LatticeSC Software Drivers

User's Guide

Revision 0.2
February 7, 2006

For LatticeSC Driver Software Release 0.3.1

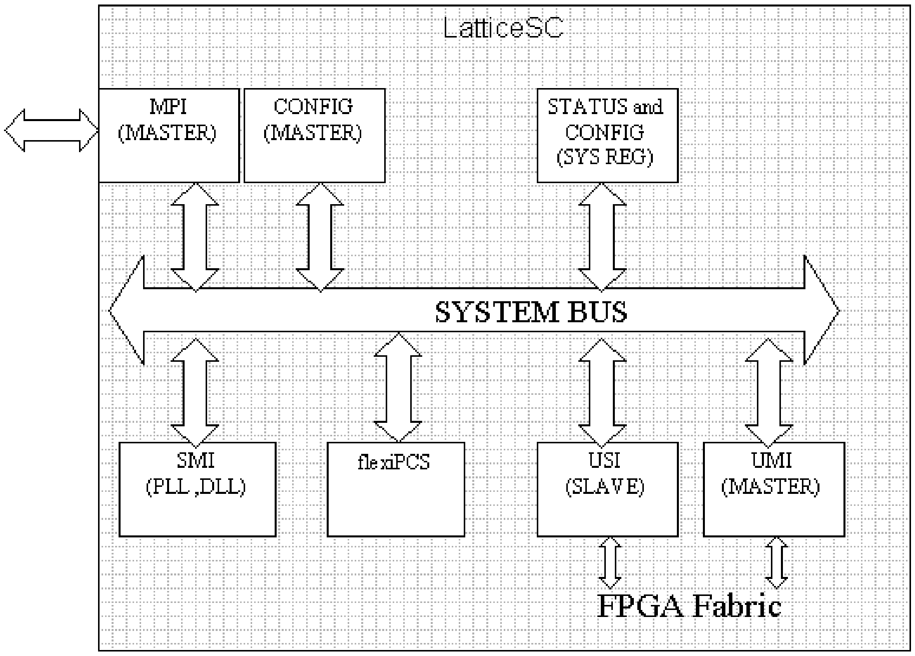
LATTICE SEMICONDUCTOR

Table of Contents

1 LatticeSC Software Driver Overview

1.1 Introduction

This guide describes the purpose, design and use of software drivers for the LatticeSC FPGA. The LatticeSC software drivers make it easy to access the large variety of built-in IP in the LatticeSC. Configuration and control of these built-in devices can be accomplished through register access via an internal system bus. The System Bus is linked with the outside world through an industry standard PowerPC PowerQUICC interface. The host processor of an embedded system therefore has direct access to the features of SC and the custom IP design implemented in SC fabric. This industry-standard interface to a host CPU affords many opportunities for software to assist with managing and controlling IP within the SC devices. Recognizing this potential, software drivers have been created to aid embedded software engineers with access LatticeSC resources and interface to registers in their designs.



1.2 Why Software Drivers?

The LatticeSC device is a system on a chip (SoC) with various built-in IP cores. With demands for shorter development times, a need arises to abstract the hardware details and provide the system developer with a higher level of functional control. This can be accomplished with software drivers. Common operations can now be done with a single function call instead of reading numerous hardware documents and experimenting with custom driver code. The software drivers abstract the hardware register details and provide a higher level of service. This allows the user to concentrate on their design and know that the built in features of SC can be accessed through these drivers.

The following memory map shows the available register mapped devices inside LatticeSC.

Begin Address	End Address	Size (B=Bytes)	Description
0x00000	0x0003F	64 B	System Bus registers.
0x00040	0x003FF	960 B	Reserved.
0x00400	0x007FF	1 KB	Sixty-four Serial Memory Interfaces. For access to PLL / DLL configuration memory and user control/status registers in FPGA. (Sixty -Four SMIs, 128 bits each)
0x00800	0x2FFFF	190 KB	User slave interface (USI) baseline. For pre-configuration usage of address range 0x10000-0x2FFFF, see the LatticeSC sysCONFIG Usage Guide (TN1080).
0x30000	0x364FF	25,856 B	Left PCS slave interface. See LatticeSC Family PCS Data Sheet for more detail.
0x36500	0x37FFF	6,912 B	Reserved.
0x38000	0x3E4FF	25,856 B	Right PCS slave interface. See LatticeSC Family PCS Data Sheet for more detail.
0x3E500	0x3EEFF	2,560 B	Reserved.
0x3EF00	0x3EFFF	256 B	Inter-Quad PCS slave interface. See LatticeSC Family PCS Data Sheet for more detail.
0x3F000	0x3FFFF	4 KB	Reserved.

The LatticeSC device driver package provides APIs (Application Programming Interfaces) to exercise the features of these internal register mapped devices.

1.3 Release Packages

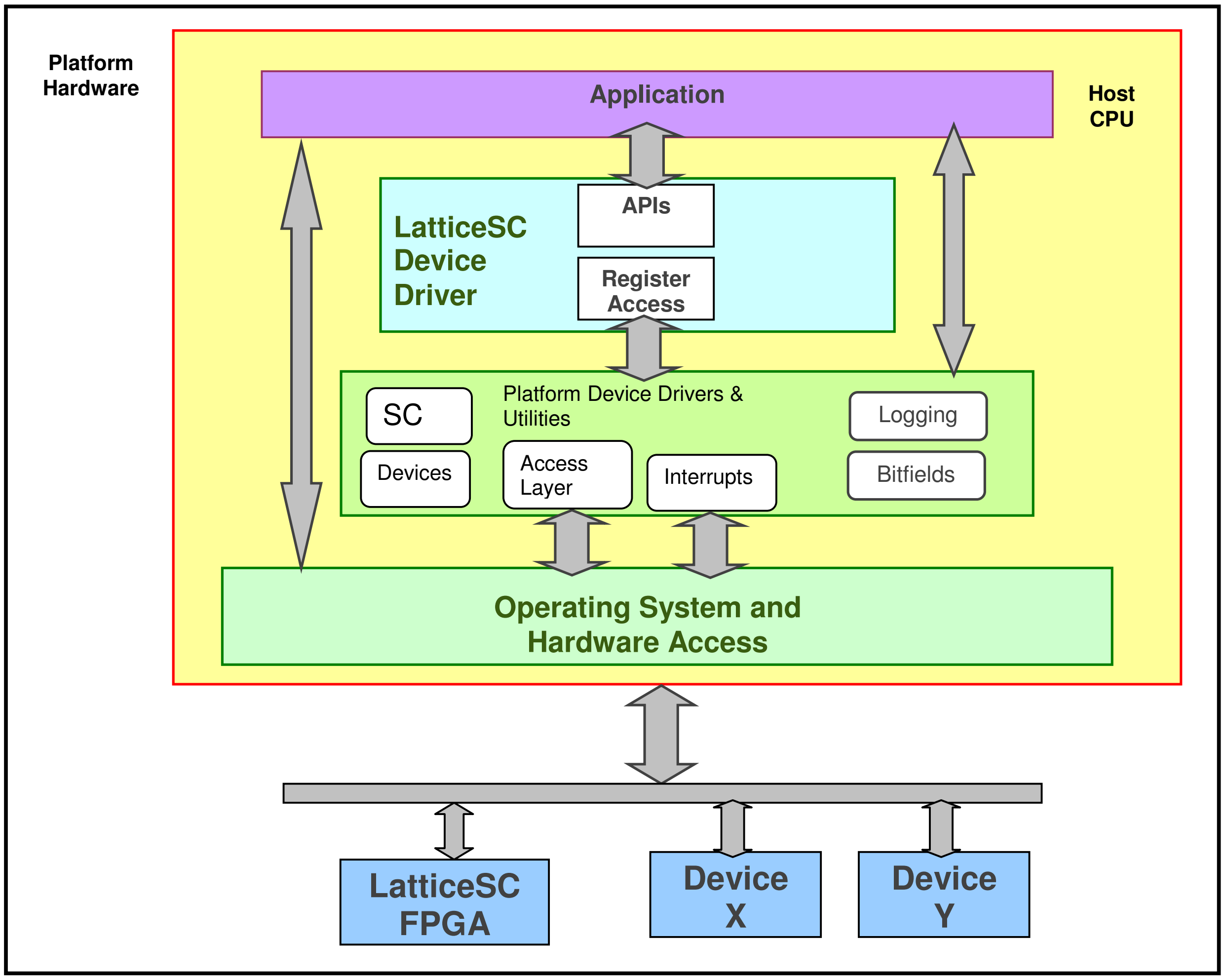
The LatticeSC Software Driver Package consists of software (source code), documentation and examples. There are two types of release packages: base Drivers and full Development. The base Drivers package contains the driver source code and all associated documentation. The full Development package contains the driver code, examples, platform support code and all documentation. The Development package is targeted to a specific hardware package and provides a turn-key solution, ie. the user can install the files, build the examples and run the resulting code on the targeted platform.

1.4 LatticeSC Driver Architecture

The software is designed as a hierarchical layering of software modules, abstracting the system and hardware details as one moves closer to the application level. This promotes higher layer software re-use among the different projects and platforms. Another benefit comes when testing, in that keeping the lower modules functionally independent from upper layer modules, the system can be built from the ground up and unit tested as each layer nears completion.

The following diagram shows how the software is broken into layers running on top of the Target OS (VxWorks, Linux, etc.):

- Operating System Access Layer
- Platform Device Drivers and Utilities Layer
- LatticeSC Device Driver Layer
- Application Layer



Target Operating System

The target platform is expected to use an embedded real-time operating system to handle all Host CPU hardware initialization and provide all standard services expected by an embedded computer system:

- DRAM, cache, memory manager setup
- Networking, serial ports, etc.
- Real-time Embedded support

- POSIX real-time extensions (with most features supported)

<u>Operating System Access Layer</u>

The OS Access Layer provides access to, or extends the capabilities of, the under-lying operating system resources. This may include implementing missing POSIX (or other) system calls to port higher layers of software, or adding driver modules into the OS to support custom devices on the platform.

