XILINX®

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 drivers with minimal memory footprints and more robust, full-featured device drivers 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), high-level device drivers that are full-featured and portable across operating systems and processors (Layer 1), and low-level drivers for simple use cases (Layer 0). The following paragraphs describe each of the layers. The user can choose to use any and all layers.

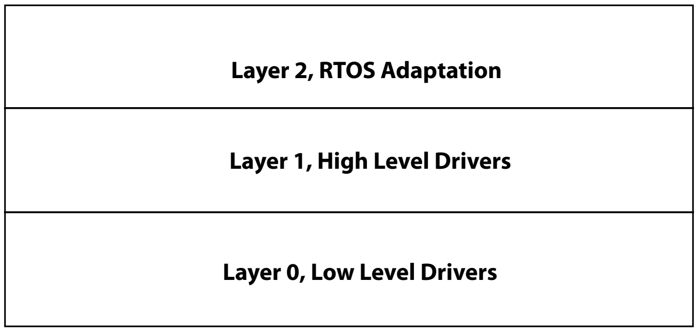

Figure 1: **Layered Architecture**

Layer 2, RTOS Adaptation

This layer consists of adapters for device drivers. An adapter converts a Layer 1 device driver interface to an interface that matches the requirements of the device driver scheme for an RTOS. Unique adapters may be necessary for each RTOS. Adapters typically have the following characteristics.

- Communicates directly to the RTOS and the Layer 1, high-level 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, High Level Drivers

This layer consists of high level device drivers . They are implemented as macros and functions and are designed to allow a developer to utilize all features of a device. These high-level drivers are independent of operating system and processor, making them highly portable. They typically have the following characteristics.

- Consistent and high-level (abstract) API that gives the user an "out-of-the-box" solution
- No RTOS or processor dependencies, making them highly portable
- Run-time error checking such as assertion of input arguments. Also provides the ability to compile away asserts.
- Comprehensive support of device features
- Abstract API that isolates the API from hardware device changes
- 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.

Layer 0, Low Level Drivers

This layer consists of low level device drivers. They are implemented as macros and functions and are designed to allow a developer to create a small system, typically for internal memory of an FPGA. They typically have the following characteristics.

- Simple, low-level API
- Small memory footprint
- Little to no error checking is performed
- Supports primary device features only
- 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
- None or minimal state is maintained
- Polled I/O only
- Blocking functions for simple use cases
- Typically provides byte interfaces but can provide buffer interfaces for packet-based devices.

Object-Oriented Device Drivers

In addition to the layered architecture, it is important that the user understand the underlying design of the device drivers. The device drivers are designed using an object-oriented methodology. The methodology is based upon components and is described in the following paragraphs. This approach pertains particularly to the Layer 1, high-level device drivers.

Component Definition

A component is a logical partition of the software which provides a functionality similar to one or more classes in C++. Each component provides a set of functions that operate on the internal data of the component. In general, components are not allowed access to the data of other components. A device driver is typically designed as a single component. A component may consist of one or more files.

Component Implementation

The component contains data variables which define the set of values that instances of that type can hold and a set of functions that operate on those data variables. Components must utilize the functions of other components in order to access the data of other components, rather than accessing component data directly. Components provide data abstraction and encapsulation by gathering the state of an object and the functions that operate on that object into a single unit and by denying direct access to its data members.

Component Data Variables

The primary mechanism for implementing a component in C is the structure. The data variables for a component are grouped in a single structure such that instances of the component each have their own data. The structure and the prototypes for all component functions are declared in the header file which is shared between the implementing component and other components which utilize it. A pointer to this structure, referred to as the instance pointer, is passed into each function of the component which operates on the instance data.

Component Interface

Each component has a set of functions which are collectively referred to as the component interface. Every function of a component which operates on the instance data utilizes a pointer, named InstancePtr, to an instance of a component as the first argument. This argument emulates the *this* pointer in C++ and allows the component function to manipulate the instance data.

Component Instance

An instance of a component is created when a variable is created using the component data type. An instance of a component maps to each physical hardware device. Each instance may have unique characteristics such as it's memory mapped address and specific device capabilities.

Component Example

The following code example illustrates a device driver component.

```
/* the device component data type */

typedef struct
{
    Xuint32 BaseAddress;  /* component data variables */
    Xuint32 IsReady;
    Xuint32 IsStarted;
} XDevice;

/* create an instance of a device */

XDevice DeviceInstance;

/* device component interfaces */
```

```
XStatus XDevice_Initialize(XDevice *InstancePtr, Xuint16 DeviceId);
XStatus XDevice_Start(XDevice *InstancePtr);
```

API and Naming Conventions

External Identifiers

External identifiers are defined as those items that are accessible to all other components 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 component in which they exist. The component name is prepended to each external name. An underscore character always separates the component name from the variable or function name.

