XSTOOLs Source Documentation 4.0

Architecture of the XSTOOLs software

RELEASE DATE 1/2/2002

Table of Contents

Introduction

This document describes the organization and functions of the source modules that are compiled to create the following XSTOOLs utilities for the XS40, XS95, XSA and XSV Boards:

XSLOAD is a command-line program that communicates with an XS Board through the parallel port so as to configure the programmable device, download/upload the RAM, or download/upload the Flash memory.

XSPORT is a much simpler program that just outputs a byte of binary bits onto the eight data pins of the parallel port. It is used to force logic levels onto the input pins of the programmable device so as to test a downloaded logic circuit.

XSSETCLK is used to set the programmable oscillator frequency on an XS Board via the parallel port.

XSTEST is used to test the functionality of an XS Board via the parallel port.

The XSTOOLs utilities make use of the objects and function libraries shown in **Figure 1**. Each type of XS Board has a corresponding object whose methods implement the functions for configuring the programmable devices, uploading/downloading the RAM and Flash, setting the oscillator frequency and testing the board functionality. In turn, a given board object is a collection of lower-level objects that each implement an interface through the parallel port to a particular device on the corresponding XS Board such as the FPGA, CPLD, RAM, Flash memory or programmable oscillator. Finally, there are some miscellaneous objects that handle JTAG bitstreams, error and progress reporting, handling of hexadecimal data records and various other utility subroutines. A short synopsis of each object is given below. More detailed explanations are provided in the next section of this document.

XSBoard: This is a virtual object that contains pointers to the following base methods provided by all the XS Board objects: configuring the main programmable device, configuring the programmable device that manages the parallel port interface on the board (if present), uploading and downloading the RAM, uploading and downloading the Flash memory (if present), setting the frequency of the programmable oscillator and testing the functions of the board.

XS40Board: This is a subclass that specializes the XSBoard object to handle the XS40 Board. This object provides methods to send configuration bitstreams to the XC4000 FPGA, upload and download the RAM, download FPGA configuration bitstreams to the AT17CXXX serial EEPROM, set the programmable oscillator frequency and test the board functions.

XS95Board: This is a subclass that specializes the XSBoard object to handle the XS95 Board. This object provides methods to download configuration bitstreams to the XC95108 CPLD, upload and download the RAM, set the programmable oscillator frequency and test the board functions.

XSABoard: This is a subclass that specializes the XSBoard object to handle the XSA Board. This object provides methods to send configuration bitstreams to the XC2S SpartanII FPGA and the XC9572 CPLD that manages its parallel port interface, upload and download the RAM, upload and download the AT49F002 Flash memory, set the programmable oscillator frequency and test the board functions.

XSVBoard: This is a subclass that specializes the XSBoard object to handle the XSV Board. This object provides methods to send configuration bitstreams to the XCV Virtex FPGA and the XC95108 CPLD that manages its parallel port interface, upload and download the RAM, upload and download the 28F016 Flash memory, set the programmable oscillator frequency and test the board functions.

PPort: This object provides a low-level interface to the PC parallel port.

CnfgPort: This object specializes the PPort object to provide a set of methods for controlling the configuration pins of an FPGA.

XCVPort: This object specializes the CnfgPort object to provide methods for downloading configuration bitstreams into a Virtex FPGA.

XC4KPort: This object specializes the CnfgPort object to provide methods for downloading configuration bitstreams into an XC4000 FPGA.

XC2SPort: This object specializes the CnfgPort object to provide methods for downloading configuration bitstreams into an XC2S SpartanII FPGA.

RAMPort: This object specializes the PPort object to allow uploads and downloads of hexadecimal records to the RAM on an XS40, XSA or XSV Board.

AT17Port: This object specializes the PPort object to allow the downloading of configuration bitstreams into an AT17CXXX serial EEPROM on the XS40 Board.

FlashPort: This object specializes the PPort object to allow uploads and downloads of hexadecimal records to the Flash memory on an XSA or XSV Board.

F28Port: This object specializes the FlashPort object to allow uploads and downloads of hexadecimal records to the 28F016 Flash memory on an XSV Board.

AT49FPort: This object specializes the FlashPort object to allow uploads and downloads of hexadecimal records to the AT49F002 Flash memory on an XSA Board.

OscPort: This object specializes the PPort object to allow the setting of the divisor on the programmable oscillator, thus determining its output frequency.

JTAGPort: This object specializes the PPort object to add JTAG capabilities to the parallel port.

XC95KPort: This object specializes the JTAGPort object to allow downloads of SVF configuration files to an XC9500 CPLD and readback of the devide ID and USERCODE registers of the CPLD.

JTAGRAMPort: This object specializes the JTAGPort object to provide upload and download paths to the RAM on the XS95 Board through the JTAG circuitry of the XC9500 CPLD.

Bitstream: This object stores arbitrary-length strings of binary bits and provides some methods for performing operations upon them.

XCBSDR: This object specializes a Bitstream object to allow it to interface to the boundary scan data register found in hardware that supports the JTAG interface. It provides methods that make it easy to access the XS95 Board RAM through the JTAG interface.

