XESS Corporation

XSTOOLs Source Documentation 3.0

Architecture of the XSTOOLs software

RELEASE DATE: 9/10/2000

Table of Contents

Introduction

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

XSLOAD is a command line program that is used to configure the programmable device, download/upload the RAM, or set the oscillator frequency on either an XS40 or XS95 Board. XSLOAD can also program an Atmel serial EEPROM if it is placed in the socket on the XS40 Board. XSLOAD can also configure the XC9500 CPLD and the Virtex FPGA on the XSV Boards (but it cannot download or upload the XSV Board RAMs).

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 FPLD so as to test a downloaded logic circuit.

GXSLOAD is a version of XSLOAD that uses a Windows graphical interface.

GXSPORT is a version of XSPORT that uses a Windows graphical interface.

GXSSETCLK is used to set the oscillator frequency of the XS95, Xs40, or XSV Boards through a Windows graphical interface.

GXSTEST is used to test the XS95, XS40, and XSV Boards through a Windows graphical interface.

The XSTOOLs utilities make use of the objects and function libraries shown in **Figure 1**:

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

JTAGPort: This object is built on top of the Pport object and adds JTAG capabilities to the parallel port. The loading of the RAM on the XS95 Board is done using JTAG as is the programming of the XC9500 CPLD on the XS95 Board.

XCPort: This object extends the JTAGPort object to provide the virtual methods that link to the actual methods for downloading the RAM and programming the FPGA or CPLD on the XS40 or XS95 Boards. It also provides methods to set the frequency of the programmable oscillator on the XS Boards.

XC40: This object adds to the XCPort object the capabilities for programming the XC4000 FPGA and the Atmel serial EEPROM on the XS40 Board. It also adds the methods for uploading and downloading the RAM on the XS40 Board.

XC95: This object adds to the XCPort object the methods for programming the XC9500 CPLD and for uploading and downloading the RAM on the XS95 Board.

XSV95: This object adds to the XC95 object the methods for programming the Flash on the XSV Board.

Bitstream: This object just stores arbitrary-length strings of binary bits.

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. The XCBSDR object provides methods that make it easy to access the XS95 Board RAM through the JTAG interface.

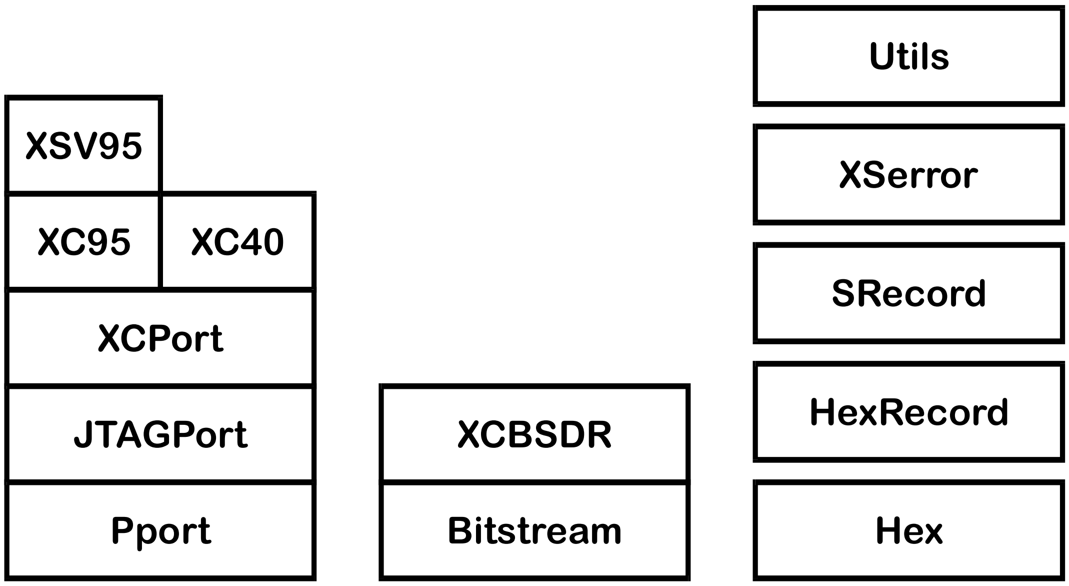
Hex: This object is used for storing hexadecimal data with an arbitrary number of digits. (But we only use hexadecimal data of four digits or less.)

HexRecord: This object stores data records as found in Intel HEX files. (The data loaded into the RAM is stored in such files.) HexRecords make use of Hex objects, but they do not inherit anything from the Hex object.

SRecord: This object stores data records as found in Motorola S files. (The data loaded into the XSV Flash is stored in such files.) SRecords make use of Hex objects, but they do not inherit anything from the Hex object.

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 functions that are used by other subroutines and methods.



• **Figure 1:** Source modules for the XSTOOLs.

Object Declarations

The header files for each of the objects and the utility functions are listed in the following sections.

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. The Pport object lets you specify inversion masks for each of these registers to counter the effects of the inverters found in the PC parallel port and XS Board hardware. For example, the XS95 and XS40 Boards have inverters on the two lowest bits of the parallel port data pins. Setting the data register inversion mask to 0x03 (00000011) will automatically invert the two lower bits of any byte written to the data register. This inversion will cancel the effect of the inverters.

```
//
// Pport objects handle the parallel port data, status, and
// control pins.
//

#ifndef Pport_H
#define Pport_H

#include "xserror.h"

class Pport
{
 public:

 /* begin Pport methods */

 /* method: Pport */
 // control register address offset
 // Constructor for a parallel port object.  Inversion masks are used to
 // correct the effects of the various inverters in the PC and XS Board.
  Pport(XSError* e, // error reporting channel
          unsigned int n=1, // parallel port number
          unsigned int dInvMask=0, // inversion mask for data output bits
          unsigned int sInvMask=0, // inversion mask for status input bits
          unsigned int cInvMask=0 // inversion mask for control output bits
          );

 /* method: ~Pport */
 // Destructor for parallel port object
  ~Pport(void);

 /* method: GetErr */
 // provides access to the error reporting channel
 XSError& GetErr(void);

 /* method: operator= */
 // Assignment operator for parallel port objects
 Pport& operator=(Pport& src);

 /* method: SetNum */
 // Sets up a port object for a given parallel port n
 void SetNum(unsigned int n);

 /* method: GetNum */
 // Gets the parallel port number for a parallel port object
 unsigned int GetNum(void) const;
```

```
 //          Determines I/O address for the given parallel port # (1-4).
```

```
/* method: SetDataInvMask */
// Set the data register inversion mask
void SetDataInvMask(unsigned int invMask=0);

/* method: SetData */
// Output a value on the designated data pins of the parallel port
void SetData(
		unsigned int v, // value to output
		unsigned int loPos=0, // low bit position of field
		unsigned int hiPos=7 // high bit position of field
		);

/* method: GetData */
// Return the current values on the designated data pins of the parallel port
unsigned int GetData(
		unsigned int loPos=0, // low bit position of field
		unsigned int hiPos=7 // high bit position of field
		) const;

/* method: SetStatusInvMask */
// Set the status register inversion mask
void SetStatusInvMask(unsigned int invMask=0);

/* method: GetStatus */
// Return the current values on the designated status pins of the parallel port
unsigned int GetStatus(
		unsigned int loPos=0, // low bit position of field
		unsigned int hiPos=7 // high bit position of field
		) const;

/* method: SetControlInvMask */
// Set the control register inversion mask
void SetControlInvMask(unsigned int invMask=0);

/* method: SetControl */
// Output a value on the designated control pins of the parallel port
void SetControl(
		unsigned int v, // value to output
		unsigned int loPos=0, // low bit position of field
		unsigned int hiPos=7 // high bit position of field
		);

/* method: GetControl */
// Return the current values on the designated control pins of the parallel port
unsigned int GetControl(
		unsigned int loPos=0, // low bit position of field
		unsigned int hiPos=7 // high bit position of field
		) const;
```

```
/* end Pport methods */
```

```
  XSError* err;                   // error reporting object
  unsigned int num;               // parallel port num 1,2,3,4
  unsigned int address;           // I/O address
  unsigned int dataInvMask;       // inversion mask for data outputs
  unsigned int statusInvMask;     // inversion mask for status inputs
  unsigned int controlInvMask;    // inversion mask for control outputs
};

/* begin Pport functions */

/* function: Inp */
//
// Here are some routines for handling primitive read/write of ports
//
// get data byte from a given 16-bit I/O address
static unsigned char Inp(unsigned short address);

/* function: Outp */
// send data byte to a given 16-bit I/O address
static void Outp(unsigned short address, unsigned char byte);

/* function: CheckBitPos */
// Check for bit position index which is too large
static bool CheckBitPos(unsigned int pos, // bit position
  XSError& err // error reporting channel
  );

/* function: GetField */
// Get the value stored in a bit field
static unsigned int GetField(unsigned int data, // port data
  unsigned int loPos, // low bit position of field
  unsigned int hiPos // high bit position of field
  );

/* function: SetField */
// Set the value in a bit field and return the entire data value after the
// field is updated.
static unsigned int SetField(unsigned int data, // port data
  unsigned int loPos, // low bit position of field
  unsigned int hiPos, // high bit position of field
  unsigned int newData // new data for field
  );

/* end Pport functions */

#endif
```