External Name Pattern:

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

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

Component Based Source File Names

Source file names are based upon the name of the component implemented within the source files 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 component. A component is typically contained in multiple source files to allow parts of the component to be user selectable.

Source File Naming Pattern:

```
x<component name>.c                      main source file
x<component name>_functionality.c        secondary source file
```

Header Source Files (*.h)

The header files contain the interfaces for a component. There will always be external interfaces which is what an application that utilizes the component invokes.

- The external interfaces for the high level drivers (Layer 1) are contained in a header file with the file name format *x<component name>.h*.
- The external interfaces for the low level drivers (Layer 0) are contained in a header file with the file name format *x<component name>_l.h*.

In the case of multiple C source files which implement the class, there may also be a header file which contains internal interfaces for the class. 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<component name>_i.h*.

Device Driver Layers

Layer 1 and Layer 0 device drivers (i.e., high-level and low-level drivers) are typically bundled together in a directory. The Layer 0 device driver files are named *x<component name>_l.h* and *x<component name>_l.c*. The "*_l*" indicates low-level driver. Layer 2 RTOS adapter files typically include the word "adapter" in the file name, such as *x<component name>_adapter.h* and *x<component 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
```

High Level Device Driver API

High level 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.

Standard Device Driver API

Initialize

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.

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 prior to invoking the driver's Initialize 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 (of Layer 1, high-level drivers) such as Initialize() 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<component 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. Note that the low-level drivers of Layer 0 do not require or make use of the configuration information defined in these two files. Other than the memory-mapped location of the device, the low-level drivers are typically fixed in the hardware features they support.

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 making use of the LookupConfig functions of the device drivers.

xparameters.h

This source file centralizes basic configuration constants for all 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.
- Device base address for each instance
- Device interrupt assignment for each 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 `<component name>` does not include the preceding "X"):

```
XPAR_X<component name>_NUM_INSTANCES
```

Each device instance will then have multiple, unique constants defined. The names of the constants typically match the hardware configuration parameters, but can also include other constants. 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_<component name>_<component instance>_DEVICE_ID
XPAR_<component name>_<component instance>_BASEADDR
XPAR_<component name>_<component instance>_HIGHADDR
```

<component instance> is typically a number between 0 and (`XPAR_X<component name>_NUM_INSTANCES` - 1). Note that the system generation tools may create these constants

with a different convention than described here. Other device specific constants are defined as needed:

```
XPAR_<component name>_<component instance>_<item description>
```

When the device specific constant applies to all instances of the device:

```
XPAR_<component name>_<item description>
```

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_<instance>_<component name>_<component instance>_VEC_ID
```

Where INTC is the name of the interrupt controller component, <instance> is the component instance of the INTC, <component name> and <component instance> is the name and instance number of the component connected to the controller. Of course XPAR_INTC must have the other required constants DEVICE_ID, BASEADDR, etc. This convention supports single and cascaded interrupt controller architectures.

For the case where an interrupt controller is considered internal to a processor, the naming convention changes:

```
XPAR_<proc name>_<component name>_<component instance>_VEC_ID
```

Where <proc name> is the name of the processor.

x<component name>_g.c

The header file *x<component name>.h* defines the type of a configuration structure. The type 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
{
   Xuint16 DeviceID;
   Xuint32 BaseAddress;

   /* Other device dependent data attributes */

} X<component name>_Config;
```

The implementation file *x<component name>_g.c* defines a table, or an array of structures of `X<component 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*.

Example

To help illustrate the relationships between these configuration files, an example is presented that contains a single interrupt controller whose component name is INTC and a single UART whose component 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_INTC_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_UART_NUM_INSTANCES      1
XPAR_UART_0_DEVICE_ID        2
XPAR_UART_0_BASEADDR         0xB0001000
```

xintc.h

```
typedef struct
{
   Xuint16 DeviceID;
   Xuint32 BaseAddress;
} XIntc_Config;
```

xintc_g.c

```
static XintcConfig[XPAR_INTC_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 "1.00b", 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<component>_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 are standardized in a 32-bit word and the 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.

Interrupt Management

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

When using aLayer 1 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>_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 designed without the use of cache and MMU management. This is expected to be accomplished by the application or by an RTOS adapter.

Revision History

The following table shows the revision history for this document.

Table 1: **D ate**	*Table 2:* **V ersion**	*Table 3:* **Revision**
06/28/02	1.0	Xilinx initial release.
7/02/02	1.1	Made IP Spec # conditional text and removed ML reference.
7/31/02	1.2	Update to non-IP chapter template
12/17/04	1.3	Minor text updates/additions.