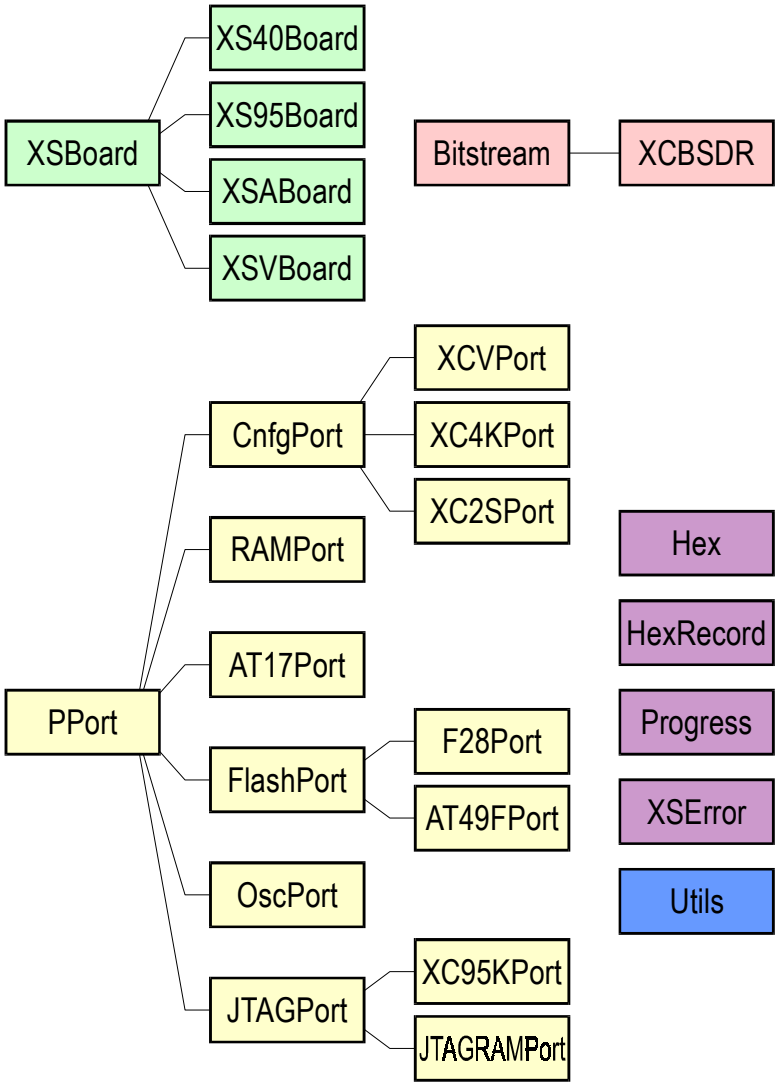
Hex: This object is used for storing hexadecimal data with an arbitrary number of digits.

HexRecord: This object extracts hexadecimal records from hex files in Intel, Motorola and XESS formats. HexRecords make use of Hex objects, but they do not inherit anything from the Hex object.

Progress: This object reports the progress of an operation using a progress bar (when using a GUI) or an updated text string (in a command-line interface).

XSError: This object specializes the ostream_with_assign object (cout and cerr are examples of this kind of object) to provide a channel for reporting errors with a consistent format.

Utils: This is a set of subroutines that are used by other subroutines and methods.



• **Figure 1:** Grouping and hierarchies for the XSTOOLs object modules.

Object Descriptions

XSBoard

This is a virtual class from which the object classes for the XS40, XS95, XSA and XSV Board classes are derived. This class provides the following virtual methods that all the derived classes must support:

`virtual bool Setup(XSError* err, const char* brdModel, unsigned int lptNum)` : This method initializes the various objects contained in the XSBoard object. All communication with the XS Board will occur through the parallel port with the index passed through lptNum. Any errors that occur during operations with the XS Board will be reported through the given error-reporting channel. The brdModel parameter indicates the particular model of XS Board associated with the object. Returns true if no errors occur.

`virtual bool Configure(string& fileName)`: This method downloads the configuration bitstream in the given file into the main programmable device on the XS Board. Returns true if no errors occur.

`virtual bool ConfigureInterface(const char* brdModel, string& fileName)`: This method downloads the configuration bitstream in the given file into the programmable device that manages the parallel port interface on the XS Board. Returns true if no errors occur.

`virtual bool DownloadRAM(string& fileName, bool doStart, bool doEnd)`: This method downloads a HEX data file in Intel, Motorola or XESS format into the RAM on the XS Board. Because multiple HEX files may be downloaded, boolean parameters are provided to indicate if the given file is the first or last in the sequence so that initialization and shutdown procedures can be activated accordingly. Returns true if no errors occur.

`virtual bool UploadRAM(string& fileName, const char* format, unsigned int loAddr, unsigned int hiAddr)`: This method uploads data between the upper and lower addresses (inclusive) from the XS Board RAM and stores it in a HEX file in Intel, Motorola or XESS format. Returns true if no errors occur.

`virtual bool DownloadFlash(string& fileName, bool doStart, bool doEnd)`: This method downloads a HEX data file in Intel, Motorola or XESS format into the Flash memory on the XS Board. Because multiple HEX files may be downloaded, boolean parameters are provided to indicate if the given file is the first or last in the sequence so that initialization and shutdown procedures can be activated accordingly. Returns true if no errors occur.

`virtual bool UploadFlash(string& fileName, const char* format, unsigned int loAddr, unsigned int hiAddr)`: This method uploads data between the upper and lower addresses (inclusive) from the XS Board Flash and stores it in a HEX file in Intel, Motorola or XESS format. Returns true if no errors occur.

`virtual bool SetFreq(int div, bool extOscPresent)`: This method sets the divisor of the programmable oscillator on the XS Board. The internal master oscillator is divided by this parameter to arrive at the clock frequency that is output to the rest of the XS Board

circuitry. An external clock source is used as the master oscillator if the extOscPresent parameter is true. Returns true if no errors occur.

`bool Test(void)`: This method executes a diagnostic test routine on the XS Board and returns true if the test passes and false if the test detects a problem.

XS40Board

The details of how the XS40 Board class methods implement the XS Board class virtual methods are given below.

`XS40Board(void)`: This constructor creates a default error channel, reads in the data file describing all the XS Board models and then calls the `Setup` method to initialize all the components (FPGA, RAM, serial EEPROM and programmable oscillator) of the XS40 Board.

`bool Setup(XSError* err, const char* brdModel, unsigned int lptNum)`: This method initializes the XC4000 FPGA, RAM, AT17CXXX serial EEPROM and programmable oscillator objects contained in the XS40Board object. All communication with these components will occur through the parallel port having the index passed through lptNum. Any errors that occur during operations with the XS Board will be reported through the given error-reporting channel. Returns true if no errors occur while initializing the components.

`bool Configure(string& fileName)`: This method starts by finding the record in the XS Board information data structure that corresponds with the particular XS40 Board model being used. Then it checks the input file to see if it is a .BIT file containing a configuration bitstream. If so, it extracts the target device from the bitstream file and compares it to the type of FPGA device on the XS40 Board. If they match, then the FPGA is initialized for the configuration process and the file is passed to the method that downloads bitstreams to the FPGA. This method returns false if any of the conditional checks fail. Otherwise it returns the status from the method that downloads the bitstream.