JTAGPort

This object adds JTAG capabilities to the parallel port 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 some methods that make it easier to move between states of the TAP machine and to access the boundary scan instruction and data registers.

```
//
// JTAGPort objects add JTAG capabilities to a parallel port object.
// A JTAG port can control the TCK, TMS, and TDI pins and read the
// TDO pin.  It stores and updates the state of the TAP state machine
// based on the TMS level at every TCK pulse.
//

#ifndef JTAGPort_H
#define JTAGPort_H

#include <assert.h>
#include <stdarg.h>

#include "bitstrm.h"
#include "pport.h"


// identifiers for all possible TAP states
typedef enum
{
  TestLogicReset=0, RunTestIdle=1,
              SelectDRScan=2,    SelectIRScan=3,
              CaptureDR=4,       CaptureIR=5,
              ShiftDR=6,         ShiftIR=7,
              Exit1DR=8,         Exit1IR=9,
              PauseDR=10,        PauseIR=11,
              Exit2DR=12,        Exit2IR=13,
              UpdateDR=14,       UpdateIR=15
}TAPState;


class JTAGPort : public Pport
{
  public:

  /* begin JTAGPort methods */

  /* method: JTAGPort */
  // Create a JTAG controller port
   JTAGPort(XSError* e, // pointer to error reporting object
              unsigned int portNum, // parallel port number
              unsigned int dInvMask, // inversion mask for the data pins
              unsigned int sInvMask, // inversion mask for the status pins
              unsigned int cInvMask) // inversion mask for the control pins
```

```
/* method: SetTCK */
// Set the value of the JTAG clock bit
void SetTCK(unsigned int b);

/* method: GetTCK */
// Get the value output on the JTAG clock bit
unsigned int GetTCK(void);

/* method: PulseTCK */
// Toggle the TCK output twice (return it to its original level)
void PulseTCK(void);

/* method: SetTMS */
// Set the value of the JTAG TMS bit
void SetTMS(unsigned int b);

/* method: GetTMS */
// Get the value output on the JTAG TMS bit
unsigned int GetTMS(void);

/* method: SetTDI */
// Set the value of the JTAG TDI bit
void SetTDI(unsigned int b);

/* method: GetTDI */
// Get the value output on the JTAG TDI bit
unsigned int GetTDI(void);

/* method: GetTDO */
// Get the value of the JTAG TDO output that enters the port
unsigned int GetTDO(void);

/* method: InitTAP */
// Initialize the TAP state controller and the attached JTAG device
void InitTAP(void);

/* method: SetTraceOnOff */
// enables/disables trace of JTAG signals
void SetTraceOnOff(bool f, // trace if true, no trace if false
		ostream& os=cerr // trace info goes to this stream
		);

/* method: GetTAPStateLabel */
// returns the text label associated with the given state
string GetTAPStateLabel(TAPState s);

/* method: GotoNextTAPState */
// Sets the TMS line to the appropriate value such that the next TCK pulse

// will move the TAP controller to the requested nextState.

// If the new state of the TAP controller doesn't match the requested
// state, the subroutine will cause the program to abort.
```

```
    /* method: GoThruTAPSequence */
    // Moves through a sequence of TAP controller states.
    // Terminate state changes when a -1 is reached in the sequence.
    void GoThruTAPSequence(TAPState nextState, ...);

    /* method: SendRcvBit */
    // Return the bit from TDO, send a bit to TDI, and pulse TCLK.
    // This subroutine assumes that the TCK is 0 and the TAP controller state
    // is Shift-DR or Shift-IR upon entry.  That's why we pick up the value
    // coming in through TDO right away.
    unsigned int SendRcvBit(unsigned int sendBit);

    /* method: SendRcvBitstream */
    // Transmit a bitstream through TDI while receiving a bitstream through TDO.
    //
    // This subroutine assumes the TAP controller state is
    // Shift-IR or Shift-DR upon entry.  Upon termination, this subroutine
    // will leave the TAP controller in the Exit1-IR or Exit1-DR state.
    //
    // Either sendBits or rcvBits can be a zero-length bitstream if you don't
    // want to transmit or receive bits during a particular call to this subroutine.
    // For example, you may want to load the BSDR with a bit pattern but
    // you might not care what data gets shifted out of the BSDR during
    // the loading.
    //
    // SendBits and RcvBits can point to the same bitstream without
    // causing problems.
    //
    // Note that the LSB of a bitstream has an index of 0.
    void SendRcvBitstream(Bitstream& sendBits, Bitstream& rcvBits);

    /* method: LoadBSIRthenBSDR */
    // Load the BSIR with an instruction, execute the instruction,
    // and then capture and reload the BSDR.
    //
    // This subroutine assumes the TAP controller is in the Test-Logic-Reset
    // or Run-Test/Idle state upon entry.  The TAP controller is returned to the
    // Run-Test/Idle state upon exit.
    void LoadBSIRthenBSDR( Bitstream& instruction,
              Bitstream& send, Bitstream& recv );

    /* end JTAGPort methods */

    private:

    TAPState currentTAPState;          // state of Test Access Port

    bool traceFlag;               // trace JTAG states on/off

    ostream* osTrace;    // output stream for trace info

   };
```

```
/* end JTAGPort functions */

#endif
```

XCPort

XCPort builds on JTAGPort by adding virtual functions that link to the actual methods in the XC40 and XC95 classes for programming the FPGA or CPLD. Virtual functions are also added to link to the routines for uploading and downloading the XS Board RAM. Methods to set the frequency of the programmable oscillator on the XS Board are also provided.

```
//
// XCPort objects specialize a JTAGPort to handle programming of
// XS40 and XS95 Boards.
//

#ifndef XCPort_H
#define XCPort_H

#include <fstream.h>

#include "xcbsdr.h"
#include "jtagport.h"
#include "hexrecrd.h"

const unsigned int MILLISECONDS = 1000;
const unsigned int MICROSECONDS = 1;

class XCPort : public JTAGPort
{
  public:

  /* begin XCPort methods */

  /* method: XCPort */
  // instantiate an XCPort object on a given parallel port
   XCPort(XSError* e, unsigned int portNum);

  /* method: SetChipType */
  // sets the chip identifier in the object
  bool SetChipType(string& type);
```

```
  // configure the FPLD so it can interface to the XS Board RAM
  virtual bool ConfigureRAMInterface() = 0;
```

```
// download the XS Board RAM with the contents of a HEX file
bool DownloadRAM(string& hexfileName);

/* method: DownloadRAM */
// download the XS Board RAM with the contents arriving through a stream
virtual bool DownloadRAM(istream& is) = 0;

/* method: DownloadHexRecordToRAM */
// load RAM data at the addresses given in a hex record
// (This routine assumes the entering TAP state is Shift-DR.
// It exits with the TAP state being Shift-DR.)
virtual void DownloadHexRecordToRAM(HexRecord& hx) = 0;

/* method: UploadRAM */
bool UploadRAM(
        string& hexfileName, // dump uploaded data to this file
        unsigned long loAddr, // start fetching data from this address
        unsigned long hiAddr // stop at this address
        );

/* method: UploadRAM */
// upload the XS Board RAM to a stream
virtual bool UploadRAM(
        ostream& os, // dump uploaded data to this stream
        unsigned long loAddr, // start fetching data from this address
        unsigned long hiAddr // stop at this address
        ) = 0;

/* method: UploadHexRecordFromRAM */
// get RAM data at the addresses given and store it in a hex record
// (This routine assumes the entering TAP state is Shift-DR.
// It exits with the TAP state being Shift-DR.)
virtual void UploadHexRecordFromRAM(
        HexRecord& hx, // hex record to store data in
        unsigned long loAddr, // begin upload from this address
        unsigned long hiAddr // end upload at this address
        ) = 0;

/* method: ConfigureFPLD */
// configure the FPLD with the bit stream stored in a BIT or SVF file
bool ConfigureFPLD(string& fileName);

/* method: ConfigureFPLD */
// configure the FPLD with the bit stream arriving through a stream
virtual bool ConfigureFPLD(istream& is) = 0;
```

```
/* method: ProgramEEPROM */
```

```
// program the Atmel serial EEPROM on the XS Board with a bitstream from a file
bool ProgramEEPROM(string& bitfileName);

/* method: ProgramEEPROM */
// program EEPROM on the XS Board
```

```
  /* method: SetOscFrequency */
  // set the DS1075 oscillator frequency
  void SetOscFrequency(int div, int extOscPresent=0);

  /* method: InsertDelay */
  // subroutine for delaying by a given number of microseconds
  void InsertDelay(unsigned long d, unsigned int time_units);


  protected:

  // The RAM of the XS board is controlled through a JTAG interface,
  string chipType;              // type of chip (e.g. 4005XLPC84)
  XCBSDR* bsdrPtr;              // JTAG data register useful for XS RAM access
  Bitstream* bsirPtr;           // JTAG instruction register
  bool ramInterfaceIsRdy;       // true if the interface to up/download to the XS board RAM is ready


  private:

  /* method: ResetOsc */
  // reset the programmable oscillator
  void ResetOsc();

  /* method: SendOscBit */
  // send a bit of data to the programmable oscillator.
  void SendOscBit(unsigned char b);

  /* method: IssueOscCmd */
  // send a command to the programmable oscillator
  void IssueOscCmd(unsigned int cmd);

  /* method: SendOscData */
  // send a data word to the programmable oscillator
  void SendOscData(unsigned int data);

  /* end XCPort methods */

};

/* begin XCPort functions */

/* end XCPort functions */
```

```
#endif
```

