dynamic parameters

In this section, we will add the possibility of dynamically set the threshold during camera run with the dedicated interface.

Return in the process definition directory and create a property to set the threshold value of type int

```
> cd mythreshold
> gpproc addproperty -n threshold -t int
```

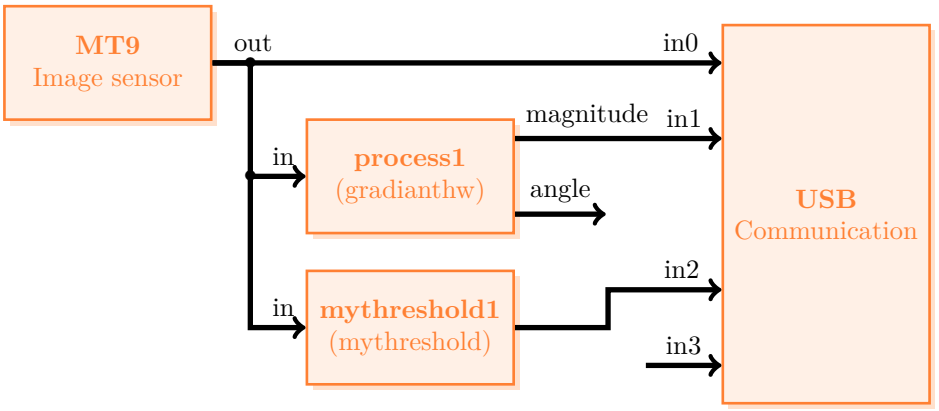

Figure 10: Details on flow connexion

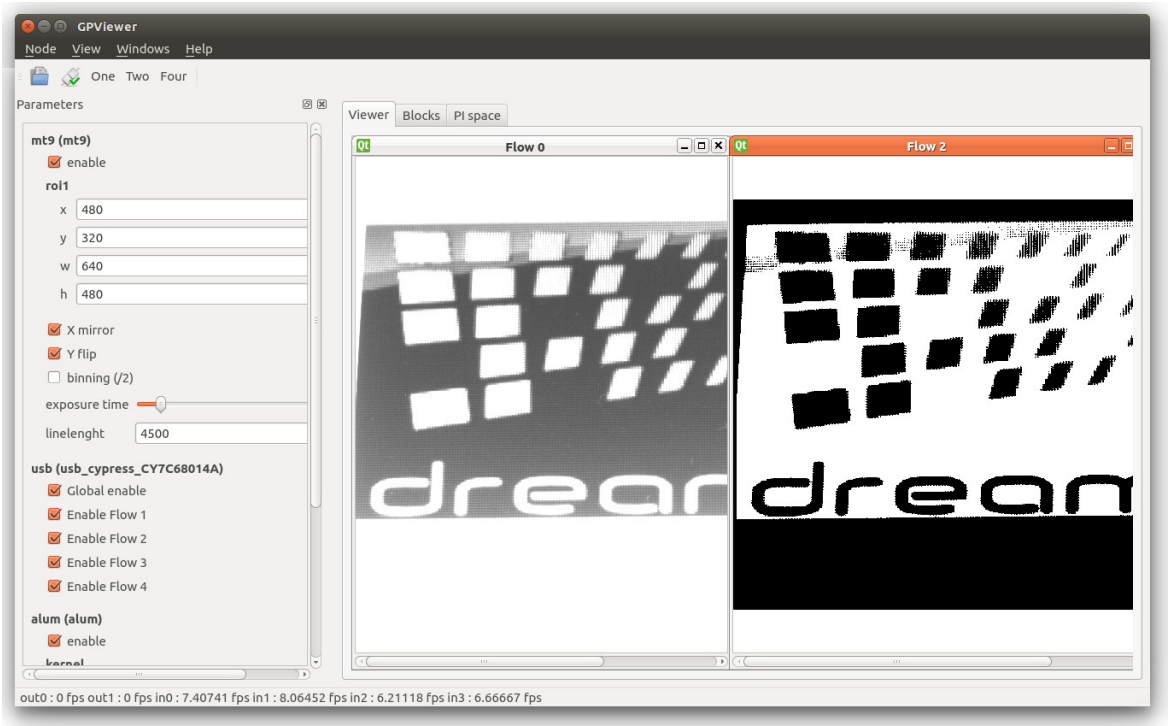

Figure 11: View of both flows, direct image sensor and result of threshold

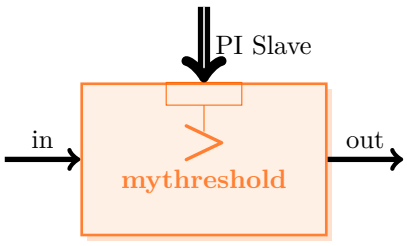

Figure 12: Threshold process block with an internal register externally driven

A property is a high level system (API, sofware, ...) and does not have any sense to the hardware level. At the low level, the only system that could be dynamically changed, is regis-

ter. A parameter is set by default as dynamic register, you just need to set a relative address and link it to the high level property with a property map expression. A property map could be expressed as a JavaScript expression and so could be very complex with conditional statements and/or mathematical functions. Here the property map is a simple link by direct value.