`bool ConfigureInterface(const char* brdModel,string& fileName)`: The XS40 Board has no auxiliary programmable device handling the parallel port interface (the XC4000 FPGA is connected directly to the port), so this method always returns false when it is called.

`bool DownloadRAM(string& fileName, bool doStart, bool doEnd)`: This method starts by checking the suffix of the file to be downloaded into RAM to see if it is an Intel, Motorola or XESS HEX file. Then it finds the record in the XS Board information data structure that corresponds with the particular XS40 Board model being used. The file name of the FPGA configuration bitstream that implements an interface between the RAM and the parallel port is extracted from the board information record. If no RAM interface bitstream is found then the method terminates, but this should never happen. If this is the first data file to be downloaded to the RAM (`doStart` is true), then the RAM interface bitstream is downloaded to the FPGA. Once the FPGA is configured with the RAM interface, then the data file can be downloaded. Any subsequent calls with doStart set to false will skip the configuration of the FPGA with the RAM interface bitstream and proceed directly to downloading the RAM data.

`bool UploadRAM(string& fileName, const char* format, unsigned int loAddr, unsigned int hiAddr)`: This method starts by finding the record in the XS

Board information data structure that corresponds with the particular XS40 Board model being used. The file name of the FPGA configuration bitstream that implements an interface between the RAM and the parallel port is extracted from the board information record. If no RAM interface bitstream is found then the method terminates, but this should never happen. Next, the RAM interface bitstream is downloaded to the FPGA and the data between the lower and upper addresses (inclusive) is uploaded from the RAM and stored in a file with the given name using the given file format.

`bool DownloadFlash(string& fileName, bool doStart, bool doEnd)`: This method starts by checking the suffix of the file to be downloaded into Flash to see if it is a Xilinx bitstream file. Then it finds the record in the XS Board information data structure that corresponds with the particular XS40 Board model being used. The file name of the FPGA configuration bitstream that implements an interface between the serial EEPROM and the parallel port is extracted from the board information record. If no interface bitstream is found then the method terminates, but this should never happen. If this is the first bitstream file to be downloaded to the serial EEPROM (`doStart` is true), then the user is given some instructions to prepare the XS40 Board for programming the serial EEPROM and the interface bitstream is downloaded to the FPGA. Once the FPGA is configured with the serial EEPROM interface, then the bitstream file is downloaded. Any subsequent calls with doStart set to false will skip the configuration of the FPGA with the RAM interface bitstream and proceed directly to downloading the bitstream data. Once the last bitstream file is downloaded (`doEnd` is true), the user is given some instructions to prepare the XS40 Board so it will configure itself from the serial EEPROM when power is next applied.

`bool UploadFlash(string& fileName, const char* format, unsigned int loAddr, unsigned int hiAddr)`: The XS40 Board does not support reading of the contents of the serial EEPROM back through the parallel port to the PC, so this method always returns false.

`bool SetFreq(int div, bool extOscPresent)`: This method starts by finding the record in the XS Board information data structure that corresponds with the particular XS40 Board model being used. The file name of the FPGA configuration bitstream that implements an interface between the programmable oscillator and the parallel port is extracted from the board information record. If no interface bitstream is found then the method terminates, but this should never happen. Then the user is given some instructions to prepare the XS40 Board for programming the oscillator and the interface bitstream is downloaded to the FPGA. Once the FPGA is configured with the interface, the oscillator is programmed with the master frequency divisor and the source of the master frequency is selected. After the oscillator is programmed, the user is given some instructions to prepare the XS40 Board so the new frequency will appear when power is next applied to the board.

`bool Test(void)`: This method starts by finding the record in the XS Board information data structure that corresponds with the particular XS40 Board model being used. The file name of the FPGA configuration bitstream that supports diagnostic testing is extracted from the board information record. If no interface bitstream is found then the method terminates, but this should never happen. Then the file name for a HEX file containing the diagnostic program for the microcontroller on the XS40 Board is extracted from the board information data structure. This program file is downloaded into the RAM and then the test interface bitstream is loaded into the FPGA. Then the diagnostic program will execute on the XS40 Board and send a “heartbeat” signal back to the PC through the parallel port. This method will time the heartbeat to determine its initial frequency. If the

diagnostic routine on the XS40 detects a problem it will terminate the heartbeat. When this method detects the absence of the heartbeat, it will report a problem with the board and list several possible causes for the problem. But if the diagnostic executes successfully, then the heartbeat is sped up. When this method detects the faster heartbeat, it will report that the diagnostic test was passed and the XS40 Board is OK.

XS95Board

The details of how the XS95 Board class methods implement the XS Board class virtual methods are given below.

`XS95Board(void)`: This constructor creates a default error channel, reads in the data file describing all the XS Board models and then calls the `Setup` method to initialize all the components (CPLD, RAM and programmable oscillator) of the XS95 Board.

`bool Setup(XSError* err, const char* brdModel, unsigned int lptNum)`: This method initializes the XC9500 CPLD, RAM and programmable oscillator objects contained in the XS95Board object. All communication with these components will occur through the parallel port having the index passed through lptNum. Any errors that occur during operations with the XS Board will be reported through the given error-reporting channel. Returns true if no errors occur while initializing the components.

`bool CheckChipID(void)`: This method reads the device identifier for the CPLD and compares it to the device ID for the XC95108 CPLD which should reside on the XS95 Board. It returns true if they match. If they do not match, the user is given the option to force the result to be true so that further processing can continue. Otherwise this method returns false.

`bool Configure(string& fileName)`: This method starts by checking the input file to see if it is a .SVF file containing a configuration bitstream for an XC9500 CPLD. If so, it checks the identifier of the CPLD device on the board to see if it is an XC95108. If not, the method terminates and returns false. Otherwise, the CPLD is initialized for configuration and the .SVF file is downloaded to it. The status from the method that downloads the bitstream is returned as the result.

`bool ConfigureInterface(const char* brdModel,string& fileName)`: The XS95 Board has no auxiliary programmable device handling the parallel port interface (the XC95108 CPLD is connected directly to the port), so this method always returns false when it is called.

