IBM CAPI SNAP framework

Version 1.0

How is data managed in the SNAP environment?

This Guide describes how to use, declare and optimize data variables in the SNAP environment.

To understand without any ambiguity all the explanation provided in this document, let's recall the basics:

- an **application** is the program running on the host server and executed on a **CPU**
- an **action** is the program running on the **FPGA**

Overview

Even if you use a standard language like C or C++, porting a program on to an FPGA requires understanding some important concepts about how data flows or is accessed. The FPGA has no Operating System which means that standard use of data in code executed on a CPU will require, on an FPGA, a formal read or write access to the physical resource. Understanding this key point will help the user to define not only which **path** to use to exchange data between the application and the action, but also the **way** to access so that you can easily and efficiently access the data.

This document has been built using snap git release tag v1.3.2. It can certainly work also for new releases.

This document will successfully go through the following items:
- understand the ways for an **action** to exchange data with an **application**
- understand the ways for an **action** to exchange data with **all different resources**
- understand the control logic used by SNAP to manage different actions

Contents

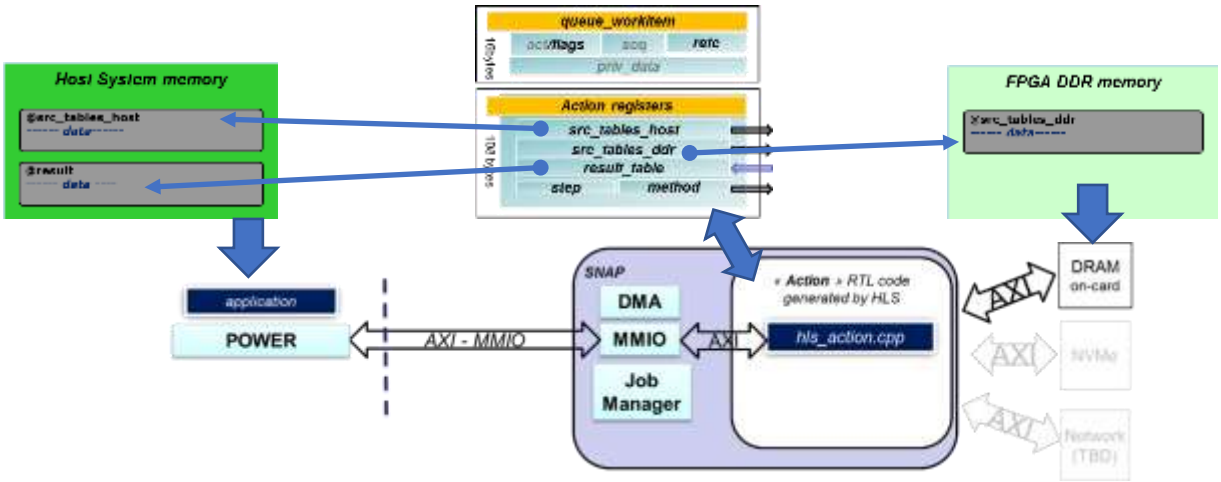
1. Actions interfaces

1.1 Configuration of an action by an application

The first key point to remember is that with CAPI, an action executed on the FPGA doesn't need a driver, which means that the application will NOT send the data to the action, since **the action will access data on its own**. The action will get information from the application as to where the required data is located.

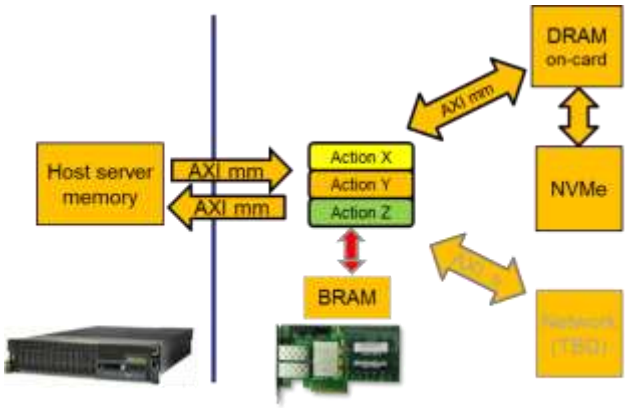
A second key point to remember is that, thanks to CAPI, when an address is defined and used in the application, **the same address will be usable without any modification in the action**.

In other words, let's take the example where a user wants to process 2 tables of data in the action, one located in the host server memory and one in the FPGA DDR memory. The application will fill a structure (Action registers) to tell the action where the required data is located, and where the action should write the result. Any other parameters can be added to control the action if needed.



This structure exchanged between the application and the action is prepared by the application and sent before starting the processing. We will use the "MMIO" interface to read or write these settings. They can be dynamically changed during the processing since this interface is asynchronous to all other interfaces.

The primary goal of the SNAP framework is to simplify the coding of the action which is implemented on the FPGA. Therefore, accessing several types of resources such as Host server memory, FPGA board DDR3 or DDR4, FPGA board Flash/NVMe memory needs to be standardized for the programmer by abstracting out the lower level details of each resource type. The other reason for this layer of abstraction is to prevent the user having to modify high level code when a new version of a resource type or a new resource type is added to the system.



