Device Driver Programmer Guide

Overview

The purpose of this document is to describe the Xilinx device driver environment. This includes the device driver architecture, the Application Programmer Interface (API) conventions, the scheme for configuring the drivers to work with reconfigurable hardware devices, and the infrastructure that is common to all device drivers.

This document is intended for the software engineer that is using the Xilinx device drivers. It contains design and implementation details necessary for using the drivers.

Goals and Objectives

The Xilinx device drivers are designed to meet the following goals and objectives:

- Provide maximum portability

  The device drivers are provided as ANSI C source code. ANSI C was chosen to maximize portability across processors and development tools. Source code is provided both to aid customers in debugging their applications as well as allow customers to modify or optimize the device driver if necessary.

  A layered device driver architecture additionally separates device communication from processor and Real Time Operating System (RTOS) dependencies, thus providing portability of core device driver functionality across processors and operating systems.

- Support FPGA configurability

  Since FPGA-based devices can be parameterized to provide varying functionality, the device drivers must support this varying functionality. The configurability of device drivers should be supported at compile-time and at run-time. Run-time configurability provides the flexibility needed for future dynamic system reconfiguration.

  In addition, a device driver supports multiple instances of the device without code duplication for each instance, while at the same time managing unique characteristics on a per instance basis.

- Support simple and complex use cases

  Device drivers are needed for simple tasks such as board bring-up and testing, as well as complex embedded system applications. A layered device driver architecture provides both simple device driver interfaces with minimal memory footprints and more robust, full-featured device driver interfaces with larger memory footprints.

- Ease of use and maintenance

  Xilinx makes use of coding standards and provides well-documented source code in order to give developers (i.e., customers and internal development) a consistent view of source code that is easy to understand and maintain. In addition, the API for all device drivers is consistent to provide customers a similar look and feel between drivers.

Device Driver Architecture

The architecture of the device drivers is designed as a layered architecture as shown in Figure . The layered architecture accommodates the many use cases of device drivers while at the same time providing portability across operating systems, toolsets, and processors. The layered architecture provides seamless integration with an RTOS (Layer 2), an abstract device driver interface that is full-featured and portable across operating systems and processors (Layer 1), and a direct hardware interface for simple use cases or those wishing to develop a custom device driver. The following paragraphs describe the layers.

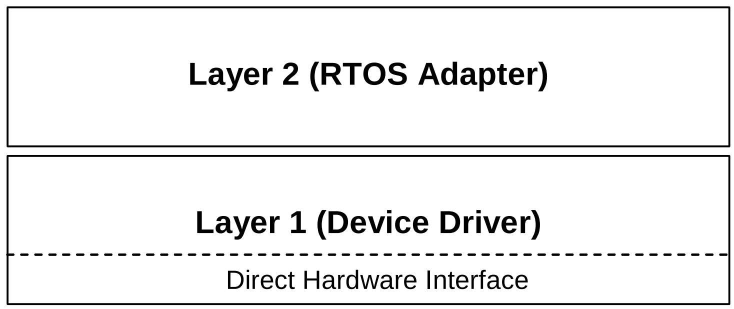

Figure 1: **Layered Architecture**

It’s important to note that the direct hardware interface does not add additional overhead to the device driver function call overhead, as it is typically implemented as a set of manifest constants and macros.

Layer 2 (RTOS Adapter)

This layer is an adapter between an RTOS and a device driver. It converts a Layer 1 device driver to an interface that matches the requirements of the driver model for an RTOS. Unique adapters may be necessary for each RTOS. Adapters typically have the following characteristics.

- Communicates directly to the RTOS as well as the Layer 1 interface of the device driver.
- References functions and identifiers specific to the RTOS. This layer is therefore not portable across operating systems.
- Can use memory management
- Can use RTOS services such as threading, inter-task communication, etc.
- Can be simple or complex depending on the RTOS interface and requirements for the device driver

Layer 1 (Device Driver)

This layer is an abstract device driver interface that shields the user from potential changes to the underlying hardware. It is implemented with macros and functions and designed to allow a developer to utilize all features of a device. The device driver is independent of operating systems and processors, making it highly portable. This interface typically has the following characteristics.

