I_2O core and Xdaq-shell frameworks

E. Cano *CERN, Geneva, Switzerland*

Abstract

This document describes the goal, architecture, and uses of the xdaq-shell (a user space-kernel space generic communication scheme) and the i2o-core (a generic API for hardware drivers development).

Keywords: I_2O, Hardware, bus, PCI, library, API, kernel space, user space, Linux, VxWorks, DMA, VME

1 Introduction

In the current CMS data acquisition (DAQ) prototyping work, hardware prototypes have to get integrated in software frameworks (slow control, fast control, high level triggers, etc.), and the hardware abstraction layer of various Operating Systems (OS) is not well standardized. As those pieces of hardware get tested and used in various environment, it is quite convenient to define an environment that would allow the user to use the same software in various those various environments.

1.1 Architecture

I_2O core is based on a portion of the I_2O specification. The I_2O specification is defined by the I_2O special interest group.

The **I_2O core** is a hardware abstraction layer that allows to write low-level pieces of software in a standardized fashion. The **I_2O core** notably handles all bus-related actions, including placing a transaction on the bus, detecting adapters and handling adapter allocation, interrupt handling. It also adds support for common tasks in device driver, like event queues, threads and semaphores.

The **Xdaq-shell** is a communication layer designed for communicating between user space and kernel space. This layer is required for enabling communication between high level applications (slow control applications, run control applications) and low level pieces of software (controls for specific boards). This layer wouldn't be necessary using an real time operating system like VxWorks, so its implementation will be quite different depending on which platform it is used on. However, the API for using the Xdaq-shell from both side (**I_2O-core** side and application side) is the same in all cases. See Figure 1 for an example in the context of an Xdaq (software-only DAQ) application.

This kernel space-user space communication is required because one wants to have communication between drivers (i2o core parts, that drives hardware) and high level (networked, etc...) applications.

The **Xdaq-shell** and **I_2O core** layer are seen from the user point of view as three separate APIs. **I_2O core** itself is the first API. This API defines prototypes of functions accessing the low level (hardware).

Xdaq-shell allows access to the "driver" (here meaning a set of kernel-level functions) from the user space,and the opposite (access from kernel space to user space). This implies one API for the kernel side and one API for the user side. In order to optimise the performance, it is better to avoid contexts switches as much as possible. Therefore, the **Xdaq-shell** is based on message passing schemes (all the parameters of an action are passed to the driver / sent back to the user at once). Those two API will contain message passing functions (send and receive), functions for allocating message, and function for resolving addresses. The **Xdaq-shell** takes care of the routing of the messages.

In order to be able to access DMAable memory from user-space program, the **Xdaq-shell** provides support for DMAable buffers (Dbuffs). The also provides support for direct mapping of the PCI memory space in user space. All those advanced features are not needed inbasic designs but can help to increase the performance when this is required.

The various schemes that can be used to increase performance (especially in Linux systems where the use of system calls costs non negleglectable time will be in Appendix C: Performance critical designs).