- Ports Device Driver Library to specific platform and OS.
- Device Access:
  - Via PCI bus or direct memory-mapped access
  - Interrupt handling
- OS Services:
  - POSIX semaphores
  - POSIX timers
  - POSIX threads

<u>Platform Layer</u>

The Platform Layer contains all the drivers and services to configure and manage the support devices on the platform hardware. These drivers are targeted at low-level access and control of the devices. The drivers know about the device's hardware details, but do not implement functional policies – this is a role for higher level application software. Examples of platform drivers are:

- FPGA initialization
- Interrupt controllers
- Clock configuration and distribution
- Device reset control

Utilities provide at group of common services enabling applications and external client systems, that connect to the target platform, to extract information or control operations remotely. Some common utilities that are provided on all platforms:

- Logging
- Register access
- Network servers

<u>LatticeSC Device Driver Layer</u>

The Device Driver Layer contains all the APIs, in source code format, that can be built into a library for use by an application. The application can then access LatticeSC through these APIs. Features include:

- Implements the software APIs to configure, monitor and control IP in SC
- Compiles and links with application code (and an optional OS)
- Performs hardware access via Access Layer functions

<u>Application Layer</u>

The Application layer is the top-level entry point of execution. It is the written by the end user of LatticeSC to perform the operations needed in their platform.

- Test, demo or sample code that use the Lattice Device Driver APIs.
- Starts all execution.
- Implements the specific use of the LatticeSC

1.5 Driver Functionality Overview

The LatticeSC Software drivers provide functions to allow:

- Register Access
- Event Handling – interrupts and alarms
- Bitstream download
- Fabric Access (USI)
- PLL/DLL's control and configuration
- Quads (PCS/SerDes) control and configuration
- Communication Protocol control and configuration

Some of these functions are simple, such as register accesses, which merely changes bit states in a register. Other functions are more complex, requiring many of lines of code to handle all the various modes and options of a protocol. The driver programming details are covered in section 3 and also in the Software Drivers Reference Manuals.

1.6 Configurable Platform Support

The drivers are targeted to run on any embedded system (platform) that contains or access SC FGPAs. The platform needs to provide some basic services to the drivers so that the driver functions can read/write registers, catch interrupts and log diagnostic messages. The drivers have been written to access these platform resource through defined interfaces. The user of the drivers is required to implement these interfaces to port the driver code to their specific platform. This can be done in numerous ways depending on the purpose of the platform. For example, for testing and simulation the platform may be a PC application that merely directs all registers access to an internal memory array simulating SC's register map. For distributed processing, the read/writes may be turned into messages and sent over a network to a remote machine containing the SC. Or in the more typical embedded case, the read/writes are just direct SC memory accesses implemented with pointers. Either way, the drivers expect the platform to provide the following three services:

- Register Access – drivers need to read/write registers in FPGA
- Interrupt handler – driver event handler needs to be called when an SC interrupt occurs
- Utilities – general purpose tools, not specific to SC, but included in package to run on

Separating the platform hardware access from the internal driver functionality allows the same application and driver source code to be used on many different hardware platforms, since the driver code is not tied to any specific hardware platform.

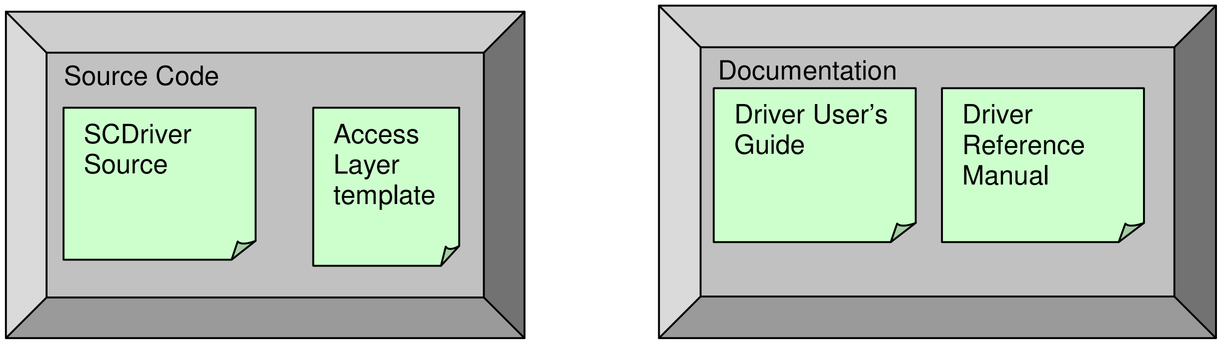
2 Installation Guide

2.1 Introduction
This section describes what is included in the LatticeSC Software Driver package, and how to obtain, package, install, and customize it for the development system being used.

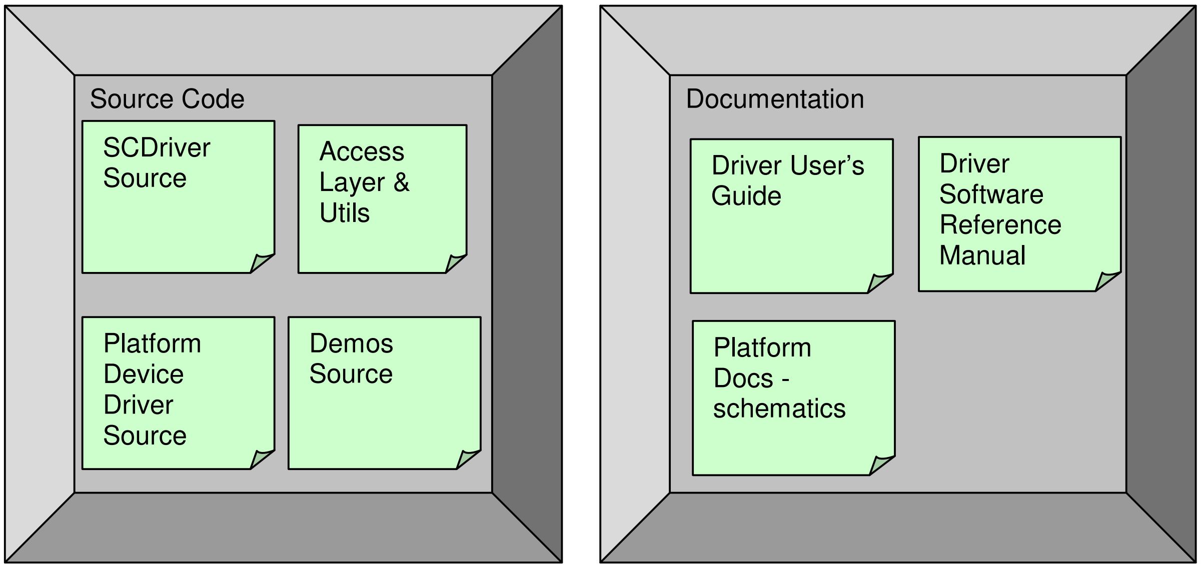
2.2 Release Packages
The LatticeSC Software Driver Package contains software (source code), documentation and examples. There are two types of release packages: base Drivers and full Development.

- The base Drivers package contains the driver source code and associated documentation.
- The full Development package contains the driver code, examples, platform support code and all documentation. The Development package is targeted to a specific hardware package and provides a turn-key solution, ie. the user can install the files, build the examples and run the resulting code on the targeted LatticeSC platform.

2.2.1 Drivers Package
This package contains the complete LatticeSC driver source code, but no supporting software that targets a specific OS, platform or build. This package allows a user to obtain and examine the driver source without dealing with the complexities of how it fits into a complete system. This package is useful for upgrading just the driver source if the all the demos are being ignored. The following figure illustrates the contents of the Drivers Package – the source code components and documentation components.



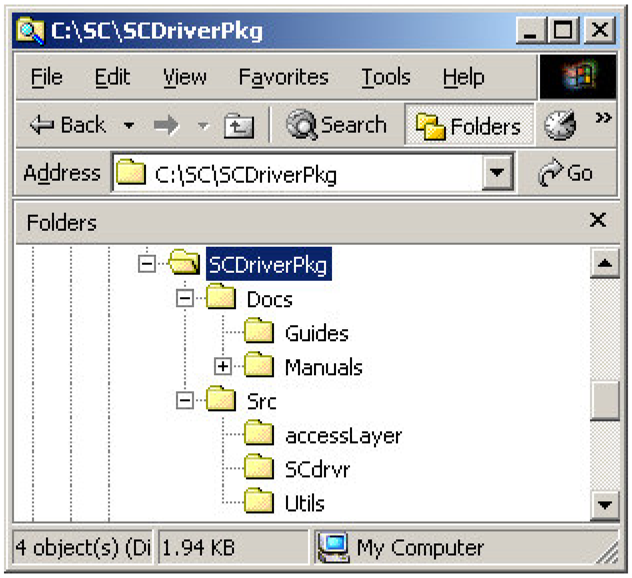
2.2.2 Development Package
The Development Package contains the complete LatticeSC driver source code along with all the supporting software that targets the specific hardware platform and builds under the specified operating system, including demo applications. This package allows a user to see how the driver fits into a complete, working system and build executable code. The following figure illustrates the contents of the Development Package – the source code components and documentation components.



2.3 Installation

The LatticeSC driver packages are available on the Lattice Semiconductor company web site on the LatticeSC products page, simply download the zip file and unzip it.

1. Go to LatticeSC FPGA products page at http://www.latticesemi.com/
2. Right-click on the driver package (SCDriver_Pkg_REL_0_3_1.zip) and choose "Save Target As" and select a place on your system, such as c:\SC
3. Unzip the downloaded file.
4. The installation creates the following directory structure:


C:\SC\SCDriverPkg
File Edit View Favorites Tools Help
Back Search Folders
Address C:\SC\SCDriverPkg Go
Folders
4 object(s) (Di 1.94 KB My Computer

```
SCDriverPkg
  Docs
    Guides
    Manuals
  Src
    accessLayer
    SCdrvr
    Utils
```

The top level directory SCDriverPkg will be created and populated with all the files and directories included in the package. The Docs/ directory contains the documentation for the release. Users guides and other documentation is available in the under Guides/ directory. An HTML reference manual for the driver APIs is located in Manuals/ directory. Click on index.html to open the reference manuals. The manuals are generated directly from the source code and provide cross-references and links to related items.

The Src/ directory contains the driver source code, a template to create the platform access layer, and source code for the utilities used by the drivers. The top level of Src/ contains the files required to customize building the code on a particular system with a specific tool chain. The file rules.make controls the build process and the file sysDefs.h adapts the source code to the platform, compiler and libraries of the target tool chain.