`bool DownloadRAM(string& fileName, bool doStart, bool doEnd)`: This method starts by checking the suffix of the file to be downloaded into RAM to see if it is an Intel, Motorola or XESS HEX file. If so, it checks the identifier of the CPLD device on the board to see if it is an XC95108. If not, the method terminates and returns false. If it is an XC95108 then the HEX data file can be downloaded immediately using the JTAG interface which is always available and does not require any reconfiguration of the CPLD.

`bool UploadRAM(string& fileName, const char* format, unsigned int loAddr, unsigned int hiAddr)`: This method starts by checking the identifier of the CPLD device to see if it is an XC95108. If not, the method terminates and returns false. If it is an XC95108 then the data between the upper and lower addresses (inclusive) can be uploaded immediately using the JTAG interface which is always available and does not require any reconfiguration of the CPLD.

`bool DownloadFlash(string& fileName, bool doStart, bool doEnd)`: The XS95 Board does not have an external Flash memory, so this method always returns false.

`bool UploadFlash(string& fileName, const char* format, unsigned int loAddr, unsigned int hiAddr)`: The XS95 Board does not have an external Flash memory, so this method always returns false.

`bool SetFreq(int div, bool extOscPresent)`: This method starts by finding the record in the XS Board information data structure that corresponds with the particular XS95 Board model being used. The file name of the CPLD configuration bitstream that implements an interface between the programmable oscillator and the parallel port is extracted from the board information record. If no interface bitstream is found then the method terminates, but this should never happen. Then the user is given some instructions to prepare the XS95 Board for programming the oscillator and the interface bitstream is downloaded to the CPLD. Once the CPLD is configured with the interface, the oscillator is programmed with the master frequency divisor and the source of the master frequency is selected. After the oscillator is programmed, the interface file is erased from the CPLD. Then the user is given some instructions to prepare the XS95 Board so the new frequency will appear when power is next applied to the board.

`bool Test(void)`: This method starts by finding the record in the XS Board information data structure that corresponds with the particular XS95 Board model being used. The file name of the CPLD configuration bitstream that supports diagnostic testing is extracted from the board information record. If no interface bitstream is found then the method terminates, but this should never happen. Then the file name for a HEX file containing the diagnostic program for the microcontroller on the XS95 Board is extracted from the board information data structure. This program file is downloaded into the RAM and then the test interface bitstream is loaded into the CPLD. Then the diagnostic program will execute on the XS95 Board and send a "heartbeat" signal back to the PC through the parallel port. This method will time the heartbeat to determine its initial frequency. If the diagnostic routine on the XS95 detects a problem it will terminate the heartbeat. When this method detects the absence of the heartbeat, it will report a problem with the board and list several possible causes for the problem. But if the diagnostic executes successfully, then the heartbeat is sped up. When this method detects the faster heartbeat, it will report that the diagnostic test was passed and the XS95 Board is OK.

XSABoard

The details of how the XSA Board class methods implement the XS Board class virtual methods are given below.

`XSABoard(void)`: This constructor creates a default error channel, reads in the data file describing all the XS Board models and then calls the `Setup` method to initialize all the components (FPGA, CPLD, SDRAM, Flash memory and programmable oscillator) of the XSA Board.

`bool Setup(XSError* err, const char* brdModel, unsigned int lptNum)`: This method initializes the XC2S SpartanII FPGA, XC9572XL CPLD, SDRAM, AT49F002 Flash memory and programmable oscillator objects contained in the XSABoard object. All communication with these components will occur through the parallel port having the index passed through lptNum. Any errors that occur during operations with the XS Board will be reported through the given error-reporting channel. Returns true if no errors occur while initializing the components.

`bool CheckChipID(void)`: This method reads the device identifier for the CPLD and compares it to the device ID for the XC9572XL CPLD which should reside on the XSA Board to handle the parallel port interface. It returns true if they match. If they do not match, the user is given the option to force the result to be true so that further processing can continue. Otherwise this method returns false.

`bool Configure(string& fileName)`: This method starts by checking the identifier for the XC9572XL CPLD that manages the parallel port interface for the XSA Board. This method terminates if the correct CPLD identifier is not found as this usually indicates a problem with the XSA Board. Then this method checks the input file to see if it is a .BIT or .SVF file containing a configuration bitstream. .SVF files are passed to the ConfigureInterface method which reprograms the interface CPLD. With a .BIT file, the method finds the record in the XS Board information data structure that corresponds with the particular XSA Board model being used. It extracts the target device from the bitstream file and compares it to the type of FPGA device on the XSA Board. If they do not match the method terminates and returns false. Otherwise, the method gets the contents of the USERCODE signature register from the interface CPLD. The value in this register indicates the function that the CPLD is programmed to perform. The CPLD will be reprogrammed if it contains a circuit that does not allow downloading of a configuration bitstream to the FPGA. Otherwise the CPLD will be left alone because it does allow this function. Then the FPGA is initialized and the configuration bitstream in the file is downloaded to the FPGA.

`bool ConfigureInterface(const char* brdModel,string& fileName)`: This method checks the input file to see if it is an .SVF file containing a CPLD configuration bitstream. Then the chip indentifier of the interface CPLD on the XSA Board is fetched to make sure it is an XC9572XL. Then the bitstream in the .SVF file is programmed into the CPLD.

`bool DownloadRAM(string& fileName, bool doStart, bool doEnd)`: This method starts by checking the suffix of the file to be downloaded into RAM to see if it is an Intel, Motorola or XESS HEX file. Then it finds the record in the XS Board information data structure that corresponds with the particular XSA Board model being used. The file name of the FPGA configuration bitstream that implements an interface between the RAM and the parallel port is extracted from the board information record. If no RAM interface bitstream is found then the method terminates, but this should never happen. If this is the first data file to be downloaded to the RAM (`doStart` is true), then the RAM interface bitstream is downloaded to the FPGA. Once the FPGA is configured with the RAM interface, then the data file can be downloaded. Any subsequent calls with `doStart` set to false will skip the configuration of the FPGA with the RAM interface bitstream and proceed directly to downloading the RAM data.