XC40

The XC40 object contains the actual methods for programming the XC4000 or Virtex FPGA on the XS40 or XSV Boards and the serial EEPROM and the RAM on the XS40 Board. The programming of the XC4000 or Virtex FPGA is done through parallel port pins attached to the PROGRAM, DIN, and CCLK pins of the FPGA. The RAM on the XS40 Board is accessed by programming the XC4000 FPGA with a state machine that manages the reading and writing processes under control of the parallel port. This state machine is wiped out when the user-specified bitstream is loaded into the FPGA.

```
//
// The XC40 object specializes the XCPORT object so that the XC4000 FPGA on the
// XS40 Board can be configured and the RAM can be loaded.  The XC40 object controls
// the /PROGRAM, DIN, and CCLK pins.
//

#ifndef XC40_H
#define XC40_H

#include "xcport.h"

class XC40 : public XCPort
{
 public:

 /* begin XC40 methods */

 /* method: XC40 */
 // Create an XC40 object.  The type of XC4000 FPGA on the XS40 Board is inferred
 // by reading the chip type from a bitfile that is passed to the constructor.
 // The directory containing the RAM interface bitstreams for all the different types
 // of XC4000 chips is also passed as an argument.
  XC40(XSError* e, // error reporting channel
            unsigned int portNum, // parallel port number
            string& bitfileName, // bitfile with XC4000 configuration bitstream
            string& intfcDir // directory where RAM interface bitstreams are stored
            );

 /* method: ConfigureFPLD */
 // Send out a byte of configuration information
 void ConfigureFPLD(unsigned char b);

 /* method: ConfigureRAMInterface */
 // configures an FPGA so the XS Board RAM can be uploaded/downloaded
 bool ConfigureRAMInterface();
```

```
 /* method: ProgramEEPROM */
 // program an Atmel serial EEPROM with a bitstream
 bool ProgramEEPROM(istream& is);

 /* method: DownloadRAM */
```

```
bool DownloadRAM(istream& is);

/* method: UploadRAM */
// upload the XS Board RAM to a stream
bool UploadRAM(
        ostream& os,            // dump uploaded data to this stream
        unsigned long loAddr, // start fetching data from this address
        unsigned long hiAddr // stop at this address
        );

private:

/* method: DownloadHexRecordToRAM */
// load RAM data at the addresses given in a hex record
void DownloadHexRecordToRAM(HexRecord& hx);

/* method: UploadHexRecordFromRAM */
// get RAM data at the addresses given and store it in a hex record
void UploadHexRecordFromRAM(
        HexRecord& hx, // hex record to store data in
        unsigned long loAddr, // begin upload from this address
        unsigned long hiAddr // end upload at this address
        );

/* method: SetPROGRAM */
// Set the value on the /PROGRAM pin of the XC4000 FPGA
void SetPROGRAM(unsigned int b);

/* method: GetPROGRAM */
// Get the value output on the configuration initiation pin
unsigned int GetPROGRAM(void);

/* method: PulsePROGRAM */
// Toggle the /PROGRAM output (return it to its original level)
void PulsePROGRAM(void);

/* method: SetCCLK */
// Set the value of the configuration clock bit
void SetCCLK(unsigned int b);

/* method: GetCCLK */
// Get the value output on the configuration clock bit
unsigned int GetCCLK(void);
```

```
/* method: PulseCCLK */
```

```
// Toggle the CCLK output (return it to its original level)
```

```
void PulseCCLK(void);

/* method: SetDIN */
```

```
  // Set the value of the configuration data bit
  void SetDIN(unsigned int b);

  /* method: SetTMS */
  // Get the value output on the configuration data bit
  unsigned int GetDIN(void);

  /* method: InitConfigureFPLD */
  // Initialize the XC4000 FPGA for configuration with DIN, CCLK, and PROGRAM pins
  void InitConfigureFPLD(void);

  /* method: ConfigureFPLD */
  // field type for the bitstream data
  // process a stream and send the configuration bitstream to the board
  bool ConfigureFPLD(istream& is);

  /* method: SetChipType */
  // field type for the FPGA device identifier
  // gets the chip identifier from the bitstream and stores it in the object
  bool SetChipType(istream& is);

  /* method: SendEEPROMByte */
  // Send a byte into an Atmel EEPROM
  void SendEEPROMByte(unsigned char byte);

  /* method: ProgramEEPROMPage */
  // Program a page of an Atmel EEPROM
  void ProgramEEPROMPage(unsigned char* buf, // buffer with data bytes
            unsigned int pageAddr, // address where data will be stored
            unsigned int len // number of bytes in buffer
            );

  /* method: SetChipType */
  // gets the chip identifier from the bitstream and stores it in the object
  bool SetChipType(string& bitfileName);

  /* end XC40 methods */

  string ramInterfaceDir;         // directory where RAM interface bitstreams are found
};

/* begin XC40 functions */

/* function: GetInteger */
// compute an integer from a vector of hex numbers in a stream
static long unsigned int GetInteger(istream& is, int len=2);
```

```
/* function: GetType */
```

```
// get the field type indicator from a stream
static int GetType(istream& is);
```

```
// pass over fields until a certain field is found
static bool ScanForField(istream& is, unsigned char searchType);

/* end XC40 functions */

#endif
```

XC95

The XC95 object specializes the XCPort object so it can program the CPLD on the XS95 or XSV Board and access the RAM on the XS95 Board. The primary method of the XC95 object is the method for reading an SVF file and programming the XC9500 CPLD. This is done through the JTAG port.

This object also provides a method to access the XC9500 CPLD chip identifier through the JTAG port. This chip identifier is used by the RAM downloading method to determine which pins of the JTAG boundary scan data register are connected to the XS40 RAM address, data, and control pins. Then JTAG commands are used to download and upload the RAM.

```
//
// The XC95 object specializes the XCPORT object so that the XC9500 CPLD on the
// XS95 Board can be configured and the RAM can be loaded.  The XC95 object controls
// the JTAG port of the XC9500 chip.
//

#ifndef XC95_H
#define XC95_H

#include "xcport.h"

// definitions of XC9500 CPLD types
enum{
 XC9572Type = 0,
 XC95108Type = 1,
 UnknownXC9500Type = 2,
};

class XC95 : public XCPort
{
 public:

 /* begin XC95 methods */

 /* method: XC95 */
 // instantiate an object for loading an XS95 board on the parallel port

  XC95(XSError* err, unsigned int portNum);

 /* method: ConfigureRAMInterface */
```

```
bool ConfigureRAMInterface();

/* method: ProgramEEPROM */
// program an Atmel serial EEPROM with a bitstream
bool ProgramEEPROM(istream& is);

/* method: ConfigureFPLD */
// Send out a byte of configuration information to the XC9500 Flash
void ConfigureFPLD(unsigned char b);

/* method: DownloadRAM */
// download the XS Board RAM with the contents arriving through a stream
bool DownloadRAM(istream& is);

/* method: UploadRAM */
// upload the XS Board RAM to a stream
bool UploadRAM(
		ostream& os, // dump uploaded data to this stream
		unsigned long loAddr, // start fetching data from this address
		unsigned long hiAddr // stop at this address
		);

private:

/* method: DownloadHexRecordToRAM */
// load RAM data at the addresses given in a hex record
// (This routine assumes the entering TAP state is Shift-DR.
// It exits with the TAP state being Shift-DR.)
void DownloadHexRecordToRAM(HexRecord& hx);

/* method: UploadHexRecordFromRAM */
// get RAM data at the addresses given and store it in a hex record
// (This routine assumes the entering TAP state is Shift-DR.
// It exits with the TAP state being Shift-DR.)
void UploadHexRecordFromRAM(
		HexRecord& hx, // hex record to store data in
		unsigned long loAddr, // begin upload from this address
		unsigned long hiAddr // end upload at this address
		);

/* method: InitConfigureFPLD */
// Initialize the XC9500 CPLD for configuration through the JTAG interface
void InitConfigureFPLD(void);

/* method: ConfigureFPLD */
// Process SVF and send results through JTAG configuration pins
bool ConfigureFPLD(istream& is);
```

```
/* method: DetectXC9500Type */
// detect the presence and type of the XC9500 CPLD chip on the port
bool DetectXC9500Type();
```

```
  private:
};

/* begin XC95 functions */

/* function: NextWord */
static string NextWord(string& s);

/* end XC95 functions */

#endif
```