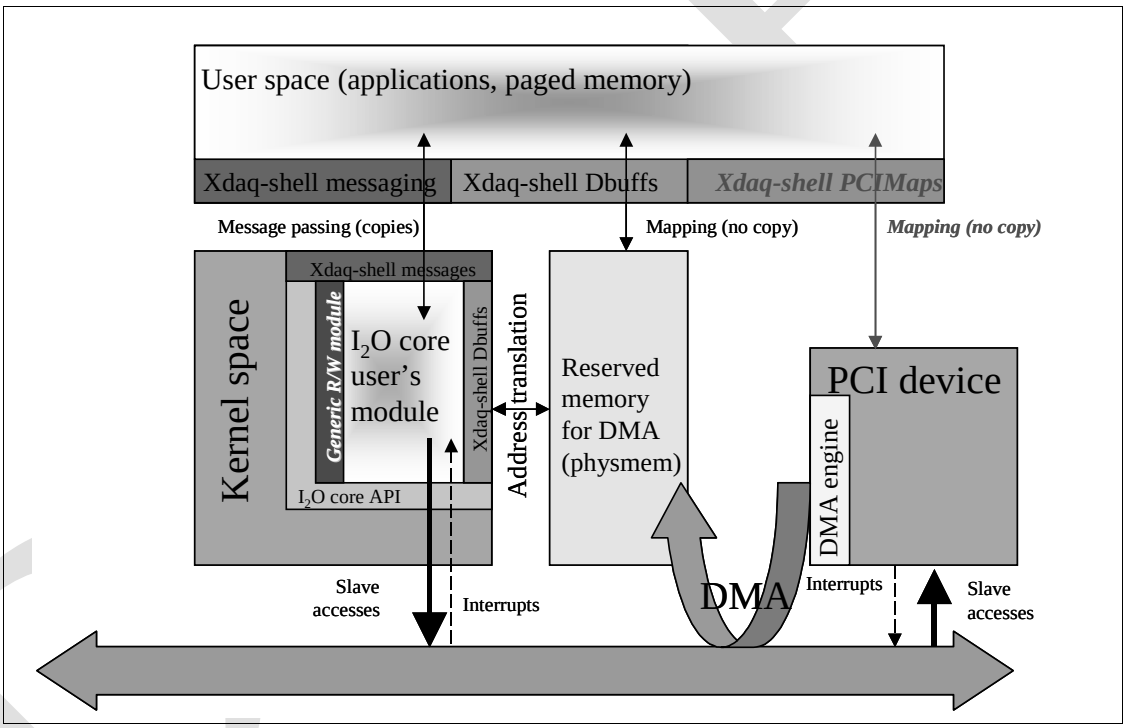

Figure 1: Kernel and user space layout in I_2O core and Xdaq-shell applications. The red parts are not yet implemented

1.2 Supported platforms and buses

For now the first prototypes are intended to be delivered for Linux and VxWorks. The target bus for the first version will be the PCI bus. Following version might support other buses as needs arise.

2 I_2O-core

As stated above, the goal of the I2O core is to provide low level access to the different adapters present in a system. In the first versions, it is limited to the PCI devices. It is possible to include extensions to VME bus in the future.

2.1 Original specification

The original specification defined an Intelligent Realtime OS (IRTOS). The API to the system calls of this system are called I2O-core. Our work is based on the I2O-core definition from Intelligent I/O Architecture Specification Version 2.0 [1]. This implementation is mainly based on chapter 5 of this document, and more precisely on sub-chapter 5.4 : IRTOS: I_2O Real-Time OS. The specification defines various concepts of Device Driver Modules, Executive, Code loading, etc. that we won't be using here. The only part of the specification used in this development is the API to low level functions.

2.2 CMS-DAQ implementation and adaptation

The CMS-DAQ version of **I_2O core** contains the following parts of the SIG version:

- Event queues (as defined in [1] 5.4.2.4)
- Threads (as defined in [1] 5.4.18)
- Buses (as defined in [1] 5.4.11)
- Adapters(as defined in [1] 5.4.12)
- Interrupts(as defined in [1] 5.4.14)
- Semaphores (as defined in [1] 5.4.20)
- Busy wait (as defined in [1] 5.4.19)

2.2.1 Object ID and object handling

The **I_2O core** handles all objects in a standard way, much like in an object-oriented system. All objects inherit from the base object. In order to simulate this the structures contain pointers to their respective "member" functions in a generic header. The user doesn't have to handle this part, normally. All object is owned by another object. When the owner object is destroyed, all the owned objects are destroyed as well automatically. The user can also destroy an object explicitly by using the `i2oObjDestroy` function. All the functions in the `i2oObj*` series work on all object IDs.

The user can also bypass the owner handling by using a NULL owner ID, but in this case, he has to handle all the destructions by hand. (Instead, he can create a first object with owner ID zero (ususally the adapter), and then make it owner of all the other ones. Then, calling `i2oObjDestroy` on the first one will automatically destroy all the owned objects.

We can add here that the `i2oExecutive*` functions return object with no owner. This object (the adapter) and be used as the owner of all your other objects. Therefore all the object can be destroyed in one operation by releasing the adapter. This avoids the tracking of too many object IDs.

2.2.2 Internal implementation

Depending on the system on which we run, we have C++ support or not. The easiest version is with VxWorks, as the IRTOS API defined by the I_2O SIG matches the VxWorks API very closely, and VxWorks supports C++, so we can implement the object concepts of **I_2O core** directly in C++.

Under Linux, we have to use tricks in order to implement the object likeliness. This is done through a private head that is common to all structure. The head is a struct containing pointers to all the functions handling that special object, in a fashion close to the one used in virtual tables. The user doesn't have to worry about those internal details, which are handled through the `i2oObj*` functions.