2.4 Environment Setup

Before compiling the code, one environment variable must be setup in your environment. The variable PLATFORM_DIR must be defined to point to the top level of the source directory, where the rules.make and sysDefs.h files are located. This variable is used by the makefiles and compiler to locate any project header files. All file accesses are relative to this location.

```
> set PLATFORM_DIR=c:\SC\SCDriverPkg\Src
```

2.5 Compiler settings

2.5.1 rules.make

The file rules.make is used to specify the exact compiler, linker and flag commands needed to build for a particular target platform. This file is included by every makefile in the project so the definitions are used by every make. The following example shows typical build rules for compiling with gcc.

```
#==========================================================================
# Generic Platform Build Rules Configuration File
#
# Makefile format.  Included into all makefiles for specifying the
# build tools, compiler options and search paths for header files.
#
# Configure this file to the specific build environment that the drivers
# and test programs are being built for.
#==========================================================================

# Specify the compiler, linker, libraries and flags your system needs
CC = gcc
CFLAGS = -c -Wall -g

CXX = g++
CXXFLAGS := $(CFLAGS)

LD = gcc
LDFLAGS = -nostdlib -r -Wl,-X
LIBS=

SC_DRVR_OPTIONS = -DENABLE_LOGGING
OPTIONS = -O $(SC_DRVR_OPTIONS)
```

2.5.2 sysDefs.h

The file sysDefs.h is included by every C source file. It allows the user to port the source code to a particular compiler. For example, if you were using a compiler for a system that didn’t support certain POSIX calls, then providing macros in this file will allow the driver source to compile. The following example code shows a sampling of types and defines that may need to be defined for the driver code if they are not provided by the compiler.

```
typedef int status_t;

#ifndef OK
#define OK 0
#endif

#ifndef ERROR
#define ERROR -1
#endif

/**
 * Time-out used for polling operations in which an event may never
 * happen and need to abort.  Choose a value that gives approx. 1/10 sec
 * delay when used as a loop counter.
 */
#define SC_TIMEOUT 1000

#define SC_USEC_DELAY(a) \
        {for (volatile int iiidelay=0; iiidelay<(a*2); iiidelay++);}

// Define missing POSIX data type definitions
typedef int  pthread_t;
typedef int  pid_t;
```

3 Programming Guide

3.1 Introduction

This section discusses the software development methodology. Since it is not known what OS, processor or target architecture the customer will be using, all LatticeSC Driver software, as well as the build tools, are designed to be platform independent. Platform and OS independence are a common theme throughout the driver architecture, software design, build tools and release setup.

The LatticeSC Driver Reference Manual Documentation should be consulted for specific API details. This guide will provide a cursory view of available methods to get an idea of how to use the driver, but the reference manual is the authority since it is generated from the source code.

3.2 Design Philosophy

The architectural design is a hierarchical layering of software modules, abstracting the system and hardware details as one moves closer to the application level. This keeps the modules functionally independent from upper layer modules, and the complete system can be built from the ground up and unit tested as each layer nears completion. The driver software design has been based on three fundamental guidelines:

- Use POSIX system calls for all OS interaction
- Abstract all hardware access for easier platform portability
- Object Oriented design of software

POSIX standardizes the set of system calls that an operating system provides to application code. Source code is then portable to other operating systems because the OS API's will be the same – same name and same parameters. Most modern, embedded operating systems support the standard and real-time POSIX extension calls, so adhering to POSIX will allow the driver code to be compiled on just about any customer's system.

Portability is a key requirement behind all LatticeSC drivers. This ensures that the driver software has the maximum usage among customer's platforms. The driver software does not directly access hardware or make assumptions about hardware access. A hardware access layer provides APIs to read/write hardware. The APIs are functions specifying a defined interface. The implementation of these defined API functions can then be modified by a customer to port to their hardware platform.

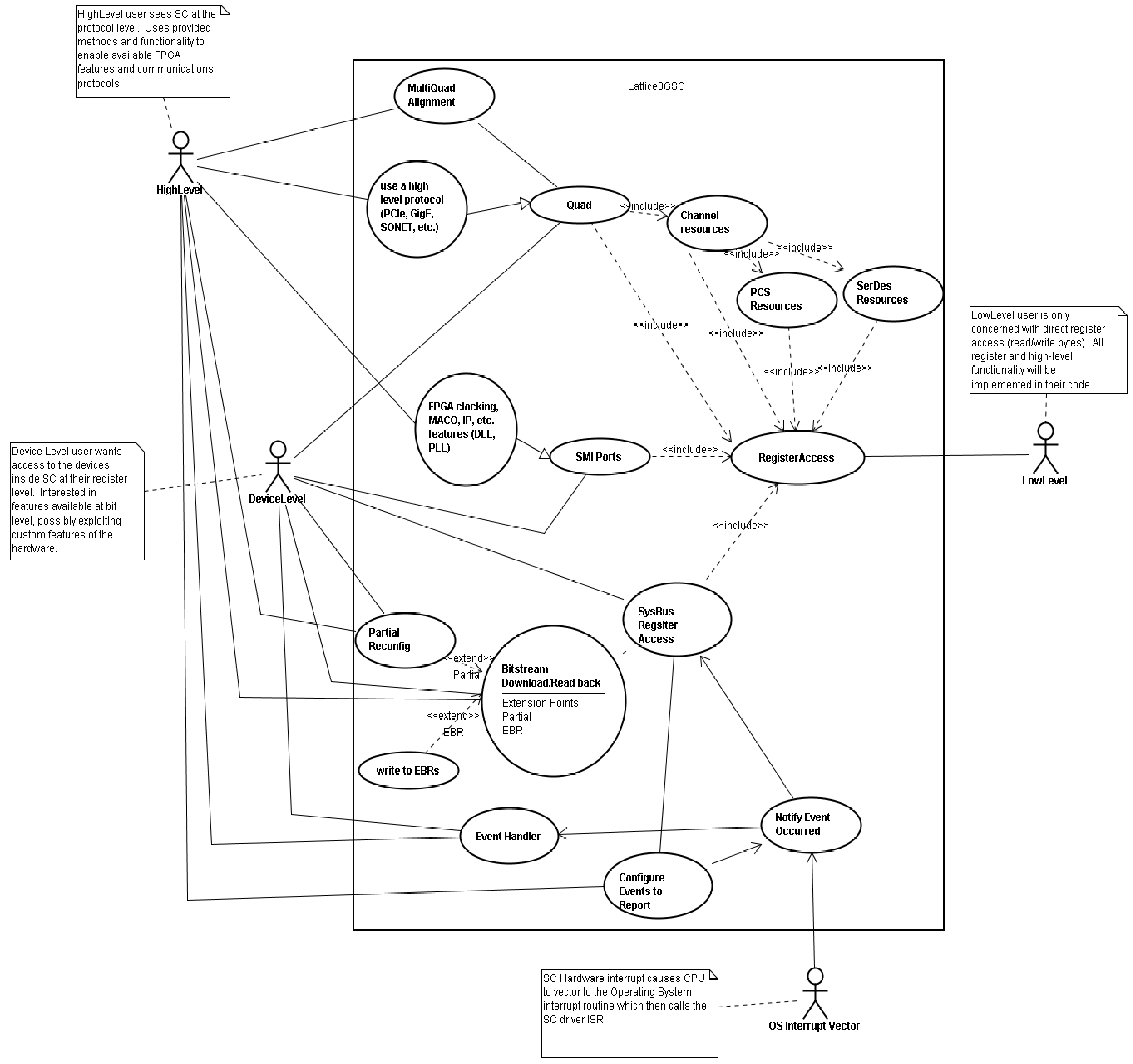
Object Oriented design provides a cleaner way of organizing the software to reflect the physical organization (hardware architecture) of the system. OO also allows a higher level of abstraction that simplifies the design and allows quicker comprehension of the design. Users can focus on just the interfaces to classes and not worry about the details of the implementation. The drivers are written in ANSI C and C++ since they are the most popular embedded programming languages, with C++ being Object Oriented.

C++ is used wherever possible. The values of C++ object-oriented programming are:

- Improved software design - information hiding through enforced public/private data attributes; interactions of software strictly controlled
- Easier to visualize architecture – code structure mimics physical objects in system
- Easier to extend features in future and provide backwards compatibility (over-load functions)
- Easier maintenance – object are self-contained, so changes to an object should not affect other, distinct objects.
- C++ allows direct memory access and inline assembly

3.3 Use Cases

The following UML diagram is an example how actors (stick figures) would interact with a LatticeSC FPGA and the objects and services they want to control. This model forms the basis for the driver implemetation and how the users will interact with LatticsSC via the drivers.


HighLevel user sees SC at the protocol level. Uses provided methods and functionality to enable available FPGA features and communications protocols.
Lattice3GSC
MultiQuad Alignment
HighLevel
use a high level protocol (PCIe, GigE, SONET, etc.)
Quad
<<include>>
Channel resources
PCS Resources
SerDes Resources
LowLevel user is only concerned with direct register access (read/write bytes). All register and high-level functionality will be implemented in their code.
FPGA clocking, MACO, IP, etc. features (DLL, PLL)
SMI Ports
RegisterAccess
LowLevel
Device Level user wants access to the devices inside SC at their register level. Interested in features available at bit level, possibly exploiting custom features of the hardware.
DeviceLevel
SysBus Regsiter Access
Partial Reconfig
<<extend>>
Partial
Bitstream Download/Read back
Extension Points
Partial
EBR
<<extend>>
EBR
write to EBRs
Event Handler
Notify Event Occurred
Configure Events to Report
SC Hardware interrupt causes CPU to vector to the Operating System interrupt routine which then calls the SC driver ISR
OS Interrupt Vector

The objects inside the LatticeSC show the functions that the driver exposes to the users. Some functions build on the features of other functions. Each object is implemented as its own class contained in the LatticeSC class.

One point to note is that the LatticeSC driver code, as depicted in the above figure, is not a stand alone module. It is like a library that needs a framework to be instantiated in. This framework, or software infrastructure, is called the Platform. The driver code expects the platform to provide register access to the SC hardware and provide useful utilities. Details of the platform will be shown in a later section, but for now be aware that first some form of framework needs to be created by the user so the driver methods have access to the hardware.