A common standard and simple interface, named as AXI4 (Advanced eXtensible Interface release 4.0 developed by ARM), has been chosen to interface all the resources that an action needs to work with. This means that SNAP provides in its internal logic the driver to interface the different resources. If one (or more) of these resources is not used or not present on the board, the associated AXI driver to this resource can be disabled so that only used logic is implemented. Also, if a new resource is added to the system, the associated AXI driver can be added to the FPGA while the high-level user code stays as is, since it deals with the AXI abstraction layer.

As described in Xilinx UG761 document, this AXI4 protocol has 3 types:

- AXI (or AXI_mm) – for high performance memory mapped requirements
- AXI-Lite – for simple, low-throughput memory mapped communication (typically used for control and status registers)
- AXI-Stream – for high-speed streaming data

In SNAP, **AXI_mm** is used for all data that is exchanged with external resources. **AXI-Lite** is used for status and control registers (MMIO). **AXI-stream** is not used yet but is meant for future usage such as connecting with the network driver attachment as soon as it is available.

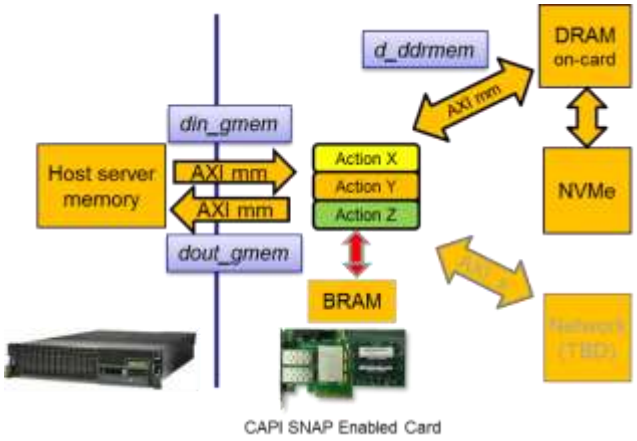
2. AXI data: the data "highway"

Let's start with the AXI data interface which allows data to be exchanged on a high-speed interface.

2.1 Type of resources

Three types of resources can be listed as:

- Common resources:
  - **Host server memory**: 1 unidirectional **port IN** + 1 unidirectional **port OUT**
- FPGA specific resources:
  - FPGA memory, named as **BRAM**: for example, a Xilinx KU060 has 3.5MB of memory used for user code variables.
- Board specific resources (for example):
  - AlphaData ADM-PCIE-KU3:
    - **8GB DDR3** – bidirectional port
    - 40Gb SQFP+ ports (future use)
  - Nallatech 250S-2T:
    - **4GB DDR4** - bidirectional port
    - **1.92TB NVMe** accessible via the DDR4



2 important points to highlight:

- **Action code accesses all resources using a unique AXI protocol** which abstracts out the specific protocols of each of the resources.
- **"Board resources" are NOT known or seen by the application** running on the server. These resources can be managed by the Host application or by the action but need to be managed by one or the other.

2.2 "Size" of the interfaces

To ease the use of these different resources by the action code, and reduce the latency to access them, the choice has been made in SNAP that all resources are accessed with a **data width set to a fixed value defined by "snap_membus_t"**. This data type is defined as follow:

~snap/actions/include/hls_snap.H:

```
#define MEMDW 512          // 512: Bus width in bits
#define BPERDW (MEMDW/8)   // Bytes per Data Word    if MEMDW=512 => BPERDW = 64

ap_uint<MEMDW> snap_membus_t
```

This means that a full word read from or written to the host server memory or the FPGA DDR will always be 512 bits / 64 Bytes data in physical size. We will see soon how to deal with this "constraint".

2.3 Ports declaration

All actions provided as examples in SNAP have the "**ports**" defined the same way. Remember that ***hls_action*** is a reserved name that is used by SNAP logic to find the top file of the code.

~snap/actions/hls_*/hw/action_*.cpp:

```
//--- TOP LEVEL MODULE ------------------------------------------------
void hls_action(
       snap_membus_t *din_gmem,
       snap_membus_t *dout_gmem,
       snap_membus_t *d_ddrmem, // CAN BE COMMENTED IF UNUSED
       snapu32_t *d_nvme,       // CAN BE COMMENTED IF UNUSED
…/…
```

All unused ports can be commented except the ports of the host memory declared as "**din_gmem**" and "**dout_gmem**" which needs to remain active. "**din_gmem**" is the unidirectional port through which all data read from the Host server memory will enter the action. "**dout_gmem**" is the unidirectional port through which all data written to the Host server memory will exit the action.

The port "**d_ddrmem**" is bidirectional which means that data read from or written to the FPGA board DDR will flow through this port. The port can address DDR3 or DDR4 types of memory since the SNAP logic will adapt automatically the right driver to the right type of DDR (card dependent).