Internally, each object type has to have a specific destructor function with a skeleton identical to this one :

```
void i2oSemDestroy (I2O_OBJ_ID objectID, I2O_STATUS * pStatus)
{
    I2O_SEM_ID semID = (I2O_SEM_ID) objectID;
    while (down_trylock(&semID->sem))
         up (&semID->sem);
    i2oObjDefaultDestroy (objectID, pStatus);
}
```

Note that only specific action is required, then one just need to call
`i2oObjDefaultDestroy (objectID, pStatus);`

The default destroy function handles everything linked to the common headers in all objects, including destroying owned objects.

A user who doesn't want to change or extend the dehaviour of objects in **I_2O core** needs not to take care of the internal implementation.

2.3 API

The API to the various components of the **I_2O core** match as much as possible the API defined in the standard header files provided by the **I_2O SIG**. The implementation matches well the API for event queues, threads, buses, interrupts and semaphores. However, a shortcut had to be developped to replace adapter allocation system from the standard I_2O (involving message passing between Device Driver Modules and IRTOS's executive) by two simple function calls. This shortcut removes a whole lot of overhead that involved message passing and a quite heavy protocol during the loading of the device driver module, a concept of I_2O that we don't use here. The resulting API is described thereafter. If anything is unclear in this definition, the user should use the standard **I_2O Architecture Specification** version 2.0 and the C header files for this version.

A simple axample of use of the **I_2O-core** can be found in Appendix B: Basic design with i2o-core and Xdaq-shell. All the functions defined in this implementation is I_2O core are listed in Appendix A : Functions summary.

All the functions in the **I_2O-core** specification accept as their last argument a pointer to an `I2O_STATUS` variable. If the pointer is non-NULL, the i2o function should only report the error in this variable. The i2o function doesn't set it to I2O_STS_OK in case of success, so that the programmer can call several i2o functions in a row, and check for successful

completion only at the end. The progammer can also pass I2O_NO_STATUS as the pointer. This informs the i2o function that he won't check for an error and that it's up to the i2o function to handle the error. This mecanism is explained extensively in I_2O specification, in paragraph 5.4.2.1.2 .

2.3.1 Generic Objects

The following functions are available for each object IDs (and `i2oObj*` operation can be applied to any ID (`I2O_ADAPTER_ID`, `I2O_BUS_ID`, etc...). Any ID can also be used as an `I2O_OBJECT_ID` (by casting it). For example, if the user wants to destroy any object, he just has to call `i2oObjDestroy ((I2O_OBJECT_ID) myObjectId)`.

In the I_2O core, all object have an owner (the owner can also be NULL). Owner is usually set at creation time. It can be modified at anytime using the function i2oObjOwnerSet. Like i2oObjDestyroy, this function works on all the object types through a cast.

The user should be aware that by destroying an object, he also destroys all the objects owned by this one. (Ownership relations create a tree organisation).

2.3.2 Adapter

In the I_2O-core semantics, and adapter is a device on a bus. From now on, an adapter is a PCI board, but this concept might be extended to VME boards. In a program using I_2O-core, the first action is usually to allocate an adapter and get the first reference to it. Those routines are the main shortcut added to the specification.

```
I2O_ADAPTER_ID i2oExecutivePciAdapterAttach(U16 PciVendorID, U16 PciDeviceID,
int  index, I2O_STATUS* pStatus);
void i2oExecutiveAdapterDetach (I2O_ADAPTER_ID id);
```

Those two functions allow the driver to allocate an adapter(and therefore access the function attached to it), and to release the adapter. In the case of extension of this to VME bus, some functions would be added here in the API, to allow configuration of VME boards in the executive (VME boards are not auto detected like PCI boards).

The adapter object also allows accesses to configuration functions:

```
i2oAdapterBusGet
i2oConfig{Read|Write}{8|16|32|64}
i2oAdapterIntLock
i2oAdapterIntUnlock
i2oAdapterPhysLocGet
```

See paragraph 5.4.12 for functions details.

2.3.3 Interrrupt

Interrupts are available through the adapter object. One just has to attach an interrupt service routine handler to the adapter. The function are defined in paragraph 5.4.14 of I_2O specification.