3.4 Example Program

The following example code shows how the platform code and LatticeSC driver code can be used in a real system. The first section (A) deals with creating a Platform object and utilities that will be used by the drivers. The second section (B) shows instantiation of the LatticeSC driver and use of some of its methods.

```
/***************************************************************
 * Example code showing use of LatticeSC drivers to download a
 * bitstream to the FPGA and then access registers in the fabric.
 ***************************************************************/
```

A

```
// Create a log for all driver messages
pmem = (uint32_t *)malloc(1024*1024);
EventLog myLog("demo_log", pmem, 1024*1024);

// Create access to the hardware Platform the SC is on
Platform SC_Brd(&myLog, "SC Eval Board");

// Now get the register access method used by the platform
pSC_regs = SC_Brd.pDevs->pSC1->getRegisterAccess();

// And create a LatticeSC device driver to access it
LatticeSC mySC(&myLog, pSC_regs);

// Create an EventHandler to service interrupts from the SC
AppHandler myHandler(&mySC);


// Install an Interrupt Handler for the SC.
SC_Brd.pDevs->pSC1->isrConnect(AppHandler::ISR, (void *)&myHandler);

/* Enable platform interrupts - everything is ready now */
SC_Brd.enableInterrupts();
```

B

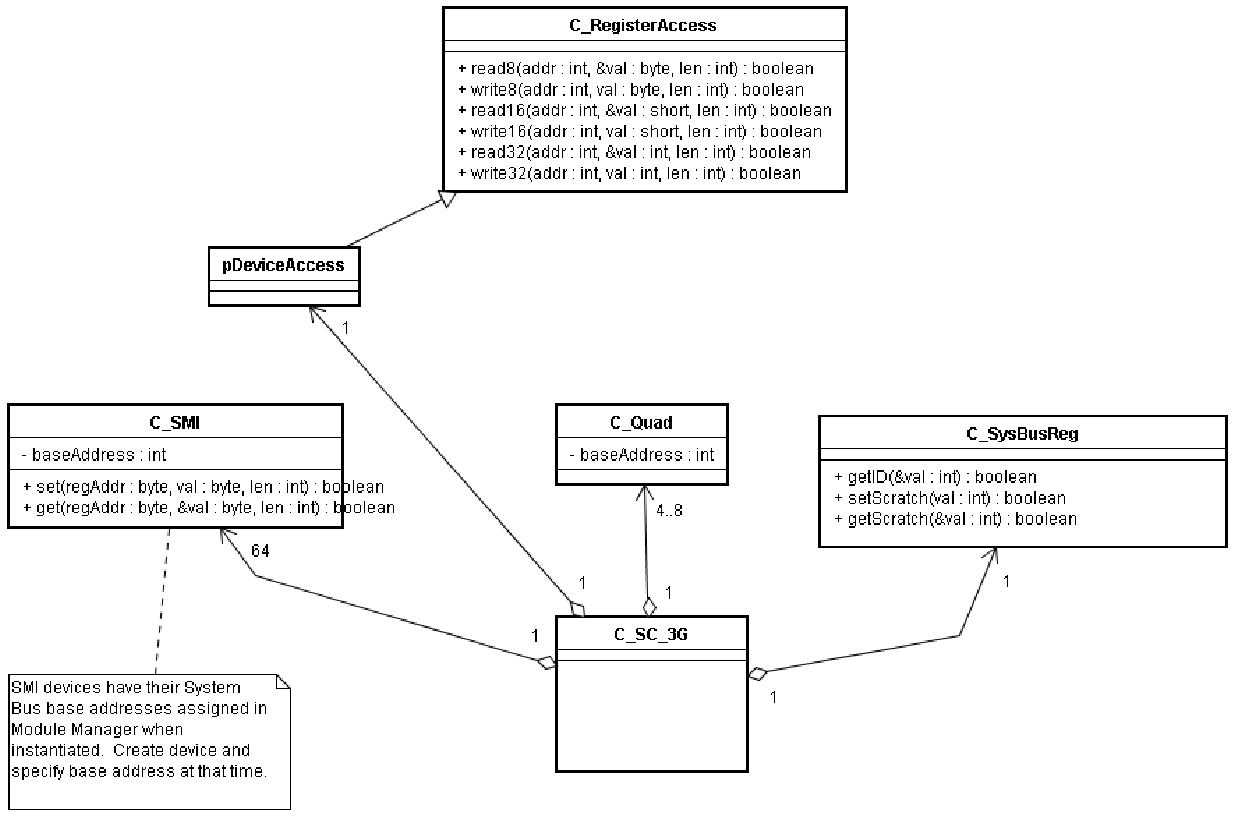
```
// Now create a BitStream object to program the SC with
BitStream bs(&mySC);

// Download .bit file using interrupts to catch any errors
bs.download("demo.bit", BitStream::SingleIntr);

// Access the SystemBus FPGA Device ID Register and display it
cout<<"ID register = 0x"<<hex<<mySC.pSysBus->getDeviceID()<<endl;

// Write to a register in the design to start a periodic timer and enable
// SC to generate interrupts back to the driver
mySC.pReg->write8(0x806, 0x05);  // start a periodic timer
mySC.pSysBus->bit->set(SysBusCtrlBits::en_mpi_usi_irq, 1);


 // Creating PLL at SMI 0x410 with clkin=133MHz
 PLL pll410(&mySC, 0x410, 133.0, PLL::PhaseShiftToClkop, 10);

 pll410.setClkopDiv(8, true);  // divide clkop freq output by 8
 pll410.setClkosPhase(60);  // set clkos phase shift 60 degrees from clkop

 // Add QUAD360
 mySC.addQuad(LatticeSC_3G::QUAD360);
 // Get comma mask register setting in the Quad
 val = mySC.QuadList[LatticeSC_3G::QUAD360]->
                                  bit->get(QuadBits::udf_comma_mask_lo);
```

In the above code, the steps are:

- Create an event logging object to record all diagnostic messages the driver and application code issue during operation.
- Create the Platform because the LatticeSC FPGA is a device contained in the SC_Brd hardware platform. The Platform contains the access methods to the hardware devices.
- Create an SC object, mySC, and give it access to the EventLog and the hardware access methods to use to read/write SC registers.
- Enable Platform Interrupt Service Routines (ISR) and interrupts for devices.
- Create a BitStream object to configure the LatticeSC FPGA
- Access registers in the design once the device is configured
- Change PLL settings
- Read a register in the Quad 360

3.5 LatticeSC Class

The following UML class diagram shows the structure of the LatticeSC class (more specifically the derived LatticeSC_3G class). The class is a container, holding references to objects contained in a LatticeSC FPGA.


C_RegisterAccess
+ read8(addr : int, &val : byte, len : int) : boolean
+ write8(addr : int, val : byte, len : int) : boolean
+ read16(addr : int, &val : short, len : int) : boolean
+ write16(addr : int, val : short, len : int) : boolean
+ read32(addr : int, &val : int, len : int) : boolean
+ write32(addr : int, val : int, len : int) : boolean
pDeviceAccess
1
C_SMI
- baseAddress : int
+ set(regAddr : byte, val : byte, len : int) : boolean
+ get(regAddr : byte, &val : byte, len : int) : boolean
64
C_Quad
- baseAddress : int
4..8
C_SysBusReg
+ getID(&val : int) : boolean
+ setScratch(val : int) : boolean
+ getScratch(&val : int) : boolean
1
1
1
1
1
C_SC_3G
SMI devices have their System Bus base addresses assigned in Module Manager when instantiated. Create device and specify base address at that time.

Major devices (objects) in an SC are the SystemBus (and associated registers), SMI ports to other devices (PLLs and DLLs), and Quads (which each contain 4 flexiPCS channels). A LatticeSC_3G class is currently the only available class of LatticeSC to instantiate. It is named 3G to indicate the SERDES are 3.125 gigabit/sec. Throughout this document, all references to an SC class or object should be interpreted as using the LatticeSC_3G class.

The constructor for a LatticeSC_3G object is:

```
   LatticeSC_3G(EventLog *pLog, RegisterAccess *pHdw, uint8_t mpiBusWidth);
```

The `EventLog` and `RegisterAccess` parameters point to objects already created in the Platform object. These two objects are covered in later sections, but for now understand that they need to be instantiated as part of the Platform before creating an SC object. The `mpiBusWidth` parameter specifies the data width of the SC MPI pins wired to the host CPU data bus. This can be 8, 16 or 32 bits and is used for downloading a fast as possible using the maximum data width. Most register accesses are always done in 8 bit accesses.

Once an SC object is created, the user can directly access any register in any device in the SC using direct register read/writes. This type of access would be performed by the Low-Level user in the previous use case model. Methods for Device-Level access are available through SysBusCtrl, SMI and Quad classes that are contained in the SC class.

3.5.1 Register Access

Direct register access uses the methods in the `RegisterAccess` class (obtained from the Platform) to read and write registers by their SystemBus offset. The methods in RegisterAccess are simply forms of:

```
   read8(addr, val)
   write8(addr, val)
```

where the data transfer size is 8, 16 or 32 bits. The `addr` parameter is the 18 bit address of the register on the SystemBus. Any device or IP register is accessible by its SystemBus address. At the lowest level, all driver methods really just do read/writes to a SystemBus address. All the remaining driver methods simply are wrappers and helper functions that abstract the hardware details (or encapsulate a defined sequence of operations).

The following code will read the JTAG DeviceID of the SC FPGA using direct register access:

```
   mySC.pReg->read32(0x0000, id);
```

The variable id will contain the 32bit value on return. The method returns true if the read was successful or false if the access failed (detected by the Platform access layer code, and may not be valid on all platforms – implementation specific).

For details on the SC FPGA register map and direct register access to devices, see the “*LatticeSC MPI/System Bus Application Note,TN1085”* and the LatticeSC data sheet.

3.5.2 System Bus Registers

The SysBusCtrl class is a “helper” class that provides methods to access the fixed control registers of the internal SystemBus. The method names provide information as to what is being done, rather than just supplying a hard-coded address. For example, the following code is more meaningful than the direct register access example above:

```
   id = mySC.pSysBus->getDeviceID();
```

See the SysBusCtrl class in the Reference Manuals for a complete list of methods.