XSV95

The XSV95 object specializes the XC95 object so it can program the Flash RAM on the XSV Board.

```
#ifndef XSV95_H
#define XSV95_H

#include "srecrd.h"
#include "xc95.h"

class XSV95 : public XC95
{
  public:

  /* begin XSV95 methods */

  /* method: XSV95 */
  // instantiate an object for loading an XSV Board
  XSV95(XSError* err, unsigned int portNum, string& intfcDir);

  /* method: ConfigureFlashInterface */
  // configure the XC95108 on the XSV for programming the Flash RAM
  bool ConfigureFlashInterface(void);

  /* method: ConfigureDefaultFlashConfigInterface */
  // configure the XC95108 on the XSV for programming the Virtex from the Flash RAM
  bool ConfigureDefaultFlashConfigInterface(void);
```

```
  /* method: DownloadFlash */
```

```
  // download the XSV Flash with the contents of an EXO file
  bool DownloadFlash(string& exofileName);

  /* method: DownloadFlash */
  // download the XSV Flash with the contents arriving through a stream
```

```
/* method: DownloadHexRecordToFlash */
// load RAM data at the addresses given in a hex record
void DownloadHexRecordToFlash(SRecord& hx);

/* method: UploadFlash */
// upload the XSV Flash to a file
bool UploadFlash(
		string& exofileName, // dump uploaded data to this file
		unsigned long loAddr, // start fetching data from this address
		unsigned long hiAddr // stop at this address
		);

/* method: UploadFlash */
// upload the XSV Flash to a stream
bool UploadFlash(
		ostream& os, // dump uploaded data to this stream
		unsigned long loAddr, // start fetching data from this address
		unsigned long hiAddr // stop at this address
		);

/* method: UploadHexRecordFromFlash */
// get Flash data at the addresses given and store it in a hex record
void UploadHexRecordFromFlash(
		SRecord& hx, // hex record to store data in
		unsigned long loAddr, // begin upload from this address
		unsigned long hiAddr // end upload at this address
		);

/* method: ReadFlashByte */
// read a byte from the Flash through the parallel port
unsigned int ReadFlashByte(unsigned int address);

/* method: WriteFlashByte */
// write a byte to the Flash through the parallel port
void WriteFlashByte(unsigned int address, unsigned int data);

/* method: ResetFlash */
// reset the Flash so data can be read from it
void ResetFlash();

/* method: ReadFlashID */
// read data from the Flash
unsigned int ReadFlashID();
```

```
/* method: ReadFlashStatus */
// get the status of the Flash
unsigned int ReadFlashStatus();

/* method: ClearFlashStatus */
// clear any error bits in the Flash status register
void ClearFlashStatus();
```

```
  /* method: LockFlashBlock */
  void LockFlashBlock(unsigned int address);

  /* method: MasterLockFlash */
  // lock all the block lock bits so they cannot be altered
  void MasterLockFlash();

  /* method: UnlockFlashBlocks */
  void UnlockFlashBlocks();

  /* method: ProgramFlash */
  // Program a byte in the Flash
  void ProgramFlash(unsigned int address, unsigned int data);

  /* method: EraseFlashBlock */
  // erase the contents of a Flash block
  void EraseFlashBlock(unsigned int address);

  /* end XSV95 methods */

  private:
  bool flashInterfaceIsRdy;
  string flashInterfaceDir;
};

/* begin XSV95 functions */

/* end XSV95 functions */

#endif
```

Bitstream

The Bitstream 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.

```
//
// Bitstream objects just store strings of boolean values and
// allow certain operations to be performed on them
//
```

```
#ifndef Bitstream_H
#define Bitstream_H

#include <iostream.h>
#include <assert.h>
#include <stdarg.h>
```

```
class Bitstream
{
 public:

 /* begin Bitstream methods */

 /* method: Bitstream */
 // allocates a bitstream containing at least n bits.
  Bitstream(unsigned int n);

 /* method: ~Bitstream */
 // frees the storage used by a bitstream.
  ~Bitstream(void);

 /* method: GetLength */
 // returns the number of bits in the bitstream
 unsigned int GetLength(void) const;

 /* method: operator== */
 // Compares two bitstreams.  Returns true if they match, false otherwise.
 bool operator==(Bitstream& b2) const;

 /* method: Subcompare */
 // compares the end of 1st bitstream against a subfield of 2nd bitstream
 bool Subcompare(unsigned int pos, // position in 1st bitstream
           Bitstream& b2, // 2nd bitstream
           unsigned int b2Pos) const // position in 2nd bitstream
           ;

 /* method: operator= */
 // copies contents of one bitstream into another (also adjusts size).
 Bitstream& operator=(Bitstream& b2);

 /* method: Copy */
 // makes a new copy of a bitstream
 Bitstream* Copy(void);

 /* method: operator[] */
 // returns the value of a bit from a bitstream.
 unsigned int operator[](unsigned int bitIndex) const;

 /* method: Clear */
 // clear a bitstream

 void Clear(void);


 /* method: SetBit */
 // sets the value of a bit in a bitstream
 void SetBit(unsigned int bitIndex, // position to set
           unsigned int val // value to set bit
           );

 /* method: SetBits */
```

```
  // with a number that is neither 0 or 1.
  void SetBits(unsigned int bitIndex, // position to start changing bits
               int firstBit, ... // list of bit values
               );

  /* method: operator^ */
  // XOR two bitstreams and return the resulting bitstream.
  Bitstream& operator^(Bitstream& b2) const;

  /* method: operator+ */
  // concatenate two bitstreams to create a third bitstream
  Bitstream& operator+(Bitstream& b2) const;

  /* method: printBits */
  // prints a bitstream to an output stream
  ostream& printBits(ostream& os) const;

  private:

  unsigned int numBits;          // number of bits in bitstream
  unsigned long* bits;   // storage for bitstream

  /* end Bitstream methods */
};

/* begin Bitstream functions */

/* function: operator<< */
// helper function which prints a bitstream to an output stream
ostream& operator<<(ostream& os, Bitstream& b);

/* end Bitstream functions */

#endif
```

XCBSDR

The XCBSDR object adds methods to the Bitstream object that make it easier to access the address, data, and control bits of the XS95 Board RAM through the JTAG port. The object maintains some arrays which store the positions in the boundary scan data register that connect to the RAM pins. These arrays are loaded by the XC95 object depending upon the type of CPLD chip that is on the XS95 Board. (This is a moot point now that XS95 Boards with XC9572 CPLDs are no longer supported and now only XC95108 CPLDs are used in the XS95 Boards.)

```
//
// The XCBSDR object specializes a bitstream into a boundary scan data register
// that is used to perform RAM operations via the JTAG port.

#ifndef XCBSDR_H
#define XCBSDR_H
```

```
#include "bitstrm.h"

const int numRAMAddressBits = 17;
const int numRAMDataBits = 8;
const int numRAMControlBits = 3;

class XCBSDR : public Bitstream
{
public:

  XCBSDR(unsigned int length,
                 unsigned int *addressBitPos,
                 unsigned int *dataBitPos,
                 unsigned int *controlBitPos);

  // place a RAM address into the bitstream buffer
  void SetRAMAddress(unsigned int address);

  // place RAM data into the bitstream buffer
  void SetRAMData(unsigned int data);

  // tristate the FPLD pins so the RAM data pins can be read
  void ReadRAMData(void);

  // get the RAM data from the bsdr bitstream
  unsigned int GetRAMData(void);

  // get the RAM address from the bsdr bitstream
  unsigned long GetRAMAddress(void);

  // place RAM control pin levels into the bsdr bitstream
  void SetRAMControls(unsigned int cs_,
                 unsigned int oe_,
                 unsigned int we_);

private:
  // The RAM of the XS board is controlled through a JTAG interface,
  // and these arrays hold the positions of the RAM pins in the data register
  // position of RAM address bits in data register
  unsigned int ramAddressBitPos[numRAMAddressBits];
  // position of RAM data bits in data register

  unsigned int ramDataBitPos[numRAMDataBits];

  // position of RAM control bits in data register
  unsigned int ramControlBitPos[numRAMControlBits];
};

#endif
```