2.3.4 Bus

The accesses to buses are defined in paragraph 5.4.11 of the I_2O specification. The API of our implementation contains the following functions :

```
i2oBusLocal
i2oBusSystem
i2oBus{Read|Write}{8|16|32|64}
i2oBusTranslate
```

The two first functions return an ID of the local bus and of the system bus.

The “translate” function allows tranlation of addresses on one bus as seen from another one, in order to allow simple programming of DMAs.

2.3.5 Thread

The **I_2O-core** library allows the user to spawn threads in its driver, but usually, the thread spawned automatically by the event queue is enough to handle the requests comming from the user, or to handle large parts of processing required after an interrupt.

The API is the following (some of the functions defined in I_2O specification are not implemented yet):

```
i2oThreadCreate
i2oThreadDelay
i2oThreadIdSelf
i2oThreadLock
i2oThreadUnlock
i2oThreadPri{Get|Set}
```

The corresponding paragraph in the specification is paragraph 5.4.18.

2.3.6 Event Queue (EventQ)

Event queue is just a process waiting on a FIFO, and looping indefinitely waiting for events to come.

It is quite important, because this loop is usually the heart of the driver, waiting for user interaction (possibly through **Xdaq-shell**). Typically, the interrupt service routine will handle the low level things in order to get the interrupt cleared and then post the remaining of the work to the driver’s event queue. Alternatively, it can also post a reply directly to an **Xdaq-shell** file.

It is defined in the I_2O specification in paragraph 5.4.2.4.

The functions are the following:

```
i2oEventQCreate
i2oEventQThreadGet
i2oEventQPost
```

The functions defining priorities of events in queues are not implemented in the present version. The event queue has a FIFO behavior.

2.3.7 Semaphore

Semaphores are also available. The are defined in specification paragraph 5.4.20. There are three types of semaphores: binary semaphores, ounting semaphores and mutual exclusion (mutex) semaphores.

The functions in the API are :

```
i2oSem{B|C|M}Create
i2oSemTake
i2oSemGive
```

The timeout functionnality is not available on Linux for now. Only immediate return and infinite wait are available for the moment.

2.4 Performance impact

Performance impact of the usage of **I_2O-core** compared with native drivers (direct calls to OS functions) will be studied once the **I_2O-core** will be ready on each platform.

3 Xdaq-shell

3.1 Architecture

The **Xdaq-shell** is defined to be a communication layer between user space and kernel space, when this distinction applies. Performance has to be as good as possible. To minimise overhead of kernel-user and user-kernel context switches, it is better to use a message passing scheme. The message structure in the **Xdaq-shell** is composed of two parts. First part is the routing layer, and is handled by **Xdaq-shell**. It is just a fixed-size header added to the messages. The structure of the messages is architecture dependant, so the user should not rely on the internal layout of the i2o_msg structure. The other part is totally defined by the user, usually also as a structure.

The current implementation is based on fixed-size FIFOs. This could lead to messages loss if some FIFOs become full.

3.2 API

All functions and structures defined in the **Xdaq-shell** start with “xdsh”.

3.2.1 Module strucuture

Each object file created by the user has to register a service to the **Xdaq-shell**, and to unregister when the module is unloaded. This is done thanks to two special functions : xdsh_init and xdsh_cleanup. Thos two functions, created by the user, typically contain the registration of the callback functions. See the example in Section 3.3, "Example programs".

Also see Section 4, "User’s manual" for details specific to each plateform (compilation options, etc.).

3.2.2 Message structure

The messages transported by the **Xdaq-shell** are very simple and contain two parts. First part is a fixed size header, which is architecture dependant, and second part is completely up to the user. This second part, the payload, can be accessed through a function call.

3.2.3 Message allocation and handling functions

The messages are allocated and deallocated by using special function calls. This allows the **Xdaq-shell** to handle messages differently according to the system it is used on. Therefore, the user has to consider a message as deallocated as soon as it has been sent, and the library automatically allocates the message frame of the right size when the user receives the message. On the other hand, it's up to the user to deallocate a message he received through a "get_message" function. This scheme allows us to pass pointers directly on systems with no memory protection (namely, VxWorks).

The message structures are allocated and de-allocated as folows :

```
struct xdsh_msg * xdsh_alloc (u32 size);
void xdsh_free (struct xdsh_msg *);
```

and by all the message passing functions.