The port "**d_nvme**" is a control port used to define the access to the NVME. All data are read and written through the DDR memory, used as a cache. Therefore, if the action needs data from the NVMe, it sends a read request of X bytes at a NVMe address, and the DDR will be filled with a block containing the data requested. The reason of this architecture is that the NVMe has a large latency and accessing it directly would slow the action program execution.

More explanation can be found in the document provided in snap NVME example:
https://github.com/open-power/snap/blob/master/actions/hls_nvme_memcopy/doc/UG_SNAP_hls_nvme_memcopy_v1.0.pdf

2.4 Ports definition

In the top-level module ***hls_action*** of each hardware action, after their declaration, all the ports are defined as "**`#pragma HLS INTERFACE m_axi`**" ports. This type of definition is used for data path using the AXI4 protocol:

```
//--- TOP LEVEL MODULE ---------------------------------------------
void hls_action(
…/…
    // Host Memory AXI Interface
#pragma HLS INTERFACE m_axi port=din_gmem bundle=host_mem offset=slave depth=512 \
  max_read_burst_length=64  max_write_burst_length=64
#pragma HLS INTERFACE s_axilite port=din_gmem bundle=ctrl_reg offset=0x030

#pragma HLS INTERFACE m_axi port=dout_gmem bundle=host_mem offset=slave depth=512 \
  max_read_burst_length=64 max_write_burst_length=64
#pragma HLS INTERFACE s_axilite port=dout_gmem bundle=ctrl_reg offset=0x040

// DDR memory Interface      //CAN BE COMMENTED IF UNUSED
#pragma HLS INTERFACE m_axi port=d_ddrmem bundle=card_mem0 offset=slave depth=512 \
  max_read_burst_length=64 max_write_burst_length=64
#pragma HLS INTERFACE s_axilite port=d_ddrmem bundle=ctrl_reg offset=0x050

    //NVME Config Interface //CAN BE COMMENTED IF UNUSED
#pragma HLS INTERFACE m_axi port=d_nvme bundle=nvme
```

Every port is defined with the name that will be used in the code to access it (din_gmem, dout_gmem, d_ddrmem, d_nvme). The declaration of the bundle is the prefix of the name of the signal which will be used by the logic to connect to the different physical interfaces through a wrapper. Modifying this bundle name will prevent your logic from being connected to the SNAP wrapper.

The parameters (slave_depth, max_read_burst_length, max_write_burst_length) are set for the Xilinx HLS tool to overwrite default settings and get the maximum AXI burst (4KB).

Refer to Xilinx UG902 Vivado HLS user guide for more details on these options.

2.5 Unused ports

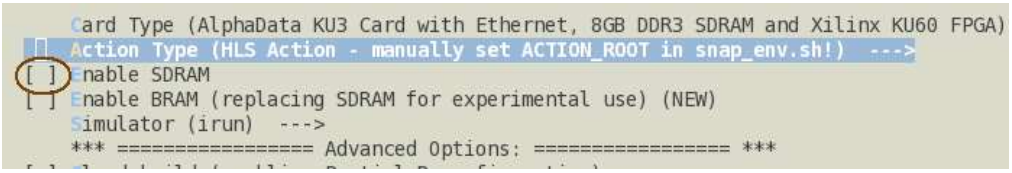
If the action code doesn't need any access to the DDR or the NVMe, then commenting these lines will remove the DDR driver and save place removing unnecessary logic.

```
//--- TOP LEVEL MODULE ---------------------------------------------
void hls_action(
       snap_membus_t *din_gmem,
       snap_membus_t *dout_gmem,
       /* snap_membus_t *d_ddrmem, // CAN BE COMMENTED IF UNUSED */
       /* snapu32_t *d_nvme,       // CAN BE COMMENTED IF UNUSED */
…/…
```

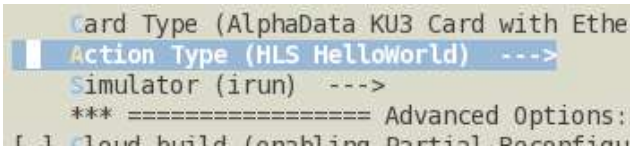
The comment must also be done in the port definition:

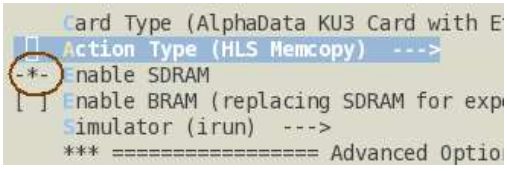
```
/*
// DDR memory Interface //CAN BE COMMENTED IF UNUSED
#pragma HLS INTERFACE m_axi port=d_ddrmem bundle=card_mem0 offset=slave depth=512 \
  max_read_burst_length=64  max_write_burst_length=64
#pragma HLS INTERFACE s_axilite port=d_ddrmem bundle=ctrl_reg offset=0x050
*/
/*
    //NVME Config Interface //CAN BE COMMENTED IF UNUSED
#pragma HLS INTERFACE m_axi port=d_nvme bundle=nvme
*/
```

Removing this port must **always** been done by also disabling the resource used in the **make snap_config** menu.



