Design Note for Developing Impulse C Applications for POSIX-Compliant Operating System Support

Using the Impulse C and Xilinx Software Development Kits to integrate existing Impulse C applications and custom hardware accelerators into the context of a real-time operating system

Contents

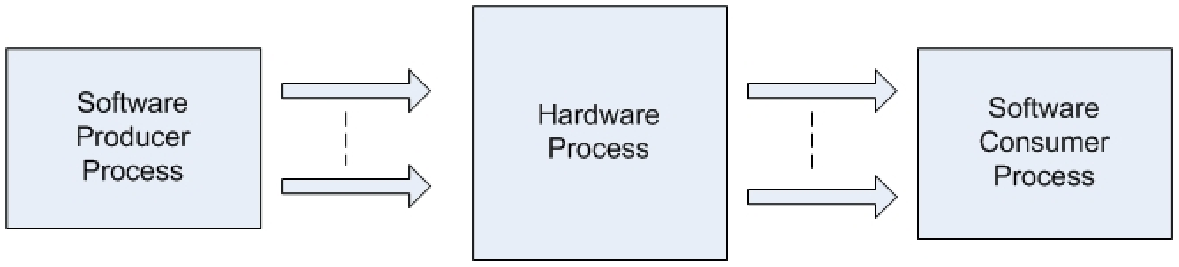
Introduction

This design note discusses the recommended Impulse C application design and implementation techniques that should be followed to help ease the POSIX integration effort. It begins with a discussion of the high-level design of the application and then moves toward more code-level items that should be understood by the developer.

Application Design

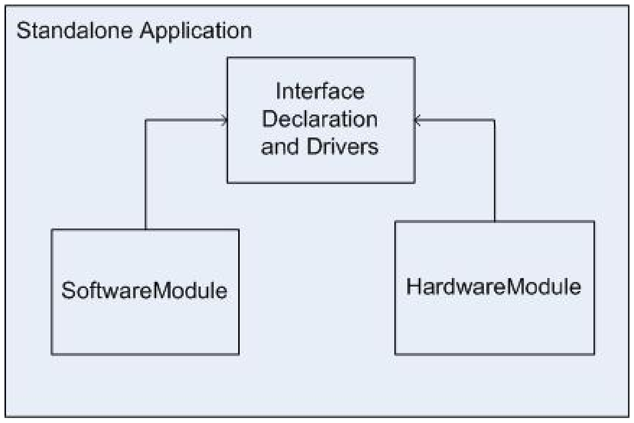
Impulse C applications are generally reducible to producer/consumer problems in which there is at least one software producer process that feeds at least one hardware process, which in turn feeds at least one software consumer process. Any reason to deviate from this design philosophy is an indication that Impulse C might not be suitable for that specific application.

Our integration process has been tested with designs that conform to the basic single producer/ consumer feedback loop setup, as shown below.



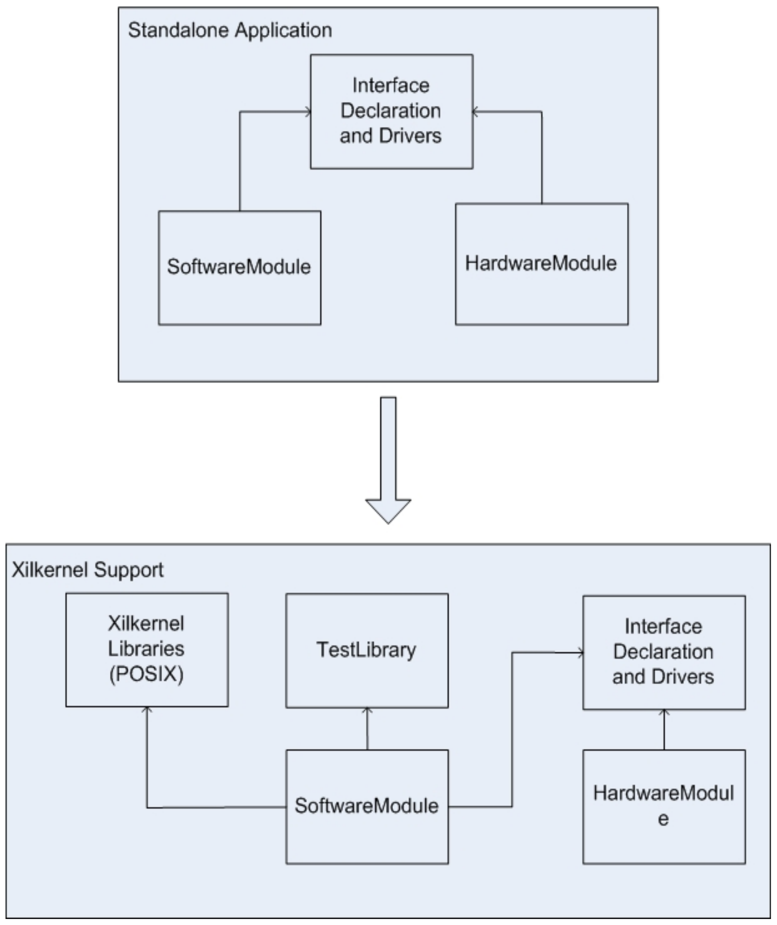
In this typical design there is a single producer that uses at least one stream to send data to a single hardware process, which in turn uses this input data to compute a result and send it to a single consumer using at least one stream. Impulse C provides additional communication mechanisms, such as signals (very similar to semaphores), which can be integrated into the POSIX environment as well if the proper Impulse C libraries are included. For the purposes of these tutorials, however, we will focus on designs that use simple streaming mechanisms to communicate between processes.