3.5.3 SMI

The LatticeSC FPGA contains a second type of data bus that interconnects some of the built-in devices. This bus is called the Serial Management Interface (SMI) bus. It connects all the PLLs and DLLs to the SystemBus via a 1 wire data bus (and possibly User Logic). The SMI address range on the SystemBus maps register accesses to an SMI device’s serial protocol. SMI is a form of device register access, so it is derived from the RegisterAccess class to perform the hardware access as standard register reads or writes. The SMI register reads and writes over-ride the standard reads and writes to implement special handling of data going to the SMI bus. The SMI only supports 8 bit data, so only read8() and write8() methods are supported. Use of the SMI is basically the same as direct register access.

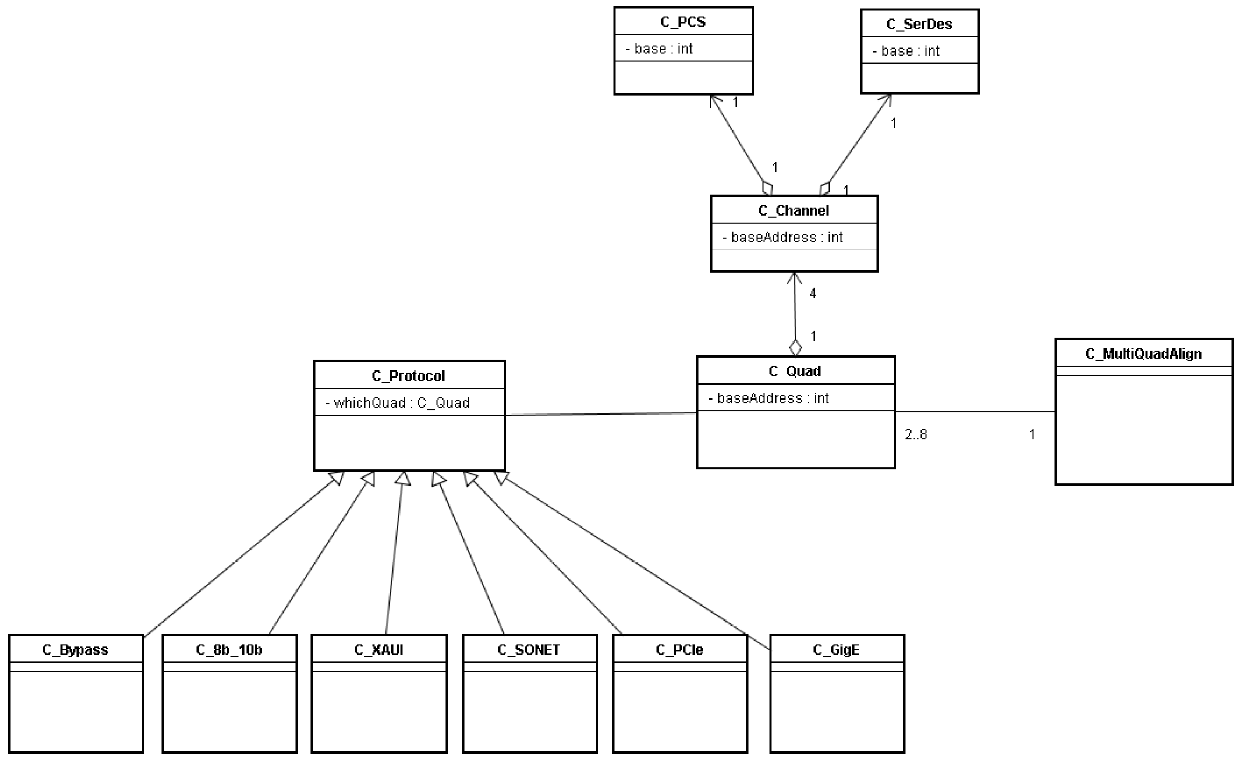
```
   MySC.pSMI->read(0x410, val);
```

There are 64 SMI ports on the SystemBus. Each port occupies 16 contiguous bytes of the SystemBus memory. The SMI port bases are therefore at 0x400, 0x410, 0x420…0x7f0. Reads and writes are straight-forward and the SMI methods handle data conversion and delays required by the SMI protocol.

The SMI class is mostly used as an internal helper class for register accesses by PLL and DLL classes. User IP could be attached to the SMI bus, but it is probably easier to use the USI interface to the SystemBus.

3.5.4 Quads

The LatticsSC_3G class has provisions for up to 8 Quads. Not every SC FPGA device will have 8 Quads nor will every Quad on a die will be pinned out to balls on all packages. Therefore the user needs to specifically create the Quads they intend to use and add them to the SC class. The user is responsible for understanding which Quads are available in the package they are targeting. These Quads are kept in a list and can be accessed through the list **`mySC.QuadList[QuadId_t].`** When a Quad is created, all the constituent parts are also created. A Quad is composed of 4 channels, with each channel having a PCS and SERDES (in other documentation, references to a flexiPCS can be thought of as a channel in this context).



The Quad class, as shown above, contains 4 Channel classes. Each Channel class contains a PCS class and a SerDes class. In the future, higher layer protocol classes will be bound to a Quad and control that Quad, since an entire Quad must operate in the same mode, ie. Gigabit Ethernet or SONET or PCI-Express.

The user creates access to a Quad with the `addQuad()` method. Access to the Quad, channel, PCS or SERDES registers is then through bitfield methods of the Quad, Channel, PCS or SERDES classes. The following example shows how:

```
   mySC.addQuad(LatticeSC_3G::QUAD360));  // new Quad object created
```

```
mySC.QuadList[LatticeSC_3G::QUAD360]->ChannelList[0]->pPCS->bit-
>set(ChanPCSBits::enable_cgalign, 1);
```

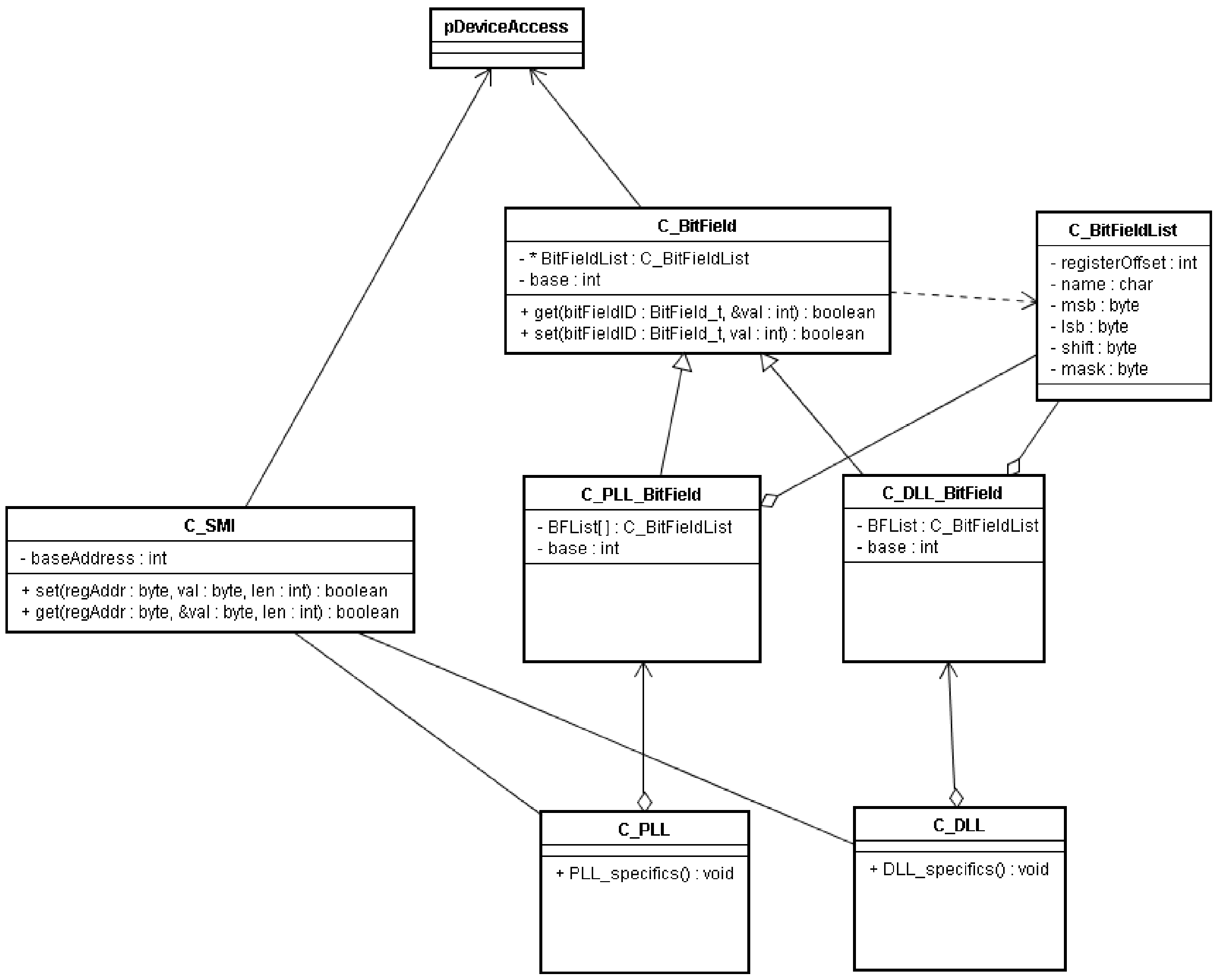
The classes also have methods to return a pointer to a Quad, PCS or SERDES so the command lines don't have to be so lengthy.

3.6 Higher-Level IP

The real benefit and power of any device driver is to provide methods that make it simple to do complex operations. The PLL and DLL classes provide methods to allow the user to change the frequency and phase of output signals with a simple function call that hides all the complex computations and inter-relations of the hardware to achieve the end results.