You will notice that all examples provided in SNAP (hls_helloworld, hls_memcopy, …) have non-selectable resources in the **make snap_config** menu so that we cannot select or unselect a resource by mistake. These preselected choices are described in ~snap/scripts/Kconfig and also per card and per action in ~snap/defconfig/ADKU3.hls_helloworld.defconfig or ~snap/defconfig/ADKU3.hls_memcopy.defconfig,…





When building your own action, the easiest way may be to pick as a template, an action which has the same interfaces that you need so that all the preselection job is already done.

You may have noticed in a previous paragraph that each port definition is followed by a **`#pragma HLS INTERFACE s_axilite`** line defining a register address for each port.

```
//--- TOP LEVEL MODULE -------------------------------------------------
void hls_action(
…/…
    // Host Memory AXI Interface
#pragma HLS INTERFACE m_axi port=din_gmem bundle=host_mem offset=slave depth=512 \
  max_read_burst_length=64  max_write_burst_length=64
#pragma HLS INTERFACE s_axilite port=din_gmem bundle=ctrl_reg offset=0x030
…/…
```

This address offset should not be used by the user code since it is forced by SNAP internal logic to 0 for all ports.

Read and Write are considered from the application / software side

Card init	Done once at card initialization by snap_maint program - Action has no access to perform				
Write@	Read@	3	2	1	0	Typical Write value
0x030		host_mem port din initialization (LSB)				0
0x034		host_mem port din initialization (MSB)				0
0x038		HLS reserved				
0x040		host_mem port dout initialization (LSB)				0
0x044		host_mem port dout initialization (MSB)				0
0x048		HLS reserved				
0x050		card_mem0 port ddr initialization (LSB)				0
0x054		card_mem0 port ddr initialization (MSB)				0
0x058		HLS reserved				

The reason is that ***Xilinx HLS uses only aligned address***. This means that with a data bus width set to 64 bytes, all the address passed through this register will be automatically converted to a 64 bytes address. To simplify things for the coder and reduce the latency taken by width conversions, all interfaces have been set to the same data bus width as explained in “2.2 “Size” of the interfaces”. This means that the user will use same alignment for all address.

To prevent from losing this byte address information, it has been decided to use a specific register that allows the action code to know which byte to address when reading a 64 bytes word. The caveat is that the conversion of this Byte address need to be done by the user before giving it to the function accessing memories.

Let’s explain deeper what is an aligned address.

Imagine that an application uses an address 0xABCD (in hexadecimal). This is a byte address. Every byte is associated to a specific address.

Reading a word at address **0xABCD** will give you a 1 Byte word **Byte_X**

Byte Address	1-Byte Data
…	
0xABD1	
0xABD0	
0xABCF	Byte_Z
0xABCE	Byte_Y
0xABCD	**Byte_X**
0xABCC	Byte_W
0xABCB	Byte_V
0xABCA	Byte_U
0xABC9	
0xABC8	
…	

1-Byte interface

2-Bytes interface

Byte Address	Data_Byte0	Data_Byte1	2-Bytes Address (Byte address >> 1)
…			…
0xABD0	Byte_Z		0x55E8
0xABCE	Byte_Y	Byte_Z	0x55E7
0xABCC	Byte_W	**Byte_X**	**0x55E6**
0xABCA	Byte_U	Byte_V	0x55E5
0xABC8			0x55E4
…			…

Now if you use a 2-Bytes interface, meaning that you can read 2 Bytes with a single address, then you have 2 choices:
- Read 2 Bytes at **0xABCC** will give you a 2 Bytes word **Byte_W** - **Byte_X**
- Read 2 Bytes at **0xABCD** will give you a 2 Bytes word **Byte_X** - **Byte_Y**

To save place and if we use only 2 Bytes access then removing all ***odd*** addresses will still let you have access to all Bytes. This is what we call a *2-Bytes aligned address*.
Thus, the new 2-Bytes address would be 0xABCC or 0xABCD shifted 1 bit to the right becomes **0x55E6.**
Having the Byte address information can let you work with the byte you expect in the 2-Bytes word read.

Extending that to our case, a 64-Bytes address needs the Byte address shifted 6 to the right (address 00h to 63h = address 00_0000b to 11_1111b)
Thus, the new 64-Bytes address would be 0xABCC or 0xABCD shifted 6 bits to the right becomes **0x2AF**

In the action code, you will so understand why an address sent by the application as a Byte address, will need to be aligned with the port width (`ADDR_RIGHT_SHIFT` is set to 6 and defined in file ~snap/actions/include/hls_snap.H)

```
    /* byte address received need to be aligned with port width */
    input_address = act_reg->Data.in.addr >> ADDR_RIGHT_SHIFT;
    output_address = act_reg->Data.out.addr >> ADDR_RIGHT_SHIFT;
```

The key point is that the coder can still work in the action and in the application with a Byte address. If he wants to access a word at a specific Byte address, then he has the information and can access the Xth Byte of the 64 words.