Those functions work as usual, and `xdsh_alloc` returns `NULL` in case of failure during allocation.

The user should only be interested in the size of the payload and in the payload itself. The payload would typically contain a structure defined by the user. Functions for accessing the payload are:

```
void * xdsh_msg_payload (struct xdsh_msg *);
u32 xdsh_msg_size (const struct xdsh_msg *);
```

Parameter `size` is the size of the *payload* in bytes. The maximum theoretical size of a message is therefore 4 Gigabytes, which if far above what the computer system can allow in pratice.

A typical use of this scheme is shown on Figure 2. This kind of use is valid in both user space and kernel space.

```
struct xdsh_msg *message = xdsh_alloc (sizeof (struct device_specific));
struct device_specific *payload =
    (struct device_specific *) xdsh_msg_payload (message);
device_specific_init (payload);
payload->field1 = some_value;
payload->field2 = yet_another_value;
```

Figure 2: Code example: Message allocation and payload access in **Xdaq-shell**

TODO : evaluation of the practical maximum sizes of messages in different situations.

3.2.4 Address resolution functions

When the user wants to access the driver functions (send command / receive replies), he should identify itself to the Xdaq-shell. This corresponds to opening a file descriptor to the driver. This creates and "access point" through which message are send, and more important replies are sent back. A program can creat many of those. A reply to a command will always come back to the same `xdsh_file`. The function used for that is:

```
struct xdsh_file * fd = xdsh_open ();
void xdsh_close (struct xdsh_file * fd);
```

The open function may return NULL on failure.

When the user wants to access a function, he should first get the address for this function, this is done through the xdsh_file structure:

```
xdsh_srvc_addr xdsh_get_address (struct xdsh_file * file,
    const char *service_name);
```

The service name is a string defined by the user, and registered by the driver program. The driver program has to provide a callback function that handles the message (either directly or by posting it to a queue), but this part is handled typically by **I_2O-core** applications. This function returns NULL_address if the service was not found. To compare xdsh_srvc_addr variables, the user has to use a special function (as the bytewise comparison trivial operator== is only defined by the c++ compiler (alas)). the special function for xdsh_srvc_addr comparison is :

```
int xdsh_service_cmp (xdsh_srvc_addr a, xdsh_srvc_addr b);
```

This one returns non-zero if the two addresses are equal.

```
typedef int (*xdsh_callback) (struct xdsh_msg *);
int xdsh_register_service (xdsh_callbak service_callback,
    const char * service_name);
void xdsh_unregister_service (const char * service_name);
```

Those functions return non-zero if something fails (for example if the service is already registered).

TODO : define return values for xdsh_callbacks and xdsh_register_service.

3.2.5 Message passing functions

Once the address is resolved for the service, the user function can access the driver, by calling message passing functions.

```
int xdsh_send_message (struct xdsh_file *fd,
    xdsh_srvc_addr service_address,
    struct xdsh_msg * message);
```

This function call will return the result of:

```
service_callback(message);
```

To get replies to the commands sent to services through a given file descriptor, the user program just has to use the reply function on the same file descriptor. The get reply function exists in two flavors: blocking and non blocking. There is no timeout method available for the moment.

```
struct xdsh_msg * xdsh_get_reply (struct xdsh_file *fd);
struct xdsh_msg * xdsh_get_reply_non_block (struct xdsh_file * fd);
```

Both function return NULL on failure. When the non-blocking function returns NULL, there is non reply message waiting in the queue. The blocking function can return NULL in some cases (like if the file descriptor is closed by another thread).

On the other side (kernel side) the functions are as follows:

```
struct xdsh_sender xdsh_get_sender (struct xdsh_msg * message);
int xdsh_send_reply (xdsh_sender * sender, struct xdsh_msg *reply);
```

As usual, `xdsh_send_reply` returns non-zero on failure.

TODO : define error codes.

It's up to the function that got the message (through "get_reply" or a callback) to dispose of the message frame. This can be done through a free, or the message frame can be reused for the reply.

3.2.6 Buffer allocation functions

In order to be able to transfer efficiently important amounts of data between hardware cards and programs, we need to use DMA-able buffers. Thoses buffers are typically allocated at the beginning of the application run and freed when the application finished. As these DMAable buffers have to be acessed through different adresses depending on the space we work on, it is handled through a structure that is common to user and kernel space (i.e. it can be passed as a member of a user message payload.