3.6.1 PLL

The LatticeSC has 8 built-in analog Phase Locked Loops (PLLs). These PLLs have register settings that control their operation. The registers are mapped to the SMI bus. The LatticeSC Driver provide a PLL class that has methods to dynamically control and fine-tune the operation of the PLL.


pDeviceAccess
C_BitField
- * BitFieldList : C_BitFieldList
- base : int
+ get(bitFieldID : BitField_t, &val : int) : boolean
+ set(bitFieldID : BitField_t, val : int) : boolean
C_BitFieldList
- registerOffset : int
- name : char
- msb : byte
- lsb : byte
- shift : byte
- mask : byte
C_SMI
- baseAddress : int
+ set(regAddr : byte, val : byte, len : int) : boolean
+ get(regAddr : byte, &val : byte, len : int) : boolean
C_PLL_BitField
- BFList[] : C_BitFieldList
- base : int
C_DLL_BitField
- BFList : C_BitFieldList
- base : int
C_PLL
+ PLL_specifics() : void
C_DLL
+ DLL_specifics() : void

The PLL class methods allow changes to phase, frequency dividers, delays and output modes. All these changes are done through BitField operations as shown in the above class diagram. The following code illustrates creating access to a PLL that has been assigned SMI port 0x410 in the FPGA design:

```
    // Creating PLL at SMI 0x410 with clkin=133MHz
    PLL pll410(&mySC, 0x410, 133.0, PLL::PhaseShiftToClkop, 10);

    pll410.setClkopDiv(8, true);  // divide clkop freq output by 8
    pll410.setClkosPhase(60);  // set clkos phase shift 60 degrees from clkop
```

Refer to the Software Driver Reference manual (Docs/Manuals/index.html) for a complete explanation of the PLL class.

NOTE: The PLL should be reset, through the IP block's reset port, whenever a change occurs. The software BitField reset bit is not working in the current hardware. The reset port should be wired to a register in the user's design and toggled, through register access, after a PLL change is made through the driver.

3.6.2 DLL

The LatticeSC has a variable number of built-in Digital Locked Loops (DLLs). These DLLs have register settings that control their operation. The registers are mapped to the SMI bus. The LatticeSC Driver provide a DLL class that has methods to dynamically control and fine-tune the operation of the DLL.The DLL class is very similar to the PLL in structure. The differences are in the methods that are specific to DLL features versus PLL features. The DLL methods concentrate more on phase and duty cycle adjustment. Again, the reference manual has all the details of the available methods.

3.6.3 SERDES Communication Protocols

These are currently under development. Eventually instantiating a class of Gigabit Ethernet will set the entire Quad to the necessary settings for Gigabit Ethernet protocol and provide status and configuration methods.

3.6.4 Bitstream Class

The BitStream class is used to download a bitstream file into the FPGA during configuration. The mode pins must select the MPI as the configuration device. The BitStream object is given access to the particular LatticeSC device when it is created. Once created, downloading can be done at any time.

The driver supports both .bit and .rbt bitstream file formats as produced by bitgen. The bitstream data can either be read from a file or read from a memory buffer (ROMed). Downloading can proceed using polling to verify that each byte is written correctly, or can use interrupts to be notified if an error has occurred. Either method is a system performance choice. Downloading with interrupts takes less than a second. The download method returns detailed error codes if anything fails. Downloading also programs the PCS/SERDES registers with their initialized values specified in IPExpress.

The following example code shows downloading a bitstream file using polling (no interrupts):

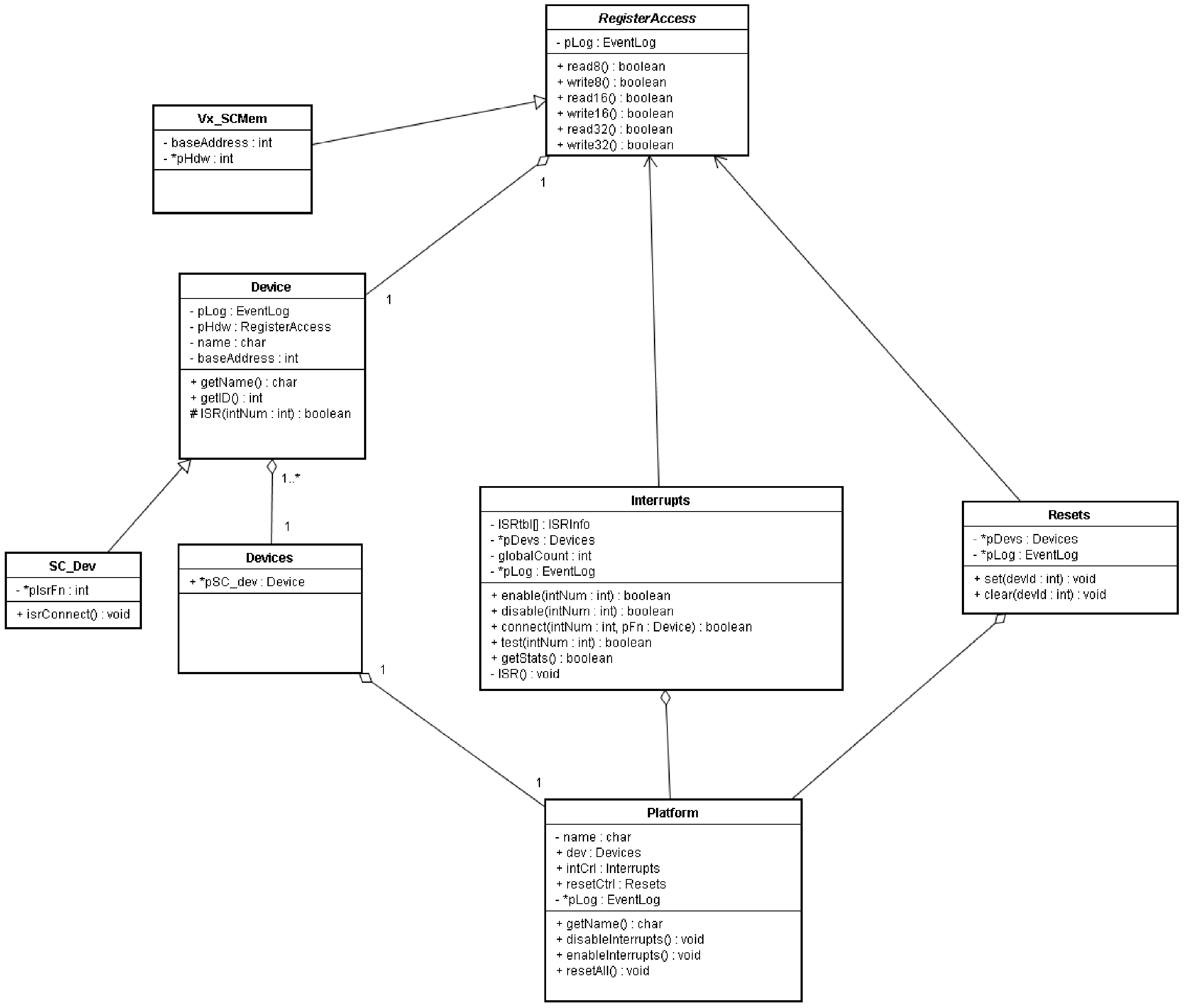
```
   // Now create a BitStream object to program the SC with
   BitStream bs(&mySC);

   // Download .bit file using polling to verify no errors
   status = bs.download("demo.bit", BitStream::SinglePoll);
```

It is important to remember that the MPI interface must be present in the design that is being downloaded. Even though the MPI and SystemBus may be accessed pre-config by Mode pin settings, MPI access will not be present after Done goes high if it is not in the design. This means the drivers will stop working (accessing) once the FPGA is configured.

Details on MPI bitstream downloading can be found in the "*LatticeSC sysCONFIG Usage Guide,TN1080"* and in the Driver Reference Manual.

3.7 Platform Access Layer

The Platform Access Layer contains classes that provide base functionality to access the hardware devices on the platform. These classes are not part of the core LatticeSC driver classes. These classes provide the infrastructure to access hardware devices. The Platform class provides the containing class that holds all the devices, resets and interrupt objects found on a platform.



The Devices class is a list of physical devices that exist on the platform. Each device has its own specific class. Each device class is derived from the Device class so they all share a common set of base functions and can be added to the Device list. The key characteristics of a device are:

- register access methods
- access to an EventLog
- a base address
- an ISR (if needed)

The SC_Dev object(s) creates the instance of the class to provide register access to the physical device. The SC class is responsible for the actual register reads and writes, and inherits methods from the RegisterAccess class. The SC class may do direct memory read/writes to the device, or it may simulate them or perform accesses through another device (JTAG port). The specific mechanism depends on the platform. Deriving from RegisterAccess guarantees that all devices implement the common read/write methods.

The Resets class provides a means (or a place holder) to assert resets to devices on a platform. Usually this would be done by setting a bit in a CPLD register that drives the physical reset pin of a device. Some platforms may not have means to reset devices, therefore the implementation of this class would have empty methods. Platforms that can provide reset functionality use the Devices class, which lists all devices on the platform, to point to the specific device and its RegisterAccess methods to read/write registers to perform the reset.

The Interrupts class provides the means to manage device interrupt propagation through the platform and operating system. The challenge is to provide a uniform method for various platform interrupt implementations.