This design typically yields the following module structure after compilation:



As one can see, there is a single module dedicated to each of the hardware and software components of the application. Ultimately, the definition for hardware and software processes can be located in any module, so long as the co_initialize function takes into account which processes are being targeted towards hardware and which are being targeted towards software. It is common practice to keep physically separate processes in separate module files, just for maintainability and readability reasons alone.

Once this application has been compiled it is ready to run as a standalone executable on the target platform, as shown in the above figure. However, since we are integrating this into a POSIX-compliant environment, we need to make manual adjustments to the source code to make it compatible. These changes (which are outlined in the **Impulse C Integration Tutorial**) will yield the following module design:



Translation Script Usage

The Python script **impulse_convert.py** is provided to assist you in the translation process from a standalone Impulse C application to one that runs within the Xilkernel. It performs minimal source code translation and generates the code that needs to be manually inserted within a larger Xilkernel project.

To use this script you must provide a single software module description file, hardware module description file, and a common header that is used to interface between these two partitions of the application. The description files are merely C source code files that consist of the Impulse C process declarations and implementations that drive the application.

A sample Impulse C application template is available to assist you in following this implementation technique. It consists of a single software, hardware, and interface header file that you can modify and extend the implement the logic of your application.

Once these files have been generated, you can run the translation script as follows:

```
python impulse_convert.py swfile.c hwfile.c interface.h
```

When run with a modified version of the simple BasicStream application (which implements a simple feedback loop for proof-of-concept purposes), the script generates output shown below. You should inspect this output carefully, but the elements in bold are most important to you in the translation process.

```
Input software module:  BasicStream_sw.c
Input hardware module:  BasicStream_hw.c
Input module interface: BasicStream.h

Determining processes.

extern void Producer(co_stream HWinput, co_stream test)
extern void Consumer(co_stream HWoutput)

Parsing the functions that were found.

Parsing: extern void Producer(co_stream HWinput, co_stream test)
Function name: Producer
tern void Producer(co_stream HWinput, co_stream test)
tern void Producer(co_stream HWinput, co_stream test)
Parameter structure wrapper:

typedef struct
{
   co_stream HWinput;
   co_stream test;
} ProducerParams;

Parsing: extern void Consumer(co_stream HWoutput)
Function name: Consumer
tern void Consumer(co_stream HWoutput)
Parameter structure wrapper:

typedef struct
{
   co_stream HWoutput;
} ConsumerParams;

Generating POSIX thread creation templates.
Note that the last parameter (NULL) will need to be manually replaced when integrated into the Xilinx SDK.

pthread_t id0;
ConsumerParams params; // CONFIGURE MANUALLY!
pthread_create(&id0, NULL, (void*)Consumer, &params);

pthread_t id1;
ProducerParams params; // CONFIGURE MANUALLY!
pthread_create(&id1, NULL, (void*)Producer, &params);

Software file update - you need to add the following code for each INIT_TIMER macro:
 #include "ProfileLibrary.h"
 XTmrCtr timer;

Remaining steps:
 1. Remove redundant main method in BasicStream_sw.c
 2. Include pthread creation function calls into the main routine
 3. Include the Impulse C libraries into the SDK build path
```

Note how the script was able to identify the software processes in this application and generate the correct POSIX thread creation code that must be inserted to start each of these threads. Behind the scenes, it also modifies the interface file to contain globally visible declarations for the software processes and their parameters so that the Xilkernel application can reference them when creating these POSIX threads.

It is up to you to follow the remaining steps in the Impulse C integration tutorial to figure out where this bold code must go in the Xilkernel project setup, but this script should rapidly speed up the process of integration.