- Consistent API that gives the user an "out-of-the-box" solution. The abstract API helps isolate the user from hardware changes.
- No RTOS or processor dependencies, making the device driver highly portable
- Run-time error checking such as assertion of input arguments, and provides the ability to compile away asserts.
- Comprehensive support of device features
- Supports device configuration parameters to handle FPGA-based parameterization of hardware devices.
- Supports multiple instances of a device while managing unique characteristics on a per instance basis.
- Polled and interrupt driven I/O
- Non-blocking function calls to aid complex applications
- May have a large memory footprint
- Typically provides buffer interfaces for data transfers as opposed to byte interfaces. This makes the API easier to use for complex applications.
- Does not communicate directly to Layer 2 adapters or application software. Utilizes asynchronous callbacks for upward communication.

Direct Hardware Interface

This interface, which is contained within the Layer 1 device driver, is a direct hardware interface. It is typically implemented as macros and manifest constants and designed to allow a developer to create a small applications or create a custom device driver. This interface typically has the following characteristics.

- Constants that define the device register offsets and bit fields, and simple macros that give the user access to the hardware registers
- Small memory footprint
- Little to no error checking is performed
- Minimal abstraction such that the API typically matches the device registers. The API is therefore less isolated from hardware device changes.
- No support of device configuration parameters
- Supports multiple instances of a device with base address input to the API
- No, or minimal, state is maintained
- Polled I/O only
- Blocking functions for simple use cases
- Typically provides byte interfaces

API and Naming Conventions

External Identifiers

External identifiers are defined as those items that are accessible to all other software in the system (global) and include functions, constants, typedefs, and variables.

An ’X’ is prepended to each Xilinx external so it does not pollute the global name space, thus reducing the risk of a name conflict with application code. The names of externals are based upon the driver in which they exist. The driver name is prepended to each external name. An underscore character always separates the driver name from the variable or function name.

External Name Pattern:

```
X<driver_name>_VariableName;
X<driver_name>_FunctionName(ArgumentType Argument)
X<driver_name>_TypeName;
```

Constants are typically defined as all uppercase and prefixed with an abbreviation of the driver name. For example, a driver named XUartLite (for the UART Lite device driver) would have constants that begin with XUL_, and a driver named XEmac (for the Ethernet 10/100 device driver) would have constants that begin with XEM_. The abbreviation utilizes the first three uppercase letters of the driver name, or the first three letters if there are only two uppercase letters in the name.

File Naming Conventions

The file naming convention utilizes long file names and is not limited to 8 characters as imposed by the older versions of the DOS operating system. Drivers may be grouped into several source files. This gives users the flexibility of linking drivers into a library and ultimately minimizing the linked object size of a driver to those functions which are actually used by an application.

Driver Source File Names

Source file names are based upon the name of the driver such that the contents of the source file are obvious from the file name. All file names must begin with the lowercase letter "x" to differentiate Xilinx source files. File extensions .h and .c are utilized to distinguish between header source files and implementation source files.

Implementation Source Files (*.c)

The C source files contain the implementation of a driver. A driver implmentation is typically contained in multiple source files to allow parts of the driver to be user selectable (i.e., linked with an application).

Source File Naming Pattern:

```
x<driver_name>.c                           main source file
x<driver_name>_functionality.c             secondary source file
```

Header Source Files (*.h)

The header files contain the interfaces for a driver. There will always be public interfaces which is what an application uses.

- The high-level, abstract interface (Layer 1) for a driver is contained in a header file with the file name format *x<driver_name>.h*.
- The direct hardware interface is contained in a header file with the file name format *x<driver_name>_hw.h* (or *x<driver_name>_l.h*).

In the case of multiple C source files which implement the driver, there may also be a header file which contains internal interfaces for the driver. The internal interfaces allow the functions within each source file to access functions in the another source file.

- The internal interfaces are contained in a header file with the file name format *x<driver_name>_i.h*.

Device Driver Layers

Layer 1 and the direct hardware interface (i.e., high-level and low-level driver interfaces) are bundled together in a directory. The direct hardware interface files are named *x<driver_name>_hw.h* (or *_l.h*) and sometimes *x<driver_name>_hw.c* (or *_l.c*). The "*_hw*" indicates hardware-specific definitions. Layer 2 RTOS adapter files typically include the word "adapter" in the file name, such as *x<driver_name>_adapter.h* and *x<driver_name>_adapter.c*. These are stored in a different directory name (e.g., one specific to the RTOS) than the device driver files.

Example File Names

The following source file names illustrates an example which is complex enough to utilize multiple C source files.

```
xuartns550.c          Main implementation file
xuartns550_intr.c     Secondary implementation file for interrupt
handling
xuartns550.h          High level external interfaces header file
xuartns550_i.h        Internal identifiers header file
xuartns550_l.h        Low level external interfaces header file
xuartns550_l.c        Low level implementation file
xuartns550_g.c        Generated file controlling parameterized
instances

and,

xuartns550_sio_adapter.c  VxWorks Serial I/O (SIO) adapter
```

Device Driver API

Device drivers are designed to have an API which includes a standard API together with functions that may be unique to that device. The standard API provides a consistent interface for Xilinx drivers such that the effort to use multiple device drivers is minimized. An example API follows.