Hex

This object stores a hexadecimal number consisting of eight hex digits or less. It is used mainly when reading a data record from an Intel HEX file (see the **HexRecord** section) or a Motorola S Record (see the **SRecord** section).

```
//
// Hex objects just store hexadecimal numbers with
// a set number of digits.
//

#ifndef Hex_H
#define Hex_H

#include <iostream.h>

class Hex
{
  public:

  /* begin Hex methods */

  /* method: Hex */
  // construct a hex number with a given number of hex digits
   Hex(unsigned int l=4);

  /* method: SetLength */
  // set the number of hex digits in the hex number
  void SetLength(unsigned int l);

  /* method: GetLength */
  // return the number of hex digits in the hex number
  unsigned int GetLength(void);

  /* method: SetHex */
  // assign a value to the hex number
  void SetHex(unsigned int h);

  /* method: operator= */
  // assign a value to the hex number
  Hex& operator=(unsigned int h);
```

```
  /* method: GetHex */
  // return the value of the hex number
  unsigned int GetHex(void);

  private:

  unsigned int hex;
  unsigned int length;
```

```
};

/* begin Hex functions */

/* function: CharToHex */
// helper function to convert ASCII '0'-'F' to number between 0 and 15, inclusive
unsigned int CharToHex(char c);

// helper function which gets a hex number from an input stream
istream& operator>> (istream& is, Hex& n);

// helper function which outputs a hex number to an output stream
ostream& operator<< (ostream& os, Hex& n);

/* end Hex functions */

#endif
```

HexRecord

This object stores the various types of data records found in an Intel HEX file. The HexRecord object is used to get the data from the file and then its contents are dumped into the RAM on the XS40 and XS95 Boards. The HexRecord object can also read hexadecimal data records of the type used in the last two chapters of "The Practical Xilinx Designer Lab Book".

```
//
// HexRecord objects store hexadecimal records consisting of
// a length, beginning address, hexadecimal data values, checksum,
// and tag.  These records are usually created by reading in
// an Intel hex file.
//

#ifndef HexRecord_H
#define HexRecord_H

#include <iostream.h>

// these are the various types of records found in a hex file
typedef enum
```

```
{
```

```
  DataRecord=0,           // actual address and data
          InfoRecord=1,           // some type of informational record
          Info2Record=2,          // another type of informational record
          InvalidRecord=3,  // erroneous record
          NoSumRecord=4,          // for storing bytes with no checksum
} HexRecordTagType;

// various types of errors that can occur when handling hex records
```

```
{
  NoHexRecordError,
                  StartHexRecordError,
                  LengthHexRecordError,
                  AddressHexRecordError,
                  TagHexRecordError,
                  DataHexRecordError,
                  CheckSumHexRecordError
} HexRecordError;


class HexRecord
{
  public:

  /* begin HexRecord methods */

  /* method: HexRecord */
  // construct and initialize an empty hex record
   HexRecord(void);

  /* method: ~HexRecord */
  // delete a hex record
   ~HexRecord(void);

  /* method: SetTag */
  // set the type tag of the hex record
  void SetTag(HexRecordTagType t);

  /* method: GetTag */
  // get the tag of the hex record
  HexRecordTagType GetTag(void) const;

  /* method: IsData */
  // does the hex record contain data?
  bool IsData(void) const;

  /* method: IsValid */
  // is the hex record valid?
  bool IsValid(void) const;

  /* method: SetLength */

  // Set the size of the data area in the hex record.  Delete existing data.
  void SetLength(unsigned int l);

  /* method: GetLength */
  // get the number of data bytes in the hex record
  unsigned int GetLength(void) const;

  /* method: SetAddress */
  // set the address at which the hex record data should be loaded
  void SetAddress(unsigned int addr);
```

```
    /* method: GetAddress */
    // get address at which hex record data will be loaded
    unsigned int GetAddress(void) const;

    /* method: operator[] */
    // index into the hex record to get a single byte
    unsigned char& operator[](unsigned int index) const;

    /* method: CalcCheckSum */
    // calculate the 8-bit checksum of the data in the hex record
    void CalcCheckSum(void);

    /* method: GetCheckSum */
    // get the calculated checksum of the hex record
    unsigned char GetCheckSum(void) const;

    /* method: SetError */
    // set the error flag in the hex record
    void SetError(HexRecordError e);

    /* method: GetError */
    // get the error flag from the hex record
    HexRecordError GetError(void) const;

    /* method: IsError */
    // is there an error in the hex record
    bool IsError(void) const;

    /* method: GetErrMsg */
    // return an error message for the hex record
    const char* GetErrMsg(void) const;

    private:

    HexRecordError err;                     // hex record error
    HexRecordTagType tag;                   // type of data stored in hex record
    unsigned int length;            // number of data bytes in the hex record
    unsigned int address;                   // destination address for the hex data
    unsigned char* data;            // data values stored in the hex record

    unsigned char checkSum;                 // checksum for the entire hex record


    /* end HexRecord methods */

  };

  /* begin HexRecord functions */

  /* function: ErrMsg */
  // return an error message for the given type of error
  const char* ErrMsg(HexRecordError e);

  // helper function that reads in a hex record from an input stream
```

```
// helper function that dumps a hex record to an output stream
extern ostream& operator<< (ostream& is, HexRecord& hx);

/* end HexRecord functions */

#endif
```

SRecord

This object stores the various types of data records found in an Motorola EXO file. The SRecord object is used to get the data from the file and then its contents are dumped into the Flash RAM on the XSV Board.

```
//
// S-record objects store hexadecimal records consisting of
// a length, beginning address, hexadecimal data values, checksum,
// and tag.  These records are usually created by reading in
// a Motoroal Exormacs file.
//

#ifndef SRecord_H
#define SRecord_H

#include <iostream.h>

// these are the various types of records found in an S-record file
typedef enum
{
  DataSRecord=0,                    // actual address and data
  InfoSRecord=1,                    // some type of informational record
  InvalidSRecord=3,    // erroneous record
} SRecordTagType;
```

```
// various types of errors that can occur when handling hex records
```

```
typedef enum
{
  NoSRecordError,
              StartSRecordError,
              LengthSRecordError,
              AddressSRecordError,
              TagSRecordError,
              DataSRecordError,
              CheckSumSRecordError
} SRecordError;


class SRecord
```

```
public:

/* begin SRecord methods */

/* method: SRecord */
// construct and initialize an empty S-record
 SRecord(void);

/* method: ~SRecord */
// delete an S-record
 ~SRecord(void);

/* method: SetTag */
// set the type tag of the S-record
void SetTag(SRecordTagType t);

/* method: GetTag */
// get the tag of the S-record
SRecordTagType GetTag(void) const;

/* method: IsData */
// does the S-record contain data?
bool IsData(void) const;

/* method: IsValid */
// is the S-record valid?
bool IsValid(void) const;

/* method: SetLength */
// Set the size of the data area in the S-record.  Delete existing data.
void SetLength(unsigned int l);

/* method: GetLength */
// get the number of data bytes in the S-record
unsigned int GetLength(void) const;
```

```
/* method: SetAddress */
```

```
// set the address at which the S-record data should be loaded
void SetAddress(unsigned int addr);

/* method: GetAddress */
// get address at which S-record data will be loaded
unsigned int GetAddress(void) const;

/* method: SetNumAddressBytes */
// set the number of bytes in the address at which the S-record data should be loaded
void SetNumAddressBytes(unsigned int numBytes);

/* method: GetNumAddressBytes */
// get the number of bytes in the address at which S-record data will be loaded
unsigned int GetNumAddressBytes(void) const;
```

```
 // index into the S-record to get a single byte
 unsigned char& operator[](unsigned int index) const;

 /* method: CalcCheckSum */
 // calculate the 8-bit checksum of the data in the S-record
 void CalcCheckSum(void);

 /* method: GetCheckSum */
 // get the calculated checksum of the S-record
 unsigned char GetCheckSum(void) const;

 /* method: SetError */
 // set the error flag in the S-record
 void SetError(SRecordError e);

 /* method: GetError */
 // get the error flag from the S-record
 SRecordError GetError(void) const;

 /* method: IsError */
 // is there an error in the S-record
 bool IsError(void) const;

 /* method: GetErrMsg */
 // return an error message for the S-record
 const char* GetErrMsg(void) const;

 private:

 SRecordError err;                       // S-record error
 SRecordTagType tag;                              // type of data stored in S-record
 unsigned int length;            // number of data bytes in the S-record
 unsigned int address;                   // destination address for the S-record data
 unsigned int numAddressBytes;           // number of bytes in destination address

 unsigned char* data;            // data values stored in the S-record

 unsigned char checkSum;                 // checksum for the entire S-record

 /* end SRecord methods */

};

/* begin SRecord functions */

/* function: ErrMsg */
// return an error message for the given type of error
const char* ErrMsg(SRecordError e);

// helper function that reads in an S-record from an input stream
extern istream& operator>> (istream& is, SRecord& hx);

// helper function that dumps an S-record to an output stream
extern ostream& operator<< (ostream& is, SRecord& hx);
```

```
/* end SRecord functions */

#endif
```