`bool UploadRAM(string& fileName, const char* format, unsigned int loAddr, unsigned int hiAddr)`: This method starts by finding the record in the XS Board information data structure that corresponds with the particular XSA Board model being used. The file name of the FPGA configuration bitstream that implements an interface between the RAM and the parallel port is extracted from the board information record. If no RAM interface bitstream is found then the method terminates, but this should never happen. Next, the RAM interface bitstream is downloaded to the FPGA and the data between the lower and upper addresses (inclusive) is uploaded from the RAM and stored in a file with the given name using the given file format.

`bool DownloadFlash(string& fileName, bool doStart, bool doEnd)`: This method starts by checking the suffix of the file to be downloaded into Flash to see if it is an Intel, Motorola or XESS HEX file. Then it finds the record in the XS Board information data structure that corresponds with the particular XSA Board model being used. The file name of the CPLD configuration bitstream that implements an interface between the Flash memory and the parallel port is extracted from the board information record. If no interface bitstream is found then the method terminates, but this should never happen. Next the chip identifier of the interface CPLD is checked to make sure it is an XC9572XL. Then the USERCODE register is queried to see if the CPLD already contains the circuitry necessary to interface the Flash memory to the parallel port. If not, then the CPLD is reprogrammed with the required circuitry. Then the HEX data file is downloaded. One or more data files may be loaded into the Flash memory. After the last data file is programmed into the Flash memory (`doEnd` is true), then the CPLD is reprogrammed with a circuit that will let it configure the FPGA with the contents of the Flash upon power-up.

`bool UploadFlash(string& fileName, const char* format, unsigned int loAddr, unsigned int hiAddr)`: This method starts by finding the record in the XS Board information data structure that corresponds with the particular XSA Board model being used. The file name of the CPLD configuration bitstream that implements an interface between the Flash memory and the parallel port is extracted from the board information record. If no interface bitstream is found then the method terminates, but this should never happen. Next the chip identifier of the interface CPLD is checked to make sure it is an XC9572XL. Then the USERCODE register is queried to see if the CPLD already contains the circuitry necessary to interface the Flash memory to the parallel port. If not, then the CPLD is reprogrammed with the required circuitry. Then the data between the lower and upper addresses (inclusive) in the Flash memory are uploaded through the parallel port and stored in the given file in the given format. Finally, the CPLD is reprogrammed with the circuit that will let it configure the FPGA with the contents of the Flash upon power-up.

`bool SetFreq(int div, bool extOscPresent)`: The programmable oscillator on the XSA Board has a direct connection to a pin on the parallel port so there is no need to reprogram the CPLD or FPGA to create an interface. So this method starts by giving the user some instructions to prepare the XSA Board for programming the oscillator. After the user completes these steps, the oscillator is programmed with the master frequency divisor and the source of the master frequency is selected. Then the user is given some instructions to prepare the XSA Board so the new frequency will appear when power is next applied to the board.

`bool Test(void)`: This method starts by finding the record in the XS Board information data structure that corresponds with the particular XSA Board model being used. The file name of the FPGA configuration bitstream that supports diagnostic testing is extracted from the board information record. If no interface bitstream is found then the method terminates, but this should never happen. Then the chip identifier for the interface CPLD is checked to make sure it is an XC9572XL. If not, this usually indicates a problem with the XSA Board so the test terminates and return false. Then the value in the USERCODE signature register is queried to see if the interface CPLD circuit supports the diagnostic test code. The CPLD is reprogrammed with the appropriate interface circuit for the diagnostic if necessary. Then the FPGA is programmed with its own diagnostic circuit and the test begins. The diagnostic program will execute on the XSA Board and send a "heartbeat" signal back to the PC through the parallel port. This method will time the heartbeat to determine its initial frequency. If the diagnostic routine on the XSA detects a

problem it will terminate the heartbeat. When this method detects the absence of the heartbeat, it will report a problem with the board and list several possible causes for the problem. But if the diagnostic executes successfully, then the heartbeat is sped up. When this method detects the faster heartbeat, it will report that the diagnostic test was passed and the XSA Board is OK.

XSVBoard

The details of how the XSV Board class methods implement the XS Board class virtual methods are given below.

`XSVBoard(void)`: This constructor creates a default error channel, reads in the data file describing all the XS Board models and then calls the `Setup` method to initialize all the components (FPGA, CPLD, RAM, Flash memory and programmable oscillator) of the XSV Board.

`bool Setup(XSError* err, const char* brdModel, unsigned int lptNum)`: This method initializes the XCV Virtex FPGA, XC95108 CPLD, RAM, 28F016 Flash memory and programmable oscillator objects contained in the XSVBoard object. All communication with these components will occur through the parallel port having the index passed through lptNum. Any errors that occur during operations with the XS Board will be reported through the given error-reporting channel. Returns true if no errors occur while initializing the components.

`bool CheckChipID(void)`: This method reads the device identifier for the CPLD and compares it to the device ID for the XC95108 CPLD which should reside on the XSV Board to handle the parallel port interface. It returns true if they match. If they do not match, the user is given the option to force the result to be true so that further processing can continue. Otherwise this method returns false.

`bool Configure(string& fileName)`: This method starts by checking the identifier for the XC95108 CPLD that manages the parallel port interface for the XSV Board. This method terminates if the correct CPLD identifier is not found as this usually indicates a problem with the XSV Board. Then this method checks the input file to see if it is a .BIT or .SVF file containing a configuration bitstream. .SVF files are passed to the ConfigureInterface method which reprograms the interface CPLD. With a .BIT file, the method finds the record in the XS Board information data structure that corresponds with the particular XSV Board model being used. It extracts the target device from the bitstream file and compares it to the type of FPGA device on the XSV Board. If they do not match the method terminates and returns false. Otherwise, the method gets the contents of the USERCODE signature register from the interface CPLD. The value in this register indicates the function that the CPLD is programmed to perform. The CPLD will be reprogrammed if it contains a circuit that does not allow downloading of a configuration bitstream to the FPGA. Otherwise the CPLD will be left alone because it does allow this function. Then the FPGA is initialized and the configuration bitstream in the file is downloaded to the FPGA.