The buffer has to be accessed through various functions. The functions usable in user side are:

```
struct xdsh_Dbuff * xdsh_Dbuff_allocate (xdsh_file * fd, u32 size);
void * xdsh_Dbuff_user_address (struct xdsh_Dbuff * dma_buffer);
void xdsh_Dbuff_free (xdsh_file * fd, struct xdsh_Dbuff *);
```

Once again **`xdsh_Dbuff_allocate`** returns `NULL` on failure.

The functions usable on the kernel side are:

```
void * xdsh_Dbuff_kernel_address (struct xdsh_Dbuff * dma_buffer);
u32 xdsh_Dbuff_physical_address (struct xdsh_Dbuff * dma_buffer);
```

3.2.7 PCI device mapping

For the high perfoemance needs, there is the possibility to map the PCI bus in user space. In thie configuration, the user loses the layering of the system. There is not anymore a kernel space driver, and just communications between user and kernel space through Xdaq-shell. Instead, the kernel space driver needs to allocate the board, get its base address, possibly perform some initialisation, register the interrupt and then leave it up to user space programs to control the board in normal operation.

The API to the PCI maps is the following :

```
struct xdsh_Pmap * xdsh_Pmap_create (u32 base_address, u32 range);
u32 * xdsh_Pmap_user_address (struct xdsh_Pmap * pmap);
void xdsh_Pmap_destroy (struct xdsh_Pmap * pmap);
```

As usual, **`xdsh_Pmap_create`** returns NULL in case of failure.

Once the PCI is mapped in user space, reading a pointer in the mapped address is as fast as reading from memory, an involves no call to the system.

3.2.8 Miscelaneous utility functions

- Debug statements

  Xdaq-shell also provides utilty functions for printing messages (on the standard output or standard error for the user program, on the syslog for the kernel part). Those functions are iprintf, eprintf, idprintf and edprintf (i standing for infomation, e standing for error, and d for debug). The "*d*" functions only work if the preprocessor macro DEBUG is defined at compile time (trough the -DDEBUG command line option by example).

  ```
  int {e,i}[d]printf (const char * format, ...);
  ```

- Pre-installed module for basic PCI access

  The Xdaq-shell kernel side comes by default with a TrivialPCI module, which maps all the accesses on the PCI bus to correponding accesses in user space. This is usefull for debuging of hardware, of first quick and dirty tests of sofware. This is not performance oriented, but can be usefull for debuging. A corresponding userspace library will be provided but is still TBD. TODO.

3.2.9 Header files

The Xdaq-shell just requires inclusion of the xdaq-shell.h header file.

3.3 Example programs

Here we present a simple program that makes a board do a DMA into a buffer. The user application then receives a message announcing the DMA completion.

First let's see the structures used in this program: we need a user payload for messages, and we have to define some commands.

```
enum message_type {
    fill_buffer,
    buffer_filled,
    fill_failed
};

struct DMA_message {
    struct xdsh_Dbuff DMA_buffer;
    enum message_type message_type;
}
const char * dma_name = "DMA_fill";
const int dma_size = 1024;
```

Figure 3: Structures used in **Xdaq-shell** example program

3.3.1 Driver side (I_2O-core side)

On the driver side, we have to register the function that handles the message. The driver side part of the example is shown on Figure 4.

3.3.2 User side

```
void xdsh_init (void)
{
    xdsh_register_service (dma_callback, dma_name);
}

void xdsh_cleanup (void)
{
    xdsh_unregister_service (dma_name);
}

int dma_callback (struct xdsh_msg * msg)
{
    struct DMA_message *m =
         (struct DMA_message *)xdsh_msg_payload (msg);
    if (m->message_type == fill_buffer) {
         do_dma (/*to*/xdsh_Dbuff_physical_address(m->DMA_buffer));
         m->message_type = buffer_filled;
         xdsh_send_reply (xdsh_get_sender(msg), msg);
    } else {
         /* unknown command */
         return -1;
    }
}
```