2.7 How to use ports

Now that the ports are declared with a name, data that flows through them need to be read or written at an address associated with the access request.

To read a data from host server memory, then the address used will be **din_gmem + input_address**.
To write a data to host server memory, then the address used will be **dout_gmem + output_address**.
To read or write a data from/to FPGA DDR memory, then the address used will be **d_ddrmem + address**.

Several ways can be used to read a data. The easiest way for a read:

```
snap_membus_t buffer[MAX_NB_OF_WORDS_READ]:
    switch (memory_type) {
        case SNAP_ADDRTYPE_HOST_DRAM:
            memcpy(buffer, (snap_membus_t  *) (din_gmem + input_address),
               size_in_bytes_to_transfer);
        break;
        case SNAP_ADDRTYPE_CARD_DRAM:
            memcpy(buffer, (snap_membus_t  *) (d_ddrmem + input_address),
               size_in_bytes_to_transfer);
        break;
    }
```

Same for a write:

```
snap_membus_t buffer[MAX_NB_OF_WORDS_READ]:
    switch (memory_type) {
        case SNAP_ADDRTYPE_HOST_DRAM:
            memcpy((snap_membus_t  *) (dout_gmem + input_address), buffer,
               size_in_bytes_to_transfer);
        break;
        case SNAP_ADDRTYPE_CARD_DRAM:
            memcpy((snap_membus_t  *) (d_ddrmem + input_address), buffer
               size_in_bytes_to_transfer);
        break;
    }
```

All data are stored in a buffer which can be used as a standard array of data.

You can also use type casting as for any C code:

```
    char text[BPERDW]; /*BPERDW (Byte per Data Word is defined as 64*/

    /* Read in one 64B word */
    memcpy((char*) text, din_gmem + input_address, BPERDW);
```

The function **memcpy** called is a HLS function which guarantee the most efficient access to the data in the way that:

- It allows to access any number of bytes you need not depending on the size of the word defined by *snap_membus_t* type. Requesting 1 byte will give you 1 byte!
- It will initiate a **burst** so that accessing a big amount of data stored at contiguous address will be done in a minimum amount of time. Accessing 4K bytes of data (maximum burst of AXI) in one shot will be much quicker than accessing 64 times 64 byte words.

There are other ways to access data which can be as efficient than the HLS **memcpy** but may lose one of these 2 above properties.

The first option is to use the closest translation of the **memcpy** function which keeps the **burst** property if no other processing is done within the loop, but force to read full **words of 64 bytes**:

```
     for (int k=0; k<size_in_words; k++)
#pragma HLS PIPELINE
            buffer[k] = (din_gmem + input_address)[k];
```

By extension, if you want to read or write a single word of 64 bytes, you can just use

```
      buffer[0] = (din_gmem + input_address)[0];
or
      (dout_gmem + input_address)[0] = buffer[0];
```

2.8 Optimizing data access

It is good to understand that Host memory access is done through a 128Bytes cache line. This means that:

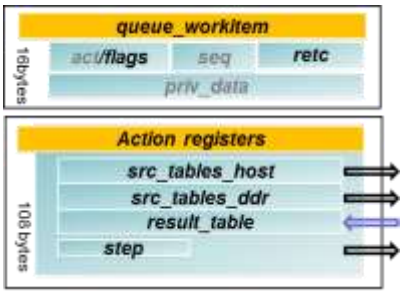
- Read 2 words from Host memory will give user the maximum bandwidth
- Write less than 2 words to host memory will use a “read-modify-write” access that may be not as efficient as writing 2 words which do not need a read access and the partial writes.

It may be more efficient to buffer data you need to use in the code prior to use them, rather than doing an access when you need the data. This would act as a cache and optimize the access.

3. AXI Lite data: the registers interface

Let's understand how to specify and use the MMIO interface used to exchange registers. As you may remember, AXI Lite resources are used for registers status and control using the MMIO interface.

```
typedef struct {
    CONTROL Control;      /* 16 bytes */
    action_job_t Data;    /* up to 108 bytes */
    uint8_t padding[SNAP_HLS_JOBSIZE - sizeof(action_job_t)];
} action_reg;
```



$ACTION_ROOT/hw/hw_action.H

3.1 Action registers: Control structure

The ***queue_workitem*** defined as the "***Control***" structure is the header of the action registers. It is managed by SNAP job manager and defined as follow:

$SNAP_ROOT/actions/include/hls_snap.H

```
typedef struct {
    snapu8_t sat; // short action type
    snapu8_t flags;
    snapu16_t seq;
    snapu32_t Retc;
    snapu64_t Reserved; // Priv_data
} CONTROL;
```

The action uses the ***flags*** field as a read only register to know if the action has been set already once (this is what we call the discovery mode).