`bool ConfigureInterface(const char* brdModel,string& fileName)`: This method checks the input file to see if it is an .SVF file containing a CPLD configuration bitstream. Then the chip indentifier of the interface CPLD on the XSA Board is fetched to make sure it is an XC95108. Then the bitstream in the .SVF file is programmed into the CPLD.

`bool DownloadRAM(string& fileName, bool doStart, bool doEnd)`: This method starts by checking the suffix of the file to be downloaded into RAM to see if it is an Intel, Motorola or XESS HEX file. Then it finds the record in the XS Board information data structure that corresponds with the particular XSV Board model being used. The file name of the FPGA configuration bitstream that implements an interface between the RAM and the parallel port is extracted from the board information record. If no RAM interface bitstream is found then the method terminates, but this should never happen. If this is the first data file to be downloaded to the RAM (`doStart` is true), then the RAM interface bitstream is downloaded to the FPGA. Once the FPGA is configured with the RAM interface, then the data file can be downloaded. Any subsequent calls with `doStart` set to false will skip the configuration of the FPGA with the RAM interface bitstream and proceed directly to downloading the RAM data.

`bool UploadRAM(string& fileName, const char* format, unsigned int loAddr, unsigned int hiAddr)`: This method starts by finding the record in the XS Board information data structure that corresponds with the particular XSV Board model being used. The file name of the FPGA configuration bitstream that implements an interface between the RAM and the parallel port is extracted from the board information record. If no RAM interface bitstream is found then the method terminates, but this should never happen. Next, the RAM interface bitstream is downloaded to the FPGA and the data between the lower and upper addresses (inclusive) is uploaded from the RAM and stored in a file with the given name using the given file format.

`bool DownloadFlash(string& fileName, bool doStart, bool doEnd)`: This method starts by checking the suffix of the file to be downloaded into Flash to see if it is an Intel, Motorola or XESS HEX file. Then it finds the record in the XS Board information data structure that corresponds with the particular XSV Board model being used. The file name of the CPLD configuration bitstream that implements an interface between the Flash memory and the parallel port is extracted from the board information record. If no interface bitstream is found then the method terminates, but this should never happen. Next the chip identifier of the interface CPLD is checked to make sure it is an XC95108.. Then the USERCODE register is queried to see if the CPLD already contains the circuitry necessary to interface the Flash memory to the parallel port. If not, then the CPLD is reprogrammed with the required circuitry. Then the HEX data file is downloaded. One or more data files may be loaded into the Flash memory. After the last data file is programmed into the Flash memory (`doEnd` is true), then the CPLD is reprogrammed with a circuit that will let it configure the FPGA with the contents of the Flash upon power-up.

`bool UploadFlash(string& fileName, const char* format, unsigned int loAddr, unsigned int hiAddr)`: This method starts by finding the record in the XS Board information data structure that corresponds with the particular XSV Board model being used. The file name of the CPLD configuration bitstream that implements an interface between the Flash memory and the parallel port is extracted from the board information record. If no interface bitstream is found then the method terminates, but this should never happen. Next the chip identifier of the interface CPLD is checked to make sure it is an XC95108. Then the USERCODE register is queried to see if the CPLD already contains the circuitry necessary to interface the Flash memory to the parallel port. If not, then the CPLD is reprogrammed with the required circuitry. Then the data between the lower and upper addresses (inclusive) in the Flash memory are uploaded through the parallel port and stored in the given file in the given format. Finally, the CPLD is reprogrammed with the circuit that will let it configure the FPGA with the contents of the Flash upon power-up.

`bool SetFreq(int div, bool extOscPresent)`: The programmable oscillator on the XSA Board has a direct connection to a pin on the parallel port so there is no need to reprogram the CPLD or FPGA to create an interface. So this method starts by giving the user some instructions to prepare the XSA Board for programming the oscillator. After the user completes these steps, the oscillator is programmed with the master frequency divisor and the source of the master frequency is selected. Then the user is given some instructions to prepare the XSA Board so the new frequency will appear when power is next applied to the board. This method starts by finding the record in the XS Board information data structure that corresponds with the particular XSV Board model being used. The file name of the FPGA configuration bitstream that implements an interface between the programmable oscillator and the parallel port is extracted from the board information record. If no interface bitstream is found then the method terminates, but this should never happen. Next the chip identifier of the interface CPLD is checked to make sure it is an XC95108. . Then the value in the USERCODE signature register is queried to see if the interface CPLD circuit supports programming of the oscillator. A flag is set to indicate whether or not the CPLD should be reprogrammed with the appropriate interface circuit to access the programmable oscillator. Then the user is given some instructions to prepare the XSV Board for programming the oscillator and the interface bitstream is downloaded to the CPLD. Once the CPLD is configured with the interface, the oscillator is programmed with the master frequency divisor and the source of the master frequency is selected. After the oscillator is programmed, the circuit in the interface CPLD is erased. Then the user is given some instructions to prepare the XSV Board so the new frequency will appear when power is next applied to the board.

`bool Test(void)`: This method starts by finding the record in the XS Board information data structure that corresponds with the particular XSV Board model being used. The file name of the FPGA configuration bitstream that supports diagnostic testing is extracted from the board information record. If no interface bitstream is found then the method terminates, but this should never happen. Then the chip identifier for the interface CPLD is checked to make sure it is an XC95108. If not, this usually indicates a problem with the XSV Board so the test terminates and return false. Then the value in the USERCODE signature register is queried to see if the interface CPLD circuit supports the diagnostic test code. The CPLD is reprogrammed with the appropriate interface circuit for the diagnostic if necessary. Then the FPGA is programmed with its own diagnostic circuit and the test begins. The diagnostic program will execute on the XSV Board and send a "heartbeat" signal back to the PC through the parallel port. This method will time the heartbeat to determine its initial frequency. If the diagnostic routine on the XSA detects a problem it will terminate the heartbeat. When this method detects the absence of the heartbeat, it will report a problem with the board and list several possible causes for the problem. But if the diagnostic executes successfully, then the heartbeat is sped up. When this method detects the faster heartbeat, it will report that the diagnostic test was passed and the XSV Board is OK.

PPort

This object handles the parallel port. It lets you set the value of bit fields in the data and control registers of the parallel port, and it lets you get values from bit fields of the status register. These three byte-wide registers are concatenated into a single 24-bit register from which bit fields are extracted using upper and lower indices in the range [0,23]. The PPort object also lets you specify a 24-bit inversion mask to counter the effect of any inverters found in the PC parallel port and/or XS Board hardware.