Figure 4: Example program for **Xdaq-shell** on server (kernel) side

On the user side, we just want to get a DMAed buffer from the driver. This program is show on Figure 5.

```
test_DMA (void)
{
    /* This part is initialisation. It is not critical (in speed) */
    struct xdsh_Dbuff * buff = xdsh_Dbuff_allocate (dma_size);
    struct xdsh_file * f = xdsh_open();
    xdsh_srvc_addr dma_addr = xdsh_get_address (f, dma_name);
    struct xdsh_msg * msg = xdsh_alloc (sizeof (struct DMA_message));
    struct DMA_message * m =
              (struct DMA_message *)xdsh_msg_payload (msg);

    /* This is the time critical part */
    m->message_type = fill_buffer;
    m->DMA_buffer = *buff;
    xdsh_send_message (f, dma_addr, msg);
    msg = xdsh_receive_relpy (f);
    m = (struct DMA_message *)xdsh_msg_payload (msg);
    if (m->message_type == buffer_filled) {
         do_something_with_buffer
              (xdsh_Dbuff_user_address(&m->DMA_buffer));
         xdsh_buffer_free (fd, &m->DMA_buffer);
    }
    /* free up thinqs here */
}
```

Figure 5: Example program for **Xdaq-shell** on user space side

3.4 Performance impact

Performance impact of the usage of **Xdaq-shell** compared with native drivers (or absence of drivers) will be studied once the **Xdaq-shell** will be ready on each platform.

4 User's manual

4.1 Getting the software

The software can be retried from the CMS CVS server as an anonymous cvs user. The command sequence is :

```
cvs -d :pserver:anonymous@cmscvs.cern.ch:/cvs_server/repositories/TriDAS login
```

(password is, at time of writing `98passwd`. It can be found on the page : http://cmsdoc/cmsoo/projects/cvs_server.html)

```
cvs -d :pserver:anonymous@cmscvs.cern.ch:/cvs_server/repositories/TriDAS \
        co -P TriDAS/daq/itools
cvs -d :pserver:anonymous@cmscvs.cern.ch:/cvs_server/repositories/TriDAS \
        co -P -r V01_00_I2O TriDAS/Auxiliary/i2o
```

4.2 VxWorks version

Change directory to 'TriDAS/daq/itools/core/src/vxworks/mv2304'. Change the Makefile so that the variable VXINCLUDEDIR (by default equal to '/cms_cluster/tornado-2.0/ppc/target/h/') points to the appropriate directory for your tornado installation.

Then run gmake. This will create and all in one .exe file containing the test program, plus all i2ocore. This still has to be upgraded.

4.3 Linux version

How to compile link and run under Linux

TODO

5 Status and plans

5.1 I_2O-core

The support included in i2ocore is quite complete up to now. i2ocore supports now Linux kernels 2.2 and 2.4, with the same sources, and VxWorks. The implementation is quite stable, now.

5.2 Xdaq-shell

The xdaq-shell implementation was also ported to kernel 2.4 of linux (still compatible with 2.2). The send-get-reply calls and the timeout get-replys aren't fully tested. The PCImaps and the default general access module is not yet written.

References

1 I_2O Special interest group, *Intellignent I/O (I_2O) Architecture Specification* see I_2O SIG web site: http://www.i2osig.org