3.7.1 Register Access

The RegisterAccess class is probably the most important class. It provides the platform specific methods to read and write hardware registers. The user of the drivers needs to create their own sub-class (derived from) the “template” RegisterAccess class. In their specific class, they would implement the actual code that does the reads and writes to the hardware. The LatticeSC drivers do all hardware accesses through these methods. The following are examples of the methods that exist in RegisterAccess:

```
   virtual bool read8 (uint32_t addr, uint8_t &val)=0;
   virtual bool write8 (uint32_t addr, uint8_t val)=0;
   virtual bool read16 (uint32_t addr, uint16_t &val)=0;
   virtual bool write16 (uint32_t addr, uint16_t val)=0;
   virtual bool read32 (uint32_t addr, uint32_t &val)=0;
   virtual bool write32 (uint32_t addr, uint32_t val)=0;
```

These are pure virtual functions and must be overridden by the user’s new, platform specific class that is derived from RegisterAccess. The child class then implements the platform specific means of reading and writing 8, 16 and 32 bits to an address. The driver code uses a reference to a RegisterAccess class to invoke the read/write method. The user provides their class when instantiating LatticeSC, which then gets registered in the driver and invoked every time the driver code does a read/write. The following code shows how a derived class `pSC_regs`, which does the actual hardware read/writes, is registered with the LatticeSC driver:

```
   LatticeSC mySC(&myLog, pSC_regs);
```

Every platform that the LatticeSC drivers are to be used on, must implement a class derived from RegisterAccess. The following snippit of code shows how the class SC_VxMem is used to provide access to the LatticeSC FPGA that is directly addressable from application code (flat memory model of VxWorks) at address 0x20000000.

```
   /*
    * LatticeSC Register Access on the VxWorks SC900 Platform.
    * This class extends the base class to provide the exact methods
    * to access the hardware registers in SC on the SC900 Comm board.
    * The hardware particulars are that the device is memory mapped at
```

```
 * 0x20000000, the MPI data bus is only 16 bits wide (900 ball package
 * limitation).  The read/write32 methods therefore need to do two
 * 16 bit accesses.  A 32 bit access will cause the SC to assert
 * TEA (Transfer Error Acknowledge).
 */

/**
 * Create a real instance of an object to access registers in an SC.
 */
SC_VxMem::SC_VxMem(EventLog *pLog, uint32_t memAddr) : RegisterAccess(pLog)
{
   baseAddr = memAddr;  // baseAddr is the 32bit CPU address of SC
   pHdw = (void *)memAddr;  // make it into a pointer
}


/**
 * Read 8 bits from an SC hardware register.
 * @param offset address of device register to read from
 * @param val location of storage for data read
 * @return true; error in reading will cause hardware exception
 */
bool SC_VxMem::read8(uint32_t offset, uint8_t &val)
{
   if (!allowed)
         return(false);

   val = *((uint8_t *)pHdw + offset);  // directly read using pointer

   RegisterAccess::read8(offset, val);  /* Use base class to log access */

   return(true);
}
```

An instance of this class needs to be created and then it address passed to the LatticeSC driver constructor and then the driver will have full access to the SC hardware.

3.7.2 Devices

Not part of this release package. To be discussed in detail in future releases.

3.7.3 Resets

Not part of this release package. To be discussed in detail in future releases.

3.7.4 Interrupts

Not part of this release package. To be discussed in detail in future releases.

3.8 Utilities

These classes are not directly a part of the LatticeSC device driver, in that they do not represent physical devices in an FPGA, but they are used in the driver operation. The EventLog and the BitFields are used extensively through out the driver. The EventLog records driver operation, and the BitFields enable easier access to registers.

3.8.1 Event Log

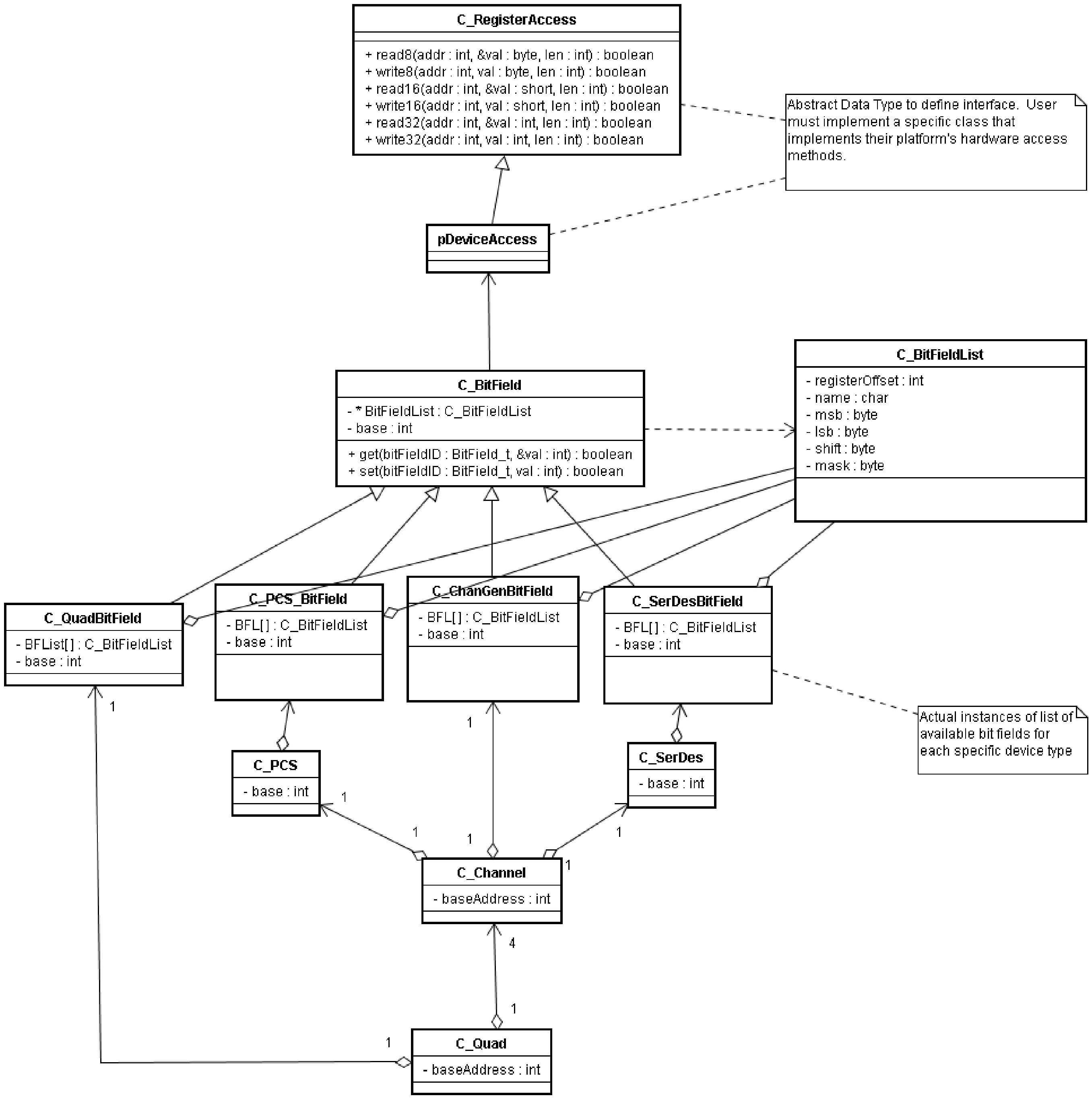
The EventLog is used to store messages from the driver in chronological order, as efficiently as possible, in memory for diagnostic purposes. The main reason for this utility is to record what operations the driver is doing. For example, when changing the frequency of a PLL (if logging is enabled), all register accesses will be recorded in the log and can be displayed to determine if the correct registers were written at the correct times. It is also useful to see what objects are created and destroyed during the life-cycle of an application.

For production or space savings, an “empty” EventLog class exists that can be used in place of EventLog. The NoEventLog class derives from EventLog so its signature is identical, but overrides all functions with empty functions so no operations are performed. The compiler may even optimize out most of the code since nothing is done in any function.

Event Log Control:

3.8.2 BitFields

The BitField class is used in place of the traditional `#defines` or macros seen in C embedded programming. The BitField class encapsulates all the information and operations needed to get or set the value of a bit field in a register. (A bit field being a contiguous set of bits in one register.) The internal class methods deal with reading the proper register, creating a mask to extract the bit(s) of interest and preserving the surrounding bit values during writes. The list of available bit fields in a particular device (Channel, PCS, PLL, etc.) is automatically generated from datasheet information, so the names of bit fields should be identical to what is found in other LatticeSC documentation. As seen in the figure below, classes derive their own specific BitField sub-class from the parent Bitfield and supply their own BitFieldList that defines the bits and registers that can be accessed. Users of these classes then only need to `set()` or `get()` the named bit field and the bit(s) value(s) are operated on accordingly, hiding all the details of the shifting, masking, reading and writing.


C_RegisterAccess
+ read8(addr : int, &val : byte, len : int) : boolean
+ write8(addr : int, val : byte, len : int) : boolean
+ read16(addr : int, &val : short, len : int) : boolean
+ write16(addr : int, val : short, len : int) : boolean
+ read32(addr : int, &val : int, len : int) : boolean
+ write32(addr : int, val : int, len : int) : boolean
Abstract Data Type to define interface. User must implement a specific class that implements their platform's hardware access methods.
pDeviceAccess
C_BitField
- * BitFieldList : C_BitFieldList
- base : int
+ get(bitFieldID : BitField_t, &val : int) : boolean
+ set(bitFieldID : BitField_t, val : int) : boolean
C_BitFieldList
- registerOffset : int
- name : char
- msb : byte
- lsb : byte
- shift : byte
- mask : byte
C_QuadBitField
- BFList[] : C_BitFieldList
- base : int
C_PCS_BitField
- BFL[] : C_BitFieldList
- base : int
C_ChanGenBitField
- BFL[] : C_BitFieldList
- base : int
C_SerDesBitField
- BFL[] : C_BitFieldList
- base : int
Actual instances of list of available bit fields for each specific device type
C_PCS
- base : int
C_SerDes
- base : int
C_Channel
- baseAddress : int
C_Quad
- baseAddress : int
1
1
1
1
1
1
1
1
4
1
1

As the above class diagram shows, BitFields and BitFieldLists are used by nearly every device class to provide named access to its registers.

3.8.3 Versions

The Version class is a utility that records the date, time and version number of a particular software module. This information can be displayed at run-time to verify the version of the software that is executing. To find out the version of LatticsSC driver software, include the following in your application code:

```
   char s[Version::MAX_STR_SIZE];
   cout<<"SC Driver Ver: "<<mySC.pVersion->getVersionStr(s)<<endl;
```

The version number is contained in the file versionNum.h in the particular software module directory. This class should not need any modification by a user, and should be portable to any system.

4 FPGA Implementation Considerations

4.1 Introduction

The LatticeSC Drivers need access to the SystemBus in order to do any register accesses. The SystemBus needs to be made available in the FPGA design. In addition to instantiating a SystemBus, the devices to access must also be instantiated. PLLs, DLLs and other IP must also be instantiated and wired to the SystemBus so their registers are accessible to the driver. This section gives hints as to what needs to be setup for proper device operation. See the *"LatticeSC MPI/System Bus Application Note,TN1085"* for details.

4.2 SystemBus Access

4.2.1 Instantiation

The SystemBus needs to be instantiated in the design for register access to device registers. The SystemBus can be created using the IPExpress tool.