The ***retc*** field is used as a write only register by the action to report the return code of the action.

$ACTION_ROOT/hw/hw_action.cpp

```
void hls_action(snap_membus_t *din_gmem,….
…/…
   switch (act_reg->Control.flags) {
    case 0:
        Action_Config->action_type = HELLOWORLD_ACTION_TYPE; //TO BE ADAPTED
        Action_Config->release_level = RELEASE_LEVEL;
        act_reg->Control.Retc = 0xe00f;
        return;
        break;
    default:
            process_action(din_gmem, dout_gmem, act_reg);
        break;
    }
```

On the application side, there is no access to the Control flag, but the **Return code** can be tested at any time:

$ACTION_ROOT/sw/snap_action.cpp

```
int main(int argc, char *argv[])
{
…/…
// test return code

    (cjob.retc == SNAP_RETC_SUCCESS) ? fprintf(stdout, "SUCCESS\n") :
         fprintf(stdout, "FAILED\n");
    if (cjob.retc != SNAP_RETC_SUCCESS) {
          fprintf(stderr, "err: Unexpected RETC=%x!\n", cjob.retc);
          goto out_error2;
    }
```

3.2 Action registers: Data structure

The "Actions registers" ***Data*** structure is where the user defines all parameters that will be used for a specific action. This structure is filled by the application for example as follows:

$ACTION_ROOT/include/action.H

```
typedef struct action_job {
    struct snap_addr src_tables_host;    /* input text in HOST: 128 bits*/
    struct snap_addr src_tables_ddr;     /* input pattern in DDR: 128 bits*/
    struct snap_addr result_table;       /* output result in HOST : 128 bits*/
    uint16_t step;
} action_job_t;
```

The smartest and easiest way is to use **snap_addr,** a SNAP predefined 16 Bytes (= 128bits) type, defined to specify in a minimum of bytes the exact location of the data to use. This type is defined as follow:

$SNAP_ROOT/software/include/snap_types.h

```
typedef struct snap_addr {
    uint64_t addr;
    uint32_t size;
    snap_addrtype_t type;       /* DRAM, NVME, ... */
    snap_addrflag_t flags;      /* SRC, DST, EXT, ... */
} snap_addr_t;
```

Rules to follow when defining the 108 Bytes of registers structure:

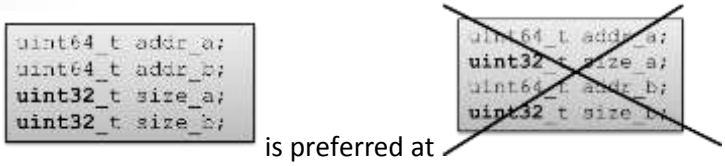
1. It has been experienced that the C compiler used by HLS prefer some simple definitions:

```
snap_addr table_src0;
snap_addr table_src1;
```

is preferable to

~~`snap_addr table_src[2];`~~
~~`...`~~

2. As the registers are defined as 64 bit registers, prefer 64 bit alignments rather than mixing sizes:



3. C compiler forces 64 bits alignment which generates SNAP errors

```
typedef struct helloworld_job {
struct snap_addr in; // input data => 16 Bytes
struct snap_addr out; // offset table => 16 Bytes
char is_the_problem; // => 1 Byte
} helloworld_job_t;
j = sizeof(helloworld_job_t); => reports 40 Bytes while it should report 33 Bytes!
```

Have a look at the issue described in https://github.com/open-power/snap/issues/488 which provides a way to circumvent this

```
typedef struct helloworld_job {
struct snap_addr in; // input data => 16 Bytes
struct snap_addr out; // offset table => 16 Bytes
char is_the_problem; // => 1 Byte
char padding[7]; // => 7 Bytes
} helloworld_job_t;
j = sizeof(helloworld_job_t); => reports 40 Bytes
```

Maximum Size:

The maximum size of the "Action registers" ***Data*** structure defined here as "**action_job**" has been set to 108 bytes.

$ACTION_ROOT/hw/hw_action.H

```
typedef struct {
    CONTROL Control;      /* 16 bytes */
    action_job_t Data;    /* up to 108 bytes */
    uint8_t padding[SNAP_HLS_JOBSIZE - sizeof(action_job_t)];
} action_reg;
```

Checking is done when compiling the (hardware) action: attempting to go beyond this limit will generate a compilation error.

```
Checking for reserved MMIO area during HLS synthesis ...
/home/…/snap//actions/hls.mk:69: recipe for target 'check' failed
```

If the definition of your structure is less than these 108 bytes a **padding** will be done so that this structure is always 108 Bytes. This constraint is due to the way Xilinx HLS tool manages registers, and the way we were constrained to use it so that it could friendly enough to use it.

In case you need more than 108 Bytes, it's very easy to define an area in host memory that will contain all the information you need to exchange between the application and the action.