Appendix A : Functions summary

I_2O core functions

```
I2O_ADAPTER_ID i2oExecutiveAdapterAttach (U16 PciVendorID,
    U16 PciDeviceID, int index, I2O_STATUS* pStatus)
void i2oExecutiveAdapterDetach (I2O_ADAPTER_ID id)
i2oAdapterIntLock
i2oAdapterIntUnlock
i2oAdapterPhysLocGet

i2oConfig{Read|Write}{8|16|32|64}

I2O_BUS_ID i2oAdapterBusGet (I2O_ADAPTER_ID  adapterId, I2O_STATUS * pStatus)
i2oBusLocal
i2oBusSystem
i2oBus{Read|Write}{8|16|32|64}
i2oBusTranslate

I2O_THREAD_ID i2oThreadCreate (I2O_OWNER_ID ownerId, I2O_THREAD_PRI threadPri,
    I2O_THREAD_OPTIONS threadOptions, I2O_SIZE threadStackSize,
    I2O_THREAD_FUNC *threadInitFunc, I2O_ARG threadArg, I2O_STATUS * pStatus)
void i2oThreadDelay (I2O_USECS usecs, I2O_STATUS * pStatus)
I2O_THREAD_ID i2oThreadIdSelf (I2O_STATUS * pStatus)
void i2oThreadLock (I2O_STATUS * pStatus)
void i2oThreadUnlock (I2O_STATUS * pStatus)
I2O_THREAD_PRI i2oThreadPriGet (I2O_THREAD_ID threadId, I2O_STATUS * pStatus)
void i2oThreadPriSet (I2O_THREAD_ID threadId, I2O_THREAD_PRI threadPri,
    I2O_STATUS * pStatus)

i2oEventQCreate
i2oEventQThreadGet
i2oEventQPost

I2O_SEM_ID i2oSemBCreate (I2O_OWNER_ID  ownerId, I2O_SEM_OPTIONS semOptions,
    I2O_SEM_B_STATE initialState, I2O_STATUS * pStatus)
I2O_SEM_ID i2oSemCCreate (I2O_OWNER_ID  ownerId, I2O_SEM_OPTIONS semOptions,
    I2O_COUNT initialCount, I2O_STATUS * pStatus)
I2O_SEM_ID i2oSemMCreate (I2O_OWNER_ID  ownerId, I2O_SEM_OPTIONS semOptions,
    I2O_STATUS * pStatus)
void i2oSemTake (I2O_SEM_ID semId, I2O_USECS timeout, I2O_STATUS * pStatus)
void i2oSemGive (I2O_SEM_ID semId, I2O_STATUS * pStatus)

I2O_INT_ID i2oIntCreate (I2O_OWNER_ID ownerId, I2O_OBJ_CONTEXT intContext,
    I2O_ADAPTER_ID adapterId, I2O_ISR_HANDLER * isrHandler, I2O_ARG isrArg,
    I2O_EVENT_QUEUE_ID evtQId, I2O_COUNT maxEvts, I2O_STATUS * pStatus)
void i2oIntEventPost (I2O_INT_ID intId, I2O_EVENT_PRI evtPri,
    I2O_EVENT_HANDLER intEvtHandler, I2O_ARG intEvtArg, I2O_STATUS * pStatus)
BOOL i2oIntInIsr (void)
I2O_INT_LOCK_KEY i2oIntLock (void)
void i2oIntUnlock (I2O_INT_LOCK_KEY key)
```

with (user defined function):

```
void threadInitFunction (I2O_ARG threadArg)
BOOL intHandled = intHandler (I2O_OBJ_CONTEXT intContext, I2O_ARG isrArg)
void intEvtHandler (I2O_OBJ_CONTEXT intContext)
```

Xdaq-shell function

Functions common to both sides :

Functions for kernel side :

Functions for user side :

```
struct xdsh_msg * xdsh_alloc (u32 size);
void xdsh_free (struct xdsh_msg *);
void * xdsh_msg_payload (struct xdsh_msg *);
u32 xdsh_msg_size (const struct xdsh_msg *);
struct xdsh_file * fd = xdsh_open ();
void xdsh_close (struct xdsh_file * fd);
xdsh_srvc_addr xdsh_get_address (struct xdsh_file * file,
    const char *service_name);
int xdsh_service_cmp (xdsh_srvc_addr a, xdsh_srvc_addr b);
typedef int (*xdsh_callback) (struct xdsh_msg *);
int xdsh_register_service (xdsh_callbak service_callback,
    const char * service_name);
void xdsh_unregister_service (const char * service_name);
int xdsh_send_message (struct xdsh_file *fd,
    xdsh_srvc_addr service_address,
    struct xdsh_msg * message);
struct xdsh_msg * xdsh_get_reply (struct xdsh_file *fd);
struct xdsh_msg * xdsh_get_reply_non_block (struct xdsh_file * fd);
struct xdsh_sender xdsh_get_sender (struct xdsh_msg * message);
int xdsh_send_reply (xdsh_sender * sender, struct xdsh_msg *reply);
struct xdsh_Dbuff * xdsh_Dbuff_allocate (xdsh_file * fd, u32 size);
void * xdsh_Dbuff_user_address (struct xdsh_Dbuff * dma_buffer);
void xdsh_Dbuff_free (xdsh_file * fd, struct xdsh_Dbuff *);
void * xdsh_Dbuff_kernel_address (struct xdsh_Dbuff * dma_buffer);
u32 xdsh_Dbuff_physical_address (struct xdsh_Dbuff * dma_buffer);
int {e,i}[d]printf (const char * format, ...);
```

Appendix B: Basic design with i2o-core and Xdaq-shell

TODO

Appendix C: Performance critical designs

TODO