The PPort object uses lower-level I/O routines from either the UNIIO or DLPORTIO DLL libraries and drivers. The UNIIO interface is used by default, but the user can select the DLPORTIO routines by setting the appropriate value in the XSTOOLS parameter file.

The PPort object can be initialized with a parallel port number in the range [1,4] in which case it will fetch the actual hardware address for the parallel port registers from the PC RAM. This address can be overridden by specifying the hardware address in the XSTOOLS parameter file.

The PPort object will check its operation by verifying that any levels on the parallel port pins match the values it has placed on them. This alerts the PPort object to any problems accessing the parallel port hardware. These checks cease after a set number of I/O operations in order to increase the port throughput.

CnfgPort

This object provides methods for setting and querying the pins of the parallel port connected to the CCLK, DIN, PROG and DONE pins of the FPGA on an XS Board. The association between each FPGA configuration pin and its bit position in the 24-bit parallel port register is made when the CnfgPort object is instantiated.

XCVPort

This object provides methods for downloading a configuration bitstream into a Virtex FPGA. The bitstream file is opened and the field containing the configuration bits is extracted and passed byte-by-byte into the FPGA. The download operates in one of two modes: fast mode where each byte is downloaded through the parallel port and into the Virtex FPGA in SelectMap configuration mode, or slow mode where each byte is serially transmitted through a single pin of the parallel port into the FPGA in slave-serial configuration mode.

A method is also provided to extract the target device from a bitstream file. This is used to verify that a bitstream file is being downloaded to the proper type of FPGA.

XC4KPort

This object provides methods for downloading a configuration bitstream into an XC4000 FPGA. The bitstream file is opened and the field containing the configuration bits is extracted and passed byte-by-byte into the FPGA. Each byte is serially transmitted through a single pin of the parallel port into the FPGA in slave-serial configuration mode.

A method is also provided to extract the target device from a bitstream file. This is used to verify that a bitstream file is being downloaded to the proper type of FPGA.

XC2SPort

This object provides methods for downloading a configuration bitstream into a SpartanII FPGA. The bitstream file is opened and the field containing the configuration bits is extracted and passed byte-by-byte into the FPGA. The download operates in one of two modes: fast mode where each byte is downloaded through the parallel port and into the SpartanII FPGA in SelectMap configuration mode, or slow mode where each byte is serially transmitted through a single pin of the parallel port into the FPGA in slave-serial configuration mode.

A method is also provided to extract the target device from a bitstream file. This is used to verify that a bitstream file is being downloaded to the proper type of FPGA.

RAMPort

This object provides methods for uploading and downloading RAM of varying address and data widths through the XS Board parallel port interface.

The download method reads hexadecimal data records from Intel, Motorola or XESS HEX files and sends them to the XS Board. Before using this method, the FPGA or CPLD on the XS Board should be programmed with a state machine that manages the passage of the address and data information through the parallel port. The download method partitions the address and data into small bit fields that are passed to the state machine which re-assembles them and then loads the data into the given address location in RAM. The state machine returns an indicator of its current state through the parallel port status lines so the download method can detect any loss of synchronization during the RAM download.

The upload method works in a similar fashion. It passes an address through the parallel port to the state machine in the XS Board FPGA or CPLD. The CPLD returns the data from the given address Flash memory while the upload method is sending the next memory address.

AT17Port

This object provides methods for downloading a configuration bitstream to an Atmel AT17CXXX serial EEPROM. The bitstream file is opened and the field containing the configuration bits is extracted and partitioned into 64-byte page. Each page of data is transmitted serially through the parallel port where the FPGA connects the clock and data streams to the programming pins of the AT17CXXX chip.

FlashPort

This object provides general methods for uploading and downloading Flash memory of varying address and data widths through the XS Board parallel port interface. Virtual methods are provided for managing the low-level operations of erasing the Flash and writing individual bytes of Flash.

The download method reads hexadecimal data records from Intel, Motorola or XESS HEX files and sends them to the XS Board. Before using this method, the FPGA or CPLD on the XS Board should be programmed with a state machine that manages the passage of the address and data information through the parallel port. The download method partitions the address and data into small bit fields that are passed to the state machine which re-assembles them and then loads the data into the given address location in Flash memory. The state machine returns an indicator of its current state through the parallel port status lines so the download method can detect any loss of synchronization during the Flash download.

The upload method works in a similar fashion. It passes an address through the parallel port to the state machine in the XS Board FPGA or CPLD. The CPLD returns the data from the given Flash memory address while the upload method is sending the next memory address.

F28Port

This object specializes the FlashPort object by providing methods for erasing blocks and writing individual bytes of an Intel 28FXXX Flash memory.

AT49FPort

This object specializes the FlashPort object by providing methods for erasing blocks and writing individual bytes of an Atmel AT49FXXX Flash memory.

OscPort

This object provides a method for programming the Dallas DS1075 oscillator chip. The specified divisor for the master frequency is partitioned into a divisor of 513 or less and a frequency prescaler of 2 or 4 (if necessary). The divisor value is programmed into the divisor register of the oscillator by sending a precisely timed waveform through the parallel port pin that connects to the oscillator chip output (which operates as an input during the DS1075 programming mode.) Then the prescaler value and the bit that selects the source for the master oscillator are loaded into the multiplexer register.

JTAGPort

This object adds JTAG capabilities to the PPort object. It stores the state of the JTAG TAP state machine and updates the state as the TMS and TCK signals change state. It provides methods that make it easier to move between states of the TAP machine and to access the boundary scan instruction and data registers.

XC95KPort

This object specializes the JTAGPort object by adding a method that reads SVF files and sends the configuration commands they contain to the XC9500 CPLD through the parallel port. Methods are also provided for reading the contents of the device ID and USERCODE signature registers in the XC9500 CPLD.