Action's access to these registers:

The action will see these 108 Bytes of Data as a structure defined in *$ACTION_ROOT/include/action.H* and accessed as for any C structure:

$ACTION_ROOT/hw/hw_action.cpp

```
    /* byte address received need to be aligned with port width */
    i_idx = act_reg->Data.src_table_host.addr >> ADDR_RIGHT_SHIFT;
    o_idx = act_reg->Data.result.addr >> ADDR_RIGHT_SHIFT;
    size = act_reg->Data.src_table_host.size;
```

Application's access to these registers:

The application fills the Data structure using predefined SNAP APIs as follow:

$ACTION_ROOT/sw/snap_action.cpp

```
static void snap_prepare_action(struct snap_job *cjob…
{
    snap_addr_set(&mjob->in, table_host_addr, size_table_host, type_table_host,
              SNAP_ADDRFLAG_ADDR | SNAP_ADDRFLAG_SRC);
…/…
    snap_job_set(cjob, mjob, sizeof(*mjob), NULL, 0);
}

int main(int argc, char *argv[])
{
…/…
   // prepare params to be written in MMIO registers for action
    type_table_host = SNAP_ADDRTYPE_HOST_DRAM;
    table_host_addr = (unsigned long)ibuff;
…/…
    snap_prepare_action(&cjob, &mjob,
         (void *) table_host_addr, size_table_host, type_table_host,
         (void *) table_ddr_addr, size_table_ddr, type_table_ddr,
         (void *) result_addr, size_result, type_result);
…/…
}
```

It is important to notice that these **snap_addr_set** and **snap_job_set** APIs are not sending the data to the FPGA but just filling the data structure.

To read the result, just get it from the address you allocated and gave to the action:

$ACTION_ROOT/sw/snap_action.cpp

```
int main(int argc, char *argv[])
{
…/…
     /* If the result is in host DRAM we can write it to a file output */
     if (output != NULL) {
          fprintf(stdout, "writing output data %p %d bytes to %s\n",
                  result_addr, (int) size_result, output);
     }
}
```

SNAP management of Registers

SNAP is in-between the application and the action and its logic manages all the registers address so that it becomes transparent for the user.

For each action, SNAP gives a specific offset to add to the registers structure so that each action can have its set of registers independently.

Each register structure is then used as follows:

- The action accesses the registers as a single read/write structure.
- The application accesses the registers as 1 read only structure and 1 write only structure.

For example, when accessing the registers of an action, the SNAP application's logic writes to:

action_offset + 0x100 + register address

but will read the value at address

action_offset + 0x180 + register address.

Read and Write are considered from the application / software side

act_reg.Control	***This header is initialized by the SNAP job manager. The action will update the Return code and read the flags value.***								
CONTROL	***If the flags value is 0, then action sends only the action_RO_config_reg value and exit the action, otherwise it will process the action***								
Simu - WR	Write@	Read@	3	2	1	0	Typical Write value	Typical Read value	
0x3C40	0x100	0x180	sequence		**flags**	short action type	f001_01_00		
0x3C41	0x104	0x184	**Retc** (return code 0x102/0x104)				0	0x102 - 0x104	SUCCESS/FAILURE
0x3C42	0x108	0x188	Private Data				c0febabe		
0x3C43	0x10C	0x18C	Private Data				deadbeef		

action_reg.Data	***Action specific - user defined - need to stay in 108 Bytes***							
memcopy_job_t	***This is the way for application and action to exchange information through this set of registers***							
	Write@	Read@	3	2	1	0	Typical Write value	Typical Read value
0x3C44	0x110	0x190	snap_addr.**addr_in** (LSB)					
0x3C45	0x114	0x194	snap_addr.**addr_in** (MSB)					
0x3C46	0x118	0x198	snap_addr_in.**size**					
0x3C47	0x11C	0x19C	snap.addr_in.**flags** (SRC, DST, …)		snap.addr_in.**type** (HOST, DRAM, NVME,..)			
0x3C48	0x120	0x1A0	snap_addr.**addr_out** (LSB)					
0x3C49	0x124	0x1A4	snap_addr.**addr_out** (MSB)					
0x3C4A	0x128	0x1A8	snap.addr_out.**size**					
0x3C4B	0x12C	0x1AC	snap.addr_out.**flags** (SRC, DST, …)		snap.addr_out.**type** (HOST, DRAM, NVME,..)			

The offset 0x100 is defined in:

$ACTION_ROOT/hw/hw_action.cpp

```
void hls_action(snap_membus_t *din_gmem,…
…/…
    // Host Memory AXI Lite Master Interface - NO CHANGE BELOW
#pragma HLS DATA_PACK variable=act_reg
#pragma HLS INTERFACE s_axilite port=act_reg bundle=ctrl_reg offset=0x100
#pragma HLS INTERFACE s_axilite port=return bundle=ctrl_reg
```

. The user doesn't need to understand the offset management to read or write a register. This offset is "built-in" in SNAP and shouldn't be modified, or this will prevent access of registers

4. Control Registers

Some control registers have also been set to manage the action. These explanations are provided to be exhaustive but are not needed to program an action.