4.2.2 MPI

The MPI must be in the down-loaded bitstream in order to have MPI access to the SystemBus after the device is programmed. If the mode pins select the MPI interface, the MPI interface is available and active when the LatticeSC is first powered on. If a design is downloaded that contains the MPI interface of the SystemBus, then the routing from the dedicated MPI I/O pins remains after DONE goes high. If the design does not contain the MPI, these routing resources will not be maintained after DONE goes high (the pins can be used as general purpose I/O, and lose connection to the MPI interface ports). Therefore, to access the SystemBus via the MPI, you must have the MPI instantiated in the downloaded design. This is done in IPExpress, when creating the SystemBus, check the "Enable MPI" box.

The mpi_rst_n port of the SystemBus module must be tied high in the design, or must be routed to a pin that is driven to the correct state for operation. The major signals of the MPI interface are pre-wired to specific pins that must be connected to the proper PowerQUICC bus signals. The mpi_rst_n is not a PowerQUICC signal and so must be explicitly wired to high for normal operation.

See the *"LatticeSC sysConfig Usage Guide,TN1080"* for details on configuration and done.

4.2.3 USI

The User Slave Interface (USI) is an extension of the SystemBus into the FPGA fabric and allows user IP registers in the FPGA fabric to be accessed by the driver. This interface is a slave device on the SystemBus. Accesses from the bus into the fabric can be made this way. The USI is an option when creating the SystemBus using IPExpress. Go to the "User_Slave" tab and check the "Eanble User Slave Interface" box. This option must be enabled if you want access to the design in the fabric. IPExpress will then generate a Verilog (or VHDL) file that contains the USI ports on the SystemBus which can be used in your design.

Note that there is only one USI available on the SystemBus. If you are merging two IP designs that both use the USI, then you need to create a bus mux in your logic. You can't create two USIs and tie each design to a USI.

4.2.4 UMI

The User Master Interface (UMI) is a module that extends the FPGA fabric onto the SystemBus. It allows device registers, addressable from the SystemBus, to be accessed by FPGA fabric IP. The UMI allows the drivers to access the SystemBus from a physical interface other than the MPI. One use of the UMI is to provide access to the SystemBus from the JTAG interface. Demo versions of the LatticeSC drivers use a JTAG interface as the Platform register access method and the drivers run on a PC, reading and writing registers via the JTAG interface and onto the SystemBus. Another potential use of the UMI is to allow an embedded soft CPU to drive the SystemBus as the master.

The UMI needs to be enabled in IPExpress so its ports are present when the SystemBus is instantiated. This is done on the "User Master" tab by checking the "Enable User Master Interface" box.

4.3 Device Instantiation

PLLs, DLLs, SMI ports and PCSs (Quads) need to be created with IPExpress and wired to the SystemBus in the user's design. The MPI/SystemBus App Note has examples of creating IP and attaching it to the SystemBus.

For PLLs and DLLs, the SMI port address that it will be connected to needs to be enabled in the SMI tab of the IPExpress SystemBus module. The SMI port of the PLL or DLL must be enabled in the Advanced tab, "Provide SMI ports on Module". This same address must then be included as a preference in the project .lpf file so the instantiated PLL or DLL is connected to correct SMI port. The format of the preference line is:

```
   ASIC "pll_410/pll1_0_0" TYPE "EHXPLLA" SMI_OFFSET="0x410";
```

The ASIC keyword identifies the command as corresponding to built-in hardware (PLLs and DLLs are hard IP). The "pll_410/pll1_0_0" is the hierarchical name of the instance of the PLL. pll_410 is the name of the instance of the PLL. The pll1_0_0 is a combination of the name given to the module when created by IPExpress (pll1) and an internal tool ID (_0_0 - which should always be this value). The type "EHXPLLA" is the name of the SC PLL library element. The SMI_OFFSET is obviously the SMI port to connect to.

4.4 Bitstreams and Bitgen

When using the MPI mode to download a bitstream, certain settings should be used to generate a download-compatible bitstream.

```
bitgen -q -w -g MCCLK_FREQ:133 -g WAKE_UP:24 -g WaitStateTimeout:4 -g
CONFIG_SECURE:OFF <file>.ncd <file>.bit
```

- MCCLK_FREQ – sets the SystemBus clock frequency. This is the internal clock frequency that the bus will run at. The MPI interface to the external CPU will still run at the CPU's bus frequency.
- WaitStateTimeout –this option selects the SystemBus bus timeout counter. A value of 4 gives 256 clocks before a bus error is generated. It is very important that this value be set accordingly when using SMI because of the lengthy turnaround time for SMI serial bus cycles.
- WAKE_UP – this setting depends on how your design "comes up" after configuration. Table 7-12 in the sysConfig Guide explains the values in detail. 21 or 24 are the standard modes; at the end of downloading GSR is applied, then outputs enabled then DONE goes high.

See the "*LatticeSC sysConfig Usage Guide,TN1080"* for specific details on configuration through the MPI.

5 Building and Porting

5.1 Introduction
This section gives information to enable one to setup and use the LatticeSC drivers on a new development platform.

5.2 Building
The standard way to build the code is with makefiles. Makefiles are text-based files that specify how all the source files are compiled, with what options, and in what order to create the binary executable. Makefiles use a standard language that is supported by almost every build tool and operating system. Makefiles are the most portable method of distributing the instructions for rebuilding the source code. The LatticeSC Device Driver source code will be compiled by a customer, to integrate it into their system software. Since a customer may use any number of tools (ie WindRiver Tornado or Microsoft Visual Studio) the driver code cannot be geared for one particular tool/platform.

To be truly portable, all paths and tool names are also configurable and not hard-coded. The environment variable PLATFORM_DIR is used to locate all files in the project. All includes and links references are specified relative to this variable.

Source code dependencies are automatically built using the **`–MM`** compiler option to generate a dependency file **`.depends`**. Each Makefile generates its own **`.depends`** file in its directory automatically if the file does not exist (ie. The first time the make command is run). If new files are added to a project (ie. customizing a Platform) then run **`make depends`** to update the project's dependencies.

5.2.1 Top Level Platform Make
The top level directory of the project source contains the files and rules that are used to build the entire project. Running make at this level will cause every file for every sub-project to be compiled correctly and linked into the target goal – a relocatable object that can be linked with an application to form an executable.

This brings up a few important points:

- Relocatable object format – The result of building a particular sub-project is a .obj file that is the combination of all the individual source files. Object files were chosen over shared libraries because shared libraries must be manually installed on the target system. This would require lot of extra work and revision control during initial development of the platform software. Also it makes it easier on the debugger to just link everything into one executable file. One exception is the SystemAccess/Drivers/ which may be built using their own makefile and installed into the OS.
- There are different makefiles for different levels of a project hierarchy, but they all share the same build rules, so project wide changes can be made in one place.
- The actual compiling is done using the built in rules for building a .o file from a .c file. These standard rules uses CC and CFLAGS, so setting them in rules.make and including rules.make in every makefile automatically assures that all C files will be built correctly.
- Dependencies are automatically generated from the list of source files. The resulting .depends dependency file is included into the Makefile when building.

rules.make – This file contains the definitions for the C compiler, linker and libraries that will be used by all the makefiles in the entire platform project. This file is included in every makefile. Platform wide settings are made here. Changing CFLAGS here will effect how all source files are compiled.

PLATFORM_DIR – this environment variable is set to point to the top level of a particular platform project an individual is building code for. This environment variable is used by all makefiles to locate the rules.make file so they know which compiler and flags to use for building. It also is used to locate directories in search paths.

Makefile – this makefile contains the list of modules that are combined together to form the final goal. The final goal is to create a relocatable object file *(`<project_name>.obj`)* that can be linked with application code to create an executable. The list of modules to build is given in this makefile. They are the standard modules that together define a complete platform: SystemAccess, Platform and DeviceLib. The main purpose of this makefile is to invoke these sub-makefiles.

5.2.2 Module Level

A module is located below the top-level platform directory, and consists of project(s) that are built and combined together to form the functional module. The makefile located in the Platform/ directory is a good example of a module makefile. This makefile specifies the projects that need to be built and linked together. In this case the sub-projects Devices, Services and Utils are all built using their respective makefiles and the resulting 3 object files are combined into the Devices.obj file that is needed by the top-level makefile.

5.2.3 Sub-Project Level

The sub-project level is the lowest level of the hierarchy. It is here that a collection of source files are compiled together to form the relocatable object. The source files for a particular project are listed in the variable SRC. This listing must be kept up-to-date when new files are added to (or removed from) the project. Specific rules can be added to CFLAGS or INCLUDE_PATH to customize how the files in this project are built. They will be built using the settings from rules.make.

Note that the Device Library makefile is directly invoked from the Module level makefile and is not actually a part of the Platform makefile structure. The DevLib makefile settings are overridden on the command line when it is invoked so that CC is the CC from rules.make and other settings can be changed. This is done without changing the makefile itself, merely overriding its default definitions. This allows the Device Library makefile to be a standalone, platform independent, file.

5.2.4 To Make an Executable

The Makefile that builds an executable file that can be run on the target CPU is found in the Applications/ directory. Again, this makefile gets all its rules and settings from the top-level platform rules.make file. This makefile is only different in one special way. It is designed to build a single C source file, containing the main() function, into an object file, link it with the platform object library (*`platform_name.obj`*) and create a single executable file. Therefore it does not list the source files. It takes the executable target name, specified on the command line, and builds the corresponding .c file into an executable.

```
$ make serdes1
```

Would build `serdes1.c` into an object file, and if successful, link it with *`platform_name.obj`* and the operating system libraries to resolve all symbols and produce the executable file `serdes1`.

The makefiles found in the Applications are a good starting point for any new application code.

6 Reference Information

The following documents provide more information on topics discussed throughout this guide:

- *“LatticeSC Data Sheet”*
- “*LatticeSC sysCONFIG Usage Guide,TN1080”*
- “*LatticeSC MPI/System Bus Application Note,TN1085”*
- *“LatticeSC Communications Platform Evaluation Board”*

7 Technical Support Assistance

Hotline: 1-800-LATTICE (North America)
+1-408-826-6002 (Outside North America)

e-mail: techsupport@latticesemi.com

Internet: www.latticesemi.com