Instance Pointers

All abstract functions and macros take an Instance pointer as the first argument. The Instance pointer is a pointer to the driver data structure. Each driver defines the instance structure that containts instance-specific information. This allows the driver to support multiple instances of a device. The following code example illustrates a device driver instance.

```
/* the device driver data type */

typedef struct {
    u32 BaseAddress;  /* driver data variables */
    u32 IsReady;
    u32 IsStarted;
} XDevice;

/* create an instance of a device */
```

```
XDevice DeviceInstance;

/* device driver interfaces */

int XDevice_Initialize(XDevice *InstancePtr, u16 DeviceId);
int XDevice_Start(XDevice *InstancePtr);
```

Standard Device Driver API

CfgInitialize

This function initializes an instance of a device driver. Initialization must be performed before the instance is used. Initialization includes mapping a device to a memory-mapped address and initialization of data structures. It maps the instance of the device driver to a physical hardware device. The user is responsible for allocating an instance variable using the driver's data type, and passing a pointer to this variable to this and all other API functions. If using static initialization using the driver's _g.c configuration table, users would typically call the driver's LookupConfig() function and pass the resulting configuration entry to this CfgInitialize() function. For example:

```
XLlTemac TemacInstance;
XLlTemac_Config *MacConfigPtr;
int Status;

MacConfigPtr = XLlTemac_LookupConfig(XPAR_LLTEMAC_0_DEVICE_ID);
if (MacConfigPtr != NULL) {
  Status = XLlTemac_CfgInitialize(&TemacInstance, MacConfigPtr,
MacConfigPtr->BaseAddress);
  ...
}
```

Reset

This function resets the device driver and device with which it is associated. This function is provided to allow recovery from exception conditions. This function resets the device and device driver to a state equivalent to after the Initialize() function has been called.

SelfTest

This function performs a self-test on the device driver and device with which it is associated. The self-test verifies that the device and device driver are functional.

LookupConfig

This function retrieves a pointer to the configuration table for a device driver. The configuration table data type is typically defined in the driver's main header file, and the table itself is defined in the _g.c file of the driver. This function gives the user a mechanism to view or modify the table at run-time, which allows for run-time configuration of the device driver. Note that modification of the configuration data for a driver at run-time must be done by the user prior to invoking the driver's CfgInitialize() function.

Optional Functions

Each of the following functions may be provided by device drivers.

Start

This function is provided to start the device driver. Starting a device driver typically enables the device and enables interrupts. This function, when provided, must be called prior to other data or event processing functions.

Stop

This function is provided to stop the device driver. Stopping a device driver typically disables the device and disables interrupts.

GetStats

This function gets the statistics for the device and/or device driver.

ClearStats

This function clears the statistics for the device and/or device driver.

InterruptHandler

This function is provided for interrupt processing when the device must handle interrupts. It does not save or restore context. The user is expected to connect this interrupt handler to their system interrupt controller. Most drivers will also provide hooks, or callbacks, for the user to be notified of asynchronous events during interrupt processing (e.g., received data or device errors).

Configuration Parameters

Standard device driver API functions such as CfgInitialize() and Start() require basic information about the device such as where it exists in the system memory map or how many instances of the device there are. In addition, the hardware features of the device may change because of the ability to reconfigure the hardware within the FPGA. Other parts of the system such as the operating system or application may need to know which interrupt vector the device is attached to. For each device driver, this type of information is distributed across two files: *xparameters.h* and *x<driver_name>_g.c*.

Typically, these files are automatically generated by a system generation tool based on what the user has included in their system. However, these files can be hand coded to support internal development and integration activities.

The existence of these configuration files implies static, or compile-time, configuration of device drivers. It should be noted that a user is free to implement dynamic, or run-time, configuration of device drivers by passing its own driver configuration structure to CfgInitialize().

xparameters.h

This source file centralizes basic configuration constants for all devices and drivers within the system. Browsing this file gives the user an overall view of the system architecture. The device drivers and Board Support Package (BSP) utilize the information contained here to configure the system at runtime. The amount of configuration information varies by device, but at a minimum the following items should be defined for each device:

- Number of device instances
- Device ID for each instance
- A Device ID uniquely identifies each hardware device which maps to a device driver. A Device ID is used during initialization to perform the mapping of a device driver to a hardware

device. Device IDs are typically assigned either by the user or by a system generation tool. It is currently defined as a 16-bit unsigned integer.