4.1 Control registers used for discovery mode

The SNAP can handle several actions. To be able to know what to manage, before the first time SNAP can work with an action, it needs first to do the inventory of all the actions of a card.

On an application point of view, the user must run almost once the command ***snap_maint -v -C0*** to get a list of all the actions related to the card plugged in slot C0.

$SNAP_ROOT/software/tools/snap_maint –v

```
INFO:Connecting to host port 16384
SNAP FPGA Release: 1/0:0 Distance: 86 GIT: 0xd7960e4e
SNAP FPGA Build (Y/M/D): 2017/7/12 Time (H:M): 12:12
SNAP FPGA CIR Master: 1 My ID: 0
SNAP FPGA Up Time: 0 sec
SNAP FPGA Exploration already done (MSAT: 1 MAID: 1)

   Short |  Action Type |   Level    |
  -------+--------------+------------+-----------
     0      0x10141000      0x00000021  HLS Memcopy
```

The processing done is a 3 steps process which will:

- Query available action types
- Query number of actions
- Enable actions

The resulting table is kept in memory to connect an action to an application

Index	Action Types	Level	Short
0	MEMCOPY	0x21	0
1	MEMCOPY	0x21	0
2	CYPHER	0x2	1
3	SEARCH	0x4	2
...	...		...
15	...		...

On the action side, the code is pre-written to return the Write-only **Action_Config** registers values if the action registers have never been set (Control.flags= 0).

These **Action_Config** registers must **never be read by the action** or this will generate a discrepancy in address definition.

$ACTION_ROOT/hw/hw_action.cpp

```
void hls_action(snap_membus_t *din_gmem,…
…/…
    // Host Memory AXI Lite Master Interface - NO CHANGE BELOW
#pragma HLS DATA_PACK variable=Action_Config
#pragma HLS INTERFACE s_axilite port=Action_Config bundle=ctrl_reg offset=0x010
…/…
    switch (act_reg->Control.flags) {
    case 0:
        Action_Config->action_type = HELLOWORLD_ACTION_TYPE; //TO BE ADAPTED
        Action_Config->release_level = RELEASE_LEVEL;
        act_reg->Control.Retc = 0xe00f;
        return;
        break;
```

4.2 Control registers used for action internal logic

Xilinx HLS generates some internal logic and registers to handle the action. This logic is not accessible within the action code but will be used by SNAP and the application code to handle the action.

As for previous access, SNAP has a read and a write address to access these registers. They are defined in the *hls_action_ctrl_reg_s_axi.vhd* file generated by HLS as

```
-- ------------------------Address Info-------------------
-- 0x000 : Control signals
--         bit 0  - ap_start (Read/Write/COH)
--         bit 1  - ap_done (Read/COR)
--         bit 2  - ap_idle (Read)
--         bit 3  - ap_ready (Read)
--         bit 7  - auto_restart (Read/Write)
--         others - reserved
-- 0x004 : Global Interrupt Enable Register
--         bit 0  - Global Interrupt Enable (Read/Write)
--         others - reserved
-- 0x008 : IP Interrupt Enable Register (Read/Write)
--         bit 0  - Channel 0 (ap_done)
--         bit 1  - Channel 1 (ap_ready)
--         others - reserved
-- 0x00c : IP Interrupt Status Register (Read/TOW)
--         bit 0  - Channel 0 (ap_done)
--         bit 1  - Channel 1 (ap_ready)
--         others - reserved
```

That can also be represented as follow:

HLS_action_control	Control signals to start an Action and/or set/clear an IRQ							
	Generated by HLS in => /snap/actions/hls_helloworld/hw/hlsUppercasexxx/helloworld/syn/vhdl/hls_action_ctrl_reg_s_axi.vhd							
Write@	Read@	3	2	1	0	Typical Write value		Typical Read value	
0x00	0x00	Control signals				xxxxxxx1	Action Start	xxxxxxx2	Action done
0x04	0x04	Global Interrupt Enable Register				xxxxxxx1	IRQ enabled		
0x08	0x08	IP Interrupt Enable Register (Read/Write)				xxxxxxx1	"Done" IRQ		
0x0C	0x0C	IP Interrupt Status Register (Read/TOW)						xxxxxxx1	Clear "Done" IRQ

These control signals are generated by the port=return instruction that has been added to the act_reg register definition.

$ACTION_ROOT/hw/hw_action.cpp

```
void hls_action(snap_membus_t *din_gmem,…
…/…
    // Host Memory AXI Lite Master Interface - NO CHANGE BELOW
#pragma HLS DATA_PACK variable=act_reg
#pragma HLS INTERFACE s_axilite port=act_reg bundle=ctrl_reg offset=0x100
#pragma HLS INTERFACE s_axilite port=return bundle=ctrl_reg
```

The application will set and manage the start of the action, the set and clear of the interrupt with specific APIs defined in *~snap/sotware/lib/snap.c* such as ***snap_action_start*** or ***snap_action_completed***