Add the register 'threshold_reg' with a relative address set to 1 and link the property value

```
> gpproc addparam -n threshold_reg -r 1 -m threshold.value
Warning (1) : Your relative adress is greater than the range of relative address (0 bits).
Please specify a new PI size address with :
gpproc setpisizeaddr -v 2
```

Set the size of PI address bus to 2 bits

```
> gpproc setpisizeaddr -v 2
```

Generate template in the directory hdl and add the slave block

```
> gpproc generate -o hdl
mythreshold.vhd generated
mythreshold_slave.vhd generated
mythreshold_process.vhd generated
> gpproc addfile -p hdl/mythreshold_slave.vhd -t vhdl -g hdl
```

A new component mythreshold_slave.vhd is generated because your process block have a PI slave interface. The new structure of components generated is described on this figure.

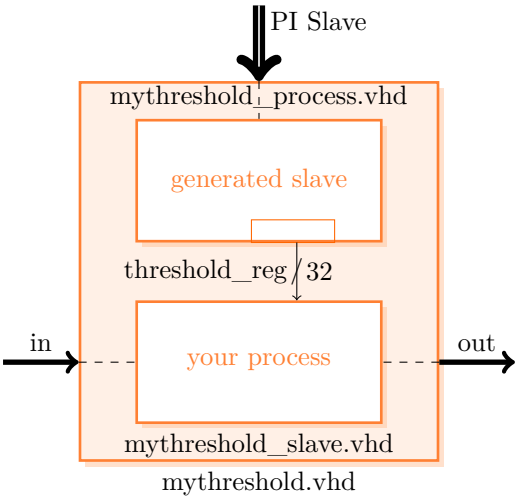

Figure 13: Details on flow connexion

To enable the threshold value that could change dynamically, get the 'threshold_reg' register value to compare with the pixel value insted of a constant value in `threshold_process.vhd`:

```
if(in_dv='1' and in_fv='1') then
    if(in_data >= threshold_reg(IN_SIZE-1 downto 0)) then
        out_data <= (others => '1');
    else
        out_data <= (others => '0');
    end if;
end if;
```

Test it

```
> cd ..
> make generate compile send view
```

To test dynamic parameter of your process, try to change the value of 'threshold' property. When the value change, the 'threshold_reg' register property map expression (here 'threshold.value') is re-evaluated. Result is written to the camera, you saw the effect on the next picture.

A visual test directly on camera with simple processes like this threshold could be enough but when you need a mathematical verification or performance evaluation, it is mandatory to launch tests on a set of known pictures. To achieve that, GPStudio propose two solutions, automatic test-bench and image sensor simulation.

4.3 Test-bench generation

GPStudio integrates an automatic test-bench generation in the same that generate skeleton for process. This test-bench generate clocks, send reset at the beginning and for each flow, generate a VHDL process. For an input, it open a file named '<flow-name>.stim' and send the content of this stimuli file to the flow interface. For an output flow, it will save all data to a file named '<flow-name>.data'.

In our case, the generated test-bench will send a picture to the 'in' flow and save data from 'out' flow.

```
> gpproc generatetb -o hdl
mythreshold_tb.vhd generated
Makefile generated
```

To create a stimuli file as input, gpproc provide a command to convert picture files (png, jpg, gif, bmp) to a stim file. Give the name of the input flow to the file.

```
> gpproc convert -i picture1.png -o in.stim
in.stim generated
```

Finally, you can launch the test with modelsim or GHDL. The first one could be used in full graphical mode but take a lot of memory and is difficult to automatize. The second one can be easily used in makefile and very small (40MB) but is more difficult to use at the beginning.

The Makefile generated is pre configured for GHDL. If you have already installed GHDL, just launch a simulation by launch Makefile. The output flow and waveform will be appear in the current directory.

4.4 Automatic test on hardware, hardware in the loop

Communication blocks in GPStudio can be use to read and write parameters, get flow from the camera but also to send flow **to** the camera. It very useful to send test image to evaluate your processing block.

A sample code in C++ is provided to code your own test. This code is located inside gui-tools/src/gphwloop.