- Base address for each device instance
- Interrupt assignments for each device instance - if interrupts can be generated.

File Format and Naming Conventions

Every device must have the following constant defined indicating how many instances of that device are present in the system (note that <driver_name> does not include the preceding "X"):

```
XPAR_X<driver_name>_NUM_INSTANCES
```

Each device instance will then have multiple, unique constants defined. There may be up to two sets of constants per driver. One set whose names typically match the hardware instance names, and one set of canonical names that are based on the device driver name and therefore consistently named across designs. There are several constants per driver. For example, each device instance has a unique device identifier (DEVICE_ID), the base address of the device's registers (BASEADDR), and the end address of the device's registers (HIGHADDR).

```
XPAR_<driver_name>_<device_instance>_DEVICE_ID
XPAR_<driver_name>_<device_instance>_BASEADDR
XPAR_<driver_name>_<device_instance>_HIGHADDR
```

<driver_name> is the uppercase name of the device driver (e.g., LLTEMAC, UARTLITE), and <device_instance> is the ordinal number of the hardware instance being identified. For example, 0 for the first Uart, 1 for the second Uart, etc... Again, there may be another set of constants that use <hardware_instance_name> instead of <driver_name>. The advantage of the canonical <driver_name> constants is that these do not change from hardware design to hardware design, thus requiring no changes in software applications that make use of these constants when re-using an application for a different hardware design.

For devices that can generate interrupts, a separate section within *xparameters.h* is used to store interrupt vector information. While the device driver implementation files do not utilize this information, their RTOS adapters, BSP files, or user application code will require them to be defined in order to connect, enable, and disable interrupts from that device. The naming convention of these constants varies whether an interrupt controller is part of the system or the device hooks directly into the processor.

For the case where an interrupt controller is considered external and part of the system, the naming convention is as follows:

```
XPAR_<intc_device_instance>_<interrupting_device_instance>_VEC_ID
```

Where <intc_device_instance> is the name of the interrupt controller device, and <interrupting_device_instance> is the name of the device connected to the controller. This convention supports single and cascaded interrupt controller architectures.

x<driver_name>_g.c

The header file *x<driver_name>.h* defines a configuration data structure for a driver. The structure will contain all of the configuration information necessary for an instance of the device. The format of the data type is as follows:

```
typedef struct {
   u16 DeviceID;
   u32 BaseAddress;
```

```
   /* Other device dependent data attributes */

} X<driver_name>_Config;
```

The implementation file *x<driver_name>_g.c* defines a table, or an array of structures of `X<driver_name>_Config` type. Each element of the array represents an instance of the device, and contains most of the per-instance XPAR constants from *xparameters.h*. This table is typically used by the LookupConfig() function prior to initializing the driver.

Example

To help illustrate the relationships between these configuration files, an example is presented that contains a single interrupt controller whose name is INTC and a single UART whose name is (UART). Only xintc.h and xintc_g.c are illustrated, but xuart.h and xuart_g.c would be very similar.

xparameters.h

```
/* Constants for INTC */
XPAR_XINTC_NUM_INSTANCES     1
XPAR_INTC_0_DEVICE_ID       21
XPAR_INTC_0_BASEADDR        0xA0000100

/* Interrupt vector assignments for this instance */
XPAR_INTC_0_UART_0_VEC_ID   0

/* Constants for UART */
XPAR_XUART_NUM_INSTANCES     1
XPAR_UART_0_DEVICE_ID       2
XPAR_UART_0_BASEADDR        0xB0001000
```

xintc.h

```
typedef struct {
   u16 DeviceID;
   u32 BaseAddress;
} XIntc_Config;
```

xintc_g.c

```
static XintcConfig[XPAR_XINTC_NUM_INSTANCES] =
{
  {
     XPAR_INTC_0_DEVICE_ID,
     XPAR_INTC_0_BASEADDR,
  }
}
```

Common Driver Infrastructure

Source Code Documentation

The comments in the device driver source code contain *doxygen* tags for *javadoc*-style documentation. *Doxygen* is a *javadoc*-like tool that works on C language source code. These tags typically start with "@" and provide a means to automatically generate HTML-based documentation for the device drivers. The HTML documentation contains a detailed description of the API for each device driver.