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.

```
//
// This object allows the reporting of errors with a standardized format.
//

#ifndef XSError_H
#define XSError_H

#include <iostream.h>
#include <string>

using std::string;

// severity of errors
typedef enum
```

```
{
```

```
  XSErrorMin,                    // minimum error index
          XSErrorNone,   // no error (not initialized)
          XSErrorMinor,  // minor error (no abort)
          XSErrorMajor,
          XSErrorFatal,          // fatal error (causes abort)
          XSErrorMax                    // maximum error index
} XSErrorSeverity;

// states of the error object
typedef enum
{
  XSErrorInitial,   // just starting error message
          XSErrorInMessage, // currently printing error message
} XSErrorState;


// error object which manages printing error messages and terminating
// the main program if necessary.
```

```
//		XSError err(cerr);
//		err.setHeader("XESS ");
//		err.setSeverity(XSErrorMinor);
//		err << "this is a minor error" << endl;
//		err.endMsg();
//  err.simpleMsg(XSErrorMinor,"this is another minor error");
class XSError : public ostream_withassign
{
  public:

  /* begin XSError methods */

  /* method: XSError */
  // init error object with output going to stream s
   XSError(ostream_withassign& s);

  /* method: ~XSError */
  // destruct error object
   ~XSError(void);

  /* method: operator= */
  // assign contents of one source error object to another
  XSError& operator=(XSError& src);

  /* method: GetNumErrors */
  // get number of errors of a certain type that have occurred
  unsigned int GetNumErrors(XSErrorSeverity s) const;

  /* method: SetNumErrors */
  // set number of errors of a certain type that have occurred
  void SetNumErrors(XSErrorSeverity s, unsigned int n);
```

```
  /* method: IsError */
  // returns true if errors were recorded by this error object
  bool IsError(void) const;

  /* method: SetSeverity */
  // set severity of next error message
  void SetSeverity(XSErrorSeverity s);

  /* method: SetHeader */
  // set header string for each error message
  void SetHeader(string& h);

  /* method: EndMsg */
  // end an error message
  void EndMsg(void);

  /* method: SimpleMsg */
  // a simple interface for sending an error message
  void SimpleMsg(XSErrorSeverity s, string& msg);
```

```
    // overload << operator to assist in displaying error messages in a window
    XSError& operator<<(long n);

    /* method: operator<< */
    // overload << operator to assist in displaying error messages in a window
    XSError& operator<<(const char*  msg);

    /* method: operator<< */
    // overload << operator to assist in displaying error messages in a window
    XSError& operator<<(string& msg);

    /* method: operator<< */
    // overload << operator to assist in displaying error messages in a window
    XSError& operator<<(ostream& s);

    /* end XSError methods */

    private:

    /* method: GetSeverity */
    XSErrorSeverity GetSeverity(void) const;

    /* method: SetState */
    void SetState(XSErrorState s );

    /* method: GetState */
    XSErrorState GetState(void) const;

    /* method: GetHeader */
```

```
    string& GetHeader(void);
```

```
    unsigned int numErrors[XSErrorMax];       // counters for various grades of errors
    XSErrorState state;                       // records state of the error reporting process
    XSErrorSeverity severity;                 // severity of current error report
    string header;                            // header for each error message
    string storedMsg;                         // stored error message for display in window
  };

  /* begin XSError functions */

  /* end XSError functions */

  #endif
```

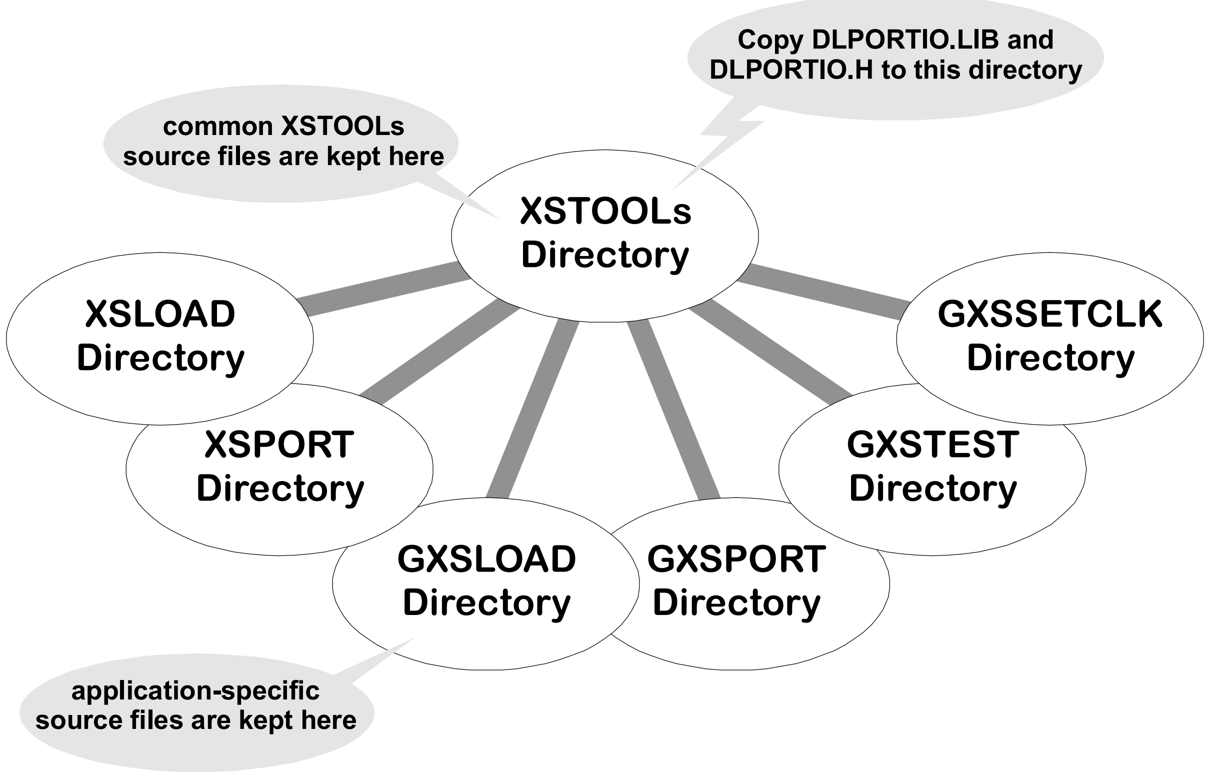
Utils

These are some utility routines used by other functions and methods.

```
#ifndef utils_H
#define utils_H

#include <string>
using std::string;

/* begin utils functions */

/* function: ConvertToUpperCase */
// convert a string to upper case
string& ConvertToUpperCase(string& s);

/* function: StripSuffix */
// strip suffix from a file name
string& StripSuffix(string& fileName);

/* function: GetSuffix */
// get suffix from a file name
string GetSuffix(string& fileName);

/* function: StripPrefix */
// strip prefixed directory path from a file name
string& StripPrefix(string& fileName);

/* function: GetPrefix */
// get prefixed directory path from a file name
string GetPrefix(string& fileName);

/* end utils functions */

#endif
```

Compiling the XSTOOLs Programs

The XSTOOLs source code was written and compiled in Microsoft Visual C++© 5.0. The ZIP file containing this document also contains the project workspaces and files for the XSTOOLs programs with the notable exception of the DLPORTIO.H and DLPORTIO.LIB files that provide an interface to the PC parallel port. These files are the copyrighted by Scientific Software Tools, Inc and must be distributed separately. You can get these files by downloading the PORT95NT.EXE file from http://www.xess.com/ho07000.html. Once you execute PORT95NT.EXE, you can copy the DLPORTIO files from their home directories into the XSTOOLS source directory. Then you can compile the XSTOOLs utilities.


Copy DLPORTIO.LIB and DLPORTIO.H to this directory
common XSTOOLs source files are kept here
XSTOOLs Directory
XSLOAD Directory
GXSSETCLK Directory
XSPORT Directory
GXSTEST Directory
GXSLOAD Directory
GXSPORT Directory
application-specific source files are kept here

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

Running the XSTOOLs

Once you have compiled the XSTOOLs programs, you can place them into the C:\XSTOOLS\BIN directory where V3.0 of the XSTOOLs are installed. This directory contains some bitstream files needed by the utilities to interface to the programmable oscillators on the XS Boards and the RAM on the XS40 Board.