JTAGRAMPort

This object specializes the JTAGPort by adding methods to support RAM upload and download operations using the JTAG circuitry of the XC9500 CPLD. The levels for the RAM address, data and control pins are shifted into the appropriate bits of the XC9500 boundary-scan data register (BSDR) through the parallel port. The JTAG EXTEST instruction outputs these levels onto the XC9500 pins that connect to the RAM chip. Then the levels on the RAM pins are sampled back into the XC9500 BSDR and are sent back through the parallel port. This low-level operation for reading and writing RAM locations via the JTAG circuitry forms the basis of the higher-level methods for downloading and uploading Intel, Motorola and XESS HEX files to and from the RAM.

Bitstream

This object allows the creation of arbitrary-length binary strings and permits some basic operations on them. It is used primarily for handling the strings of instruction and data register bits that go through the JTAG port.

XCBSDR

This object adds methods to the Bitstream object that make it easier to access the address, data, and control bits of the XC9500 BSDR.

Hex

This object stores a hexadecimal number consisting of eight hex digits or less. It is mainly used when reading a data record from an Intel, Motorola or XESS HEX file.

HexRecord

This object implements a generic hexadecimal data record with a buffer containing data bytes and a base address that records the starting address in memory for the data. This object has methods for entering data, reading the data back, determining the type of the data and setting/querying the starting address. Operators are also provided that allow storing/loading of HexRecord objects to/from Intel, Motorola and XESS HEX files.

Progress

This object provides feedback to the user concerning the percentage of a task that has been completed. A progress bar is displayed in a GUI environment while a rolling percentage indicator is used in a command-line environment.

XSError

This object extends stream objects like cerr so that they report errors with a consistent format. Each error message starts with a header that typically reports the name of the program or method where the error occurred. The severity of the error is also indicated. If the severity is high enough, the object will terminate the entire program. Otherwise, the object will record the number of each type of error that occurred. Later, the calling program can query whether an error occurred and decide what action to take.

This object also stores error messages and displays them in a Windows message window or as text in a command-line environment.

Utils

The utils.cpp file contains a set of miscellaneous subroutines that provide the following functions:

- Extracting file name prefixes and suffixes,
- Inserting precise time delays,
- Providing prompts to the user and collecting their response,
- Finding the directory where the XSTOOLS utilities are stored,
- Writing and reading control parameters to and from the XSTOOLS parameter file and the Windows registry (if present),
- Reading the XS Board description file into an internal data structure.

XSTOOLs Command-Line Programs

The descriptions of the steps carried out by each command-line XSTOOLS utility are described below

XSTEST

XSTEST begins by fetching the XS Board model and parallel port identifier used by the most recent invocation of any XSTOOLS utility. This allows the user to specify a particular board model and/or parallel port as arguments to an XSTOOLS utility and then these arguments will be inferred from then on unless specifically overridden by a new set of arguments.

Next, the model identifier is used to select which XS Board object will be instantiated. The object is setup to use the specified parallel port for communication. Then the Test method for the board object runs and displays the results of the test to the user.

XSLOAD

XSLOAD begins by fetching the XS Board model, parallel port identifier and RAM and Flash file formats used by the most recent invocation of any XSTOOLS utility. This allows the user to specify these settings once and then these arguments will be inferred from then on unless specifically overridden by a new set of arguments.

Next, the model identifier is used to select which XS Board object will be instantiated. Then the object is setup to use the specified parallel port for communication.

If the arguments to XSLOAD specify a RAM or Flash upload operation, then the appropriate upload method of the XS Board object is called and the data is stored in the specified file.

If the arguments to XSLOAD specify a RAM/Flash download, then the contents of any HEX data files are transferred to the RAM or Flash using the appropriate download method of the XS Board object. After the RAM/Flash download completes, if any arguments to XSLOAD specify a CPLD or FPGA configuration operation, then the given bitstream file is downloaded to the appropriate programmable device.

XSSETCLK

XSSETCLK begins by fetching the XS Board model and parallel port identifier used by the most recent invocation of any XSTOOLS utility. This allows the user to specify a particular board model and/or parallel port as arguments to an XSTOOLS utility and then these arguments will be inferred from then on unless specifically overridden by a new set of arguments.

Next, the model identifier is used to select which XS Board object will be instantiated. The object is setup to use the specified parallel port for communication. Then the SetFreq method for the board object is invoked with the frequency divisor and master frequency source as arguments.

XSPORT

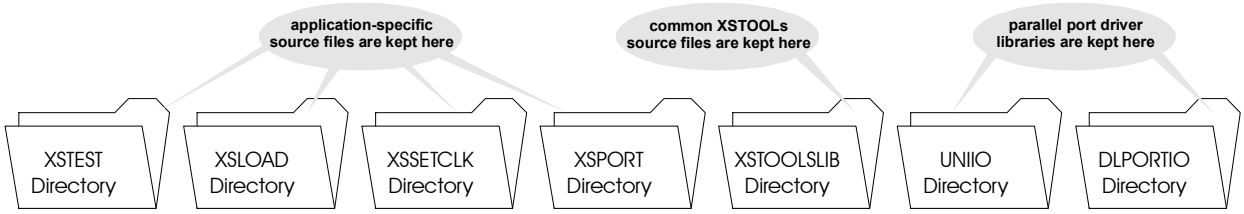
XSPORT begins by fetching the parallel port identifier used by the most recent invocation of any XSTOOLS utility. This allows the user to specify a particular parallel port as an argument to an

XSTOOLS utility and then it will be inferred from then on unless specifically overridden by a new set of arguments.

Next, a new PPort object is created that connects to the specified parallel port. The current levels on the parallel port data pins are read and stored. Then the binary string passed as an argument to XSPORT is parsed and the ones and zeroes in the string replace the bits in the stored parallel port data value. Finally, the updated data is output on the parallel port data pins and the program terminates.

Compiling the XSTOOLs Programs

The XSTOOLs source code was written and compiled using Microsoft Visual C++© 5.0. The ZIP file containing this document also contains the VC5 project workspaces and files for the XSTOOLs programs. A makefile for each utilitiy is also provided. The file organization is shown in **Figure 2**.



• **Figure 2:** XSTOOLs source directory structure.

The easiest way to use the XSTOOLS utilities after they are compiled is to install the official release of XSTOOLS 4.0. This will install the necessary support files and set the environment and registry variables so the compiled executables can find them.