Driver Versions

Some device drivers may have multiple versions. Device drivers are usually versioned when the API changes, either due to a significant hardware change or simply restructuring of the device driver code. The version of a device driver is only indicated within the comment block of a device driver file. A modification history exists at the top of each file and contains the version of the driver. An example of a device driver version is "v1.00b", where 1 is the major revision, 00 is the minor revision, and b is a subminor revision.

Currently, the user is not allowed to link two versions of the same device driver into their application. The versions of a device driver use the same function and file names, thereby preventing them from being linked into the same link image. As multiple versions of drivers are supported within the same executable, the version name will be included in the driver function and file names, as in *x<driver_name>_v1_00_a.c*, in order to avoid namespace conflicts..

Primitive Data Types

The primitive data types provided by C are minimized by the device drivers because they are not guaranteed to be the same size across processor architectures. Data types which are size specific are utilized to provide portability and are contained in the header file *xbasic_types.h*.

Device I/O

The method by which I/O devices are accessed varies between processor architectures. In order for the device drivers to be portable, this difference is isolated such that the driver for a device will work for many microprocessor architectures with minimal changes. A device I/O component, XIo, in *xio.c* and *xio.h* source files, contains functions and/or macros which provide access to the device I/O and are utilized for portability.

Error Handling

Errors that occur within device drivers are propagated to the application. Errors can be divided into two classes, synchronous and asynchronous. Synchronous errors are those that are returned from function calls (either as return status or as a parameter), so propagation of the error occurs when the function returns. Asynchronous errors are those that occur during an asynchronous event, such as an interrupt and are handled through callback functions.

Return Status

In order to indicate an error condition, functions which include error processing return a status which indicates success or an error condition. Any other return values for such functions are returned as parameters. Error codes definitions are contained in the file *xstatus.h*.

Asserts

Asserts are utilized in the device drivers to allow better debugging capabilities. Asserts are used to test each input argument into a function. Asserts are also used to ensure that the component instance has been initialized.

Asserts may be turned off by defining the symbol NDEBUG before the inclusion of the header file *xbasic_types.h*.

The assert macro is defined in *xbasic_types.h* and calls the function XAssert when an assert condition fails. This function is designed to allow a debugger to set breakpoints to check for assert conditions when the assert macro is not connected to any form of I/O.

The XAssert function calls a user defined function and then enters an endless loop. A user may change the default behavior of asserts such that an assert condition which fails does return to the user by changing the initial value of the variable XWaitInAssert to XFALSE in *xbasic_types.c*. A user defined function may be defined by initializing the variable XAssertCallbackRoutine to the function in *xbasic_types.c*.

Communication with the Application

Communication from an application to a device driver is implemented utilizing standard function calls. Asynchronous communication from a device driver to an application is accomplished with callbacks using C function pointers. It should be noted that callback functions are called from an interrupt context in many drivers. The application function called by the asynchronous callback must minimize processing to communicate to the application thread of control.

Reentrancy and Thread Safety

The device drivers are designed to be reentrant, but may not be thread-safe due to shared resources. Thread safety must be handled by the software layer above the driver.

Interrupt Management

The device drivers, when possible, use device-specific interrupt management rather than processor-specific interrupt management.

When using a driver that supports interrupts, the application must set callback handlers for asynchronous notification of interrupt events - even if the user is not interested in any of the interrupt eventes (which is not likely). The driver provides one or more X<driver_name>_Setxx functions to allow the application to set callback handlers. If the application does not set a callback handler, the driver defaults to calling a stub handler. The stub handler will assert so that the user will notice the outage during debugging. The user should correct the application to properly set callback handlers for the driver.

Multi-threading & Dynamic Memory Management

The device drivers are designed without the use of multi-threading and dynamic memory management. This is expected to be accomplished by the application or by an RTOS adapter.

Cache & MMU Management

The device drivers are typically designed without the use of cache and MMU management. This is expected to be accomplished by the application or by an RTOS adapter. However, there are certain cases (e.g., DMA) where the driver may make use of macros defined in xenv.h in order to accomplish OS-specific cache operations, such as flushin and invalidating DMA buffer descriptors. Please read each driver's comments for details.

Revision History

The following table shows the revision history for this document.

Table 1: **Date**	*Table 2:* **Version**	*Table 3:* **Revision**
06/28/02	1.0	Xilinx initial release.
07/02/02	1.1	Made IP Spec # conditional text and removed ML reference.
07/31/02	1.2	Update to non-IP chapter template
12/17/04	1.3	Minor text updates/additions.
06/22/07	1.4	Code style updates, general updates
12/14/07	1.5	CfgInitialize/LookupConfig connection, general clean-up