Using XSLOAD

You can download an FPGA design into your XS40 Board as follows:

```
C:\> XSLOAD CIRCUIT.BIT
```

where CIRCUIT.BIT is an XC4000 or Spartan bitstream file that contains the configuration for the XC4000 or XCS FPGA. This file is created using the XILINX Foundation software tools. Make sure the file contains a bitstream for the type of FPGA chip installed on your XS40 Board.

Or you can download a CPLD design into your XS95 Board as follows:

```
C:\> XSLOAD CIRCUIT.SVF
```

where CIRCUIT.SVF is a bitstream file that contains the configuration for the XC9500 CPLD.

Use one of the following commands if you need to configure the FPGA or CPLD and also download an Intel-formatted HEX file into the SRAM of the XS40 or XS95 Board:

```
C:\> XSLOAD FILE.HEX CIRCUIT.BIT
C:\> XSLOAD FILE.HEX CIRCUIT.SVF
```

where CIRCUIT.BIT and CIRCUIT.SVF are bitstream files for the XC4000 or XC9500 programmable logic device and FILE.HEX is a file containing hexadecimal data. The HEX file could contain microcontroller object code generated by the ASM51 assembler, or it could be arbitrary data from some other source. Whatever its source, the hexadecimal data is downloaded into the XS Board SRAM.

XSLOAD assumes the XS Board is connected to parallel port #1 of your PC. You can specify another port number using the -P option like so:

```
C:\> XSLOAD -P 2 FILE.HEX CIRCUIT.BIT
```

Using XSPORT

Assuming your CPLD or FPGA design has its inputs assigned to pins which are connected to the PC parallel port, then you can use XSPORT to force values on the inputs. XSPORT supports up to eight inputs. For example, to force the binary values 1, 0, and 1 onto the three least-significant bits of the parallel port, use the following command:

```
XSPORT 101
```

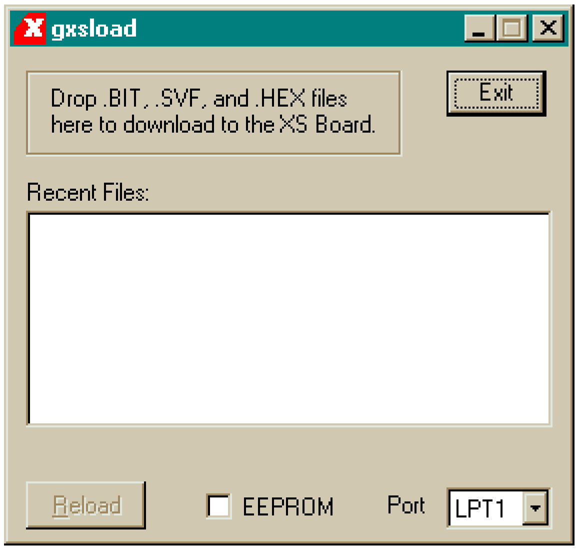
In the example given above, the upper five bits of the parallel port are forced to logic 0.

Like XSLOAD, XSPORT assumes the XS Board is connected to parallel port #1 of your PC. If you are using another port number, you can specify that like so:

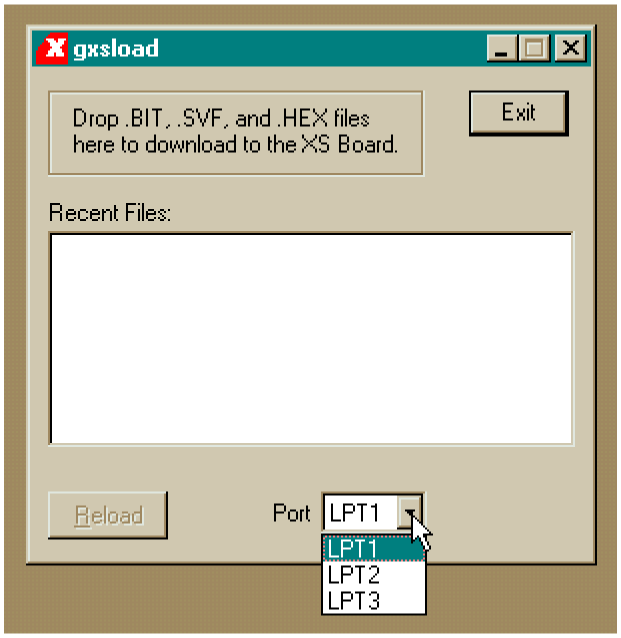
```
XSPORT -p 2 101
```

Using GXSLOAD

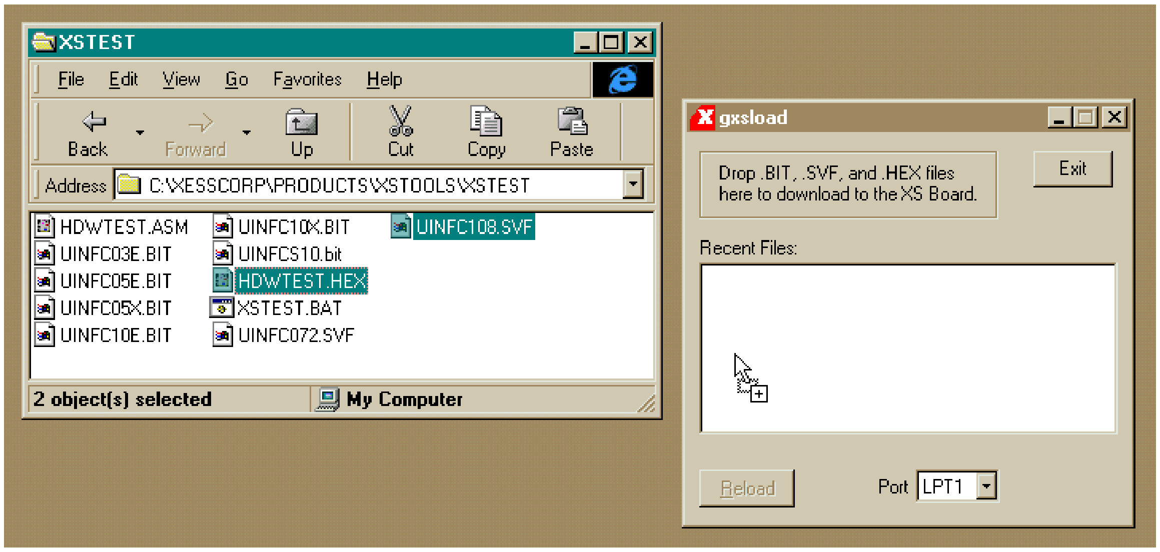
You start GXSLOAD by clicking on the GXSLOAD icon placed on the desktop during the GXSTOOLs installation. This brings up the screen shown below.


gxsload
Drop .BIT, .SVF, and .HEX files here to download to the XS Board.
Exit
Recent Files:
Reload
EEPROM
Port
LPT1

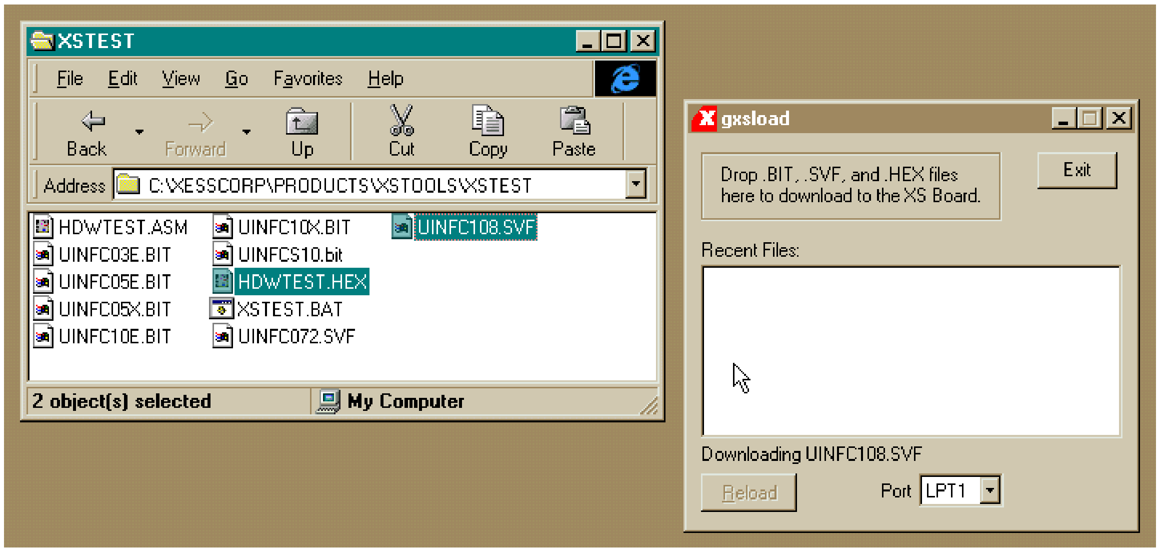
Your next step is to select the parallel port that your XS Board is connected to as shown below. GXSLOAD starts with parallel port LPT1 as the default, but you can also select LPT2 or LPT3 depending upon the configuration of your PC. If you are programming an XS40 Board with an Atmel serial EEPROM, you can also check the EEPROM box to enable the programming of the EEPROM. In most cases, however, you will leave the box unchecked so that the FPGA or CPLD on the XS Board will be programmed.


gxsload
Drop .BIT, .SVF, and .HEX files here to download to the XS Board.
Exit
Recent Files:
Reload
Port
LPT1
LPT1
LPT2
LPT3

After setting the parallel port and EEPROM flag, you can download files to the XS Board simply by dragging them to the GXSLOAD window as shown below. Once you release the mouse left-button and drop the files, GXSLOAD will begin sending the files to the XS Board through the parallel port connection. If you drag & drop a non-downloadable file (one with a suffix other than .BIT, .SVF, or .HEX), GXSLOAD will ignore it.

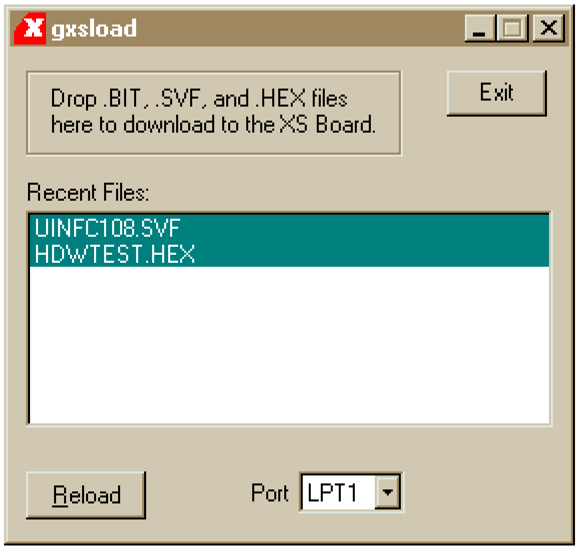


During the process, GXSLOAD will display the name of the file currently being downloaded below the Recent Files window as shown below.



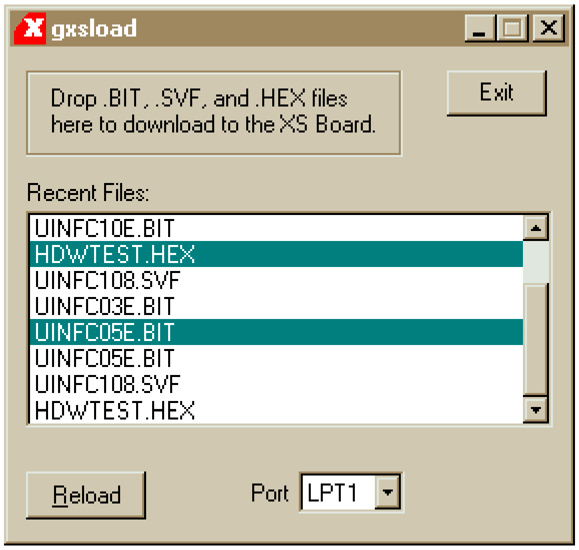
Downloading UINFC108.SVF

Once the downloading is finished, the file names are added to the Recent Files window and the Reload button is enabled. Now you can download these files to the XS Board just by clicking on the Reload button. This is a useful shortcut to have as you make changes to your design in Foundation and need to test the modifications.


gxsload
Drop .BIT, .SVF, and .HEX files here to download to the XS Board.
Exit
Recent Files:
Reload
Port
LPT1

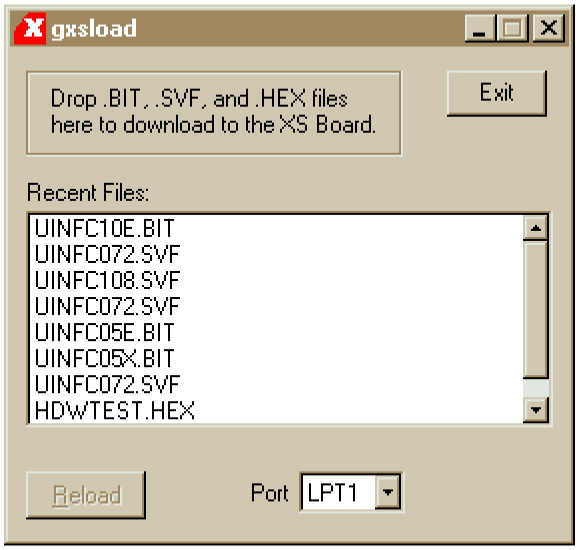
UINFC108.SVF
HDWTEST.HEX

The Recent Files window records the name of each file you download. As shown below, a scrollbar will appear once you have dropped more than eight files on the GXSLOAD window. You can click your mouse on multiple file names to toggle their selections on or off. Then clicking on the Reload button will download the highlighted files to the XS Board.


gxsload
Drop .BIT, .SVF, and .HEX files here to download to the XS Board.
Exit
Recent Files:
Reload
Port
LPT1

UINFC10E.BIT
HDWTEST.HEX
UINFC108.SVF
UINFC03E.BIT
UINFC05E.BIT
UINFC05E.BIT
UINFC108.SVF
HDWTEST.HEX

Note that the Reload button is disabled if you do not select any files to be downloaded. This situation is shown below.



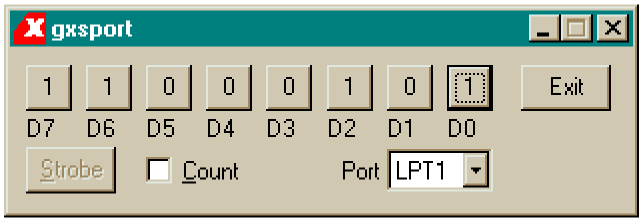
Drop .BIT, .SVF, and .HEX files here to download to the XS Board.

Recent Files:

```
UINFC10E.BIT
UINFC072.SVF
UINFC108.SVF
UINFC072.SVF
UINFC05E.BIT
UINFC05X.BIT
UINFC072.SVF
HDWTEST.HEX
```

Using GXSPORT

Once you have loaded the XS Board with a configuration file using GXSLOAD, you can then use GXSPORT to exercise the functions of your design. Click on the GXSPORT icon to bring up the window shown below.



The window contains several controls which perform the following functions:

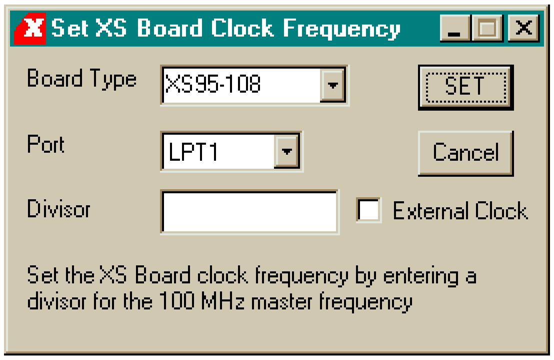
- The Port list box lets you select the parallel port that your XS Board is connected to just like you did with the GXSLOAD program.
- There are eight buttons, each of which is associated with one of the eight data bits of the parallel port. On startup, each button is labeled with the binary value currently output on the associated data pin. When you click one of these buttons, the displayed binary value toggles **but this new value does not appear on the data pin until you press the Strobe button** (see below).
- The Strobe button transfers the bit values displayed on the data button to the data pins of the parallel port. The Strobe button is enabled if at least one value on a data button is different from the actual value output on its data pin. The Strobe button is disabled if the value on each data pin matches the value on its associated button, because then there is no need to transfer the values.

- If you check the Count box, the value output on the data pins will increment every time you click on the Strobe button. In this case, the Strobe button will stay enabled.

- Clicking the Exit button terminates GXSPORT without updating the data pins with any new values that may have been entered.

Using GXSSETCLK

You start GXSSETCLK by clicking on the GXSSETCLK icon placed on the desktop during the GXSTOOLs installation. This brings up the screen shown below.



Board Type: XS95-108
Port: LPT1
Divisor:

Set the XS Board clock frequency by entering a divisor for the 100 MHz master frequency

Your next step is to select the parallel port that your XS Board is connected to from the port pulldown list. GXSSETCLK starts with parallel port LPT1 as the default, but you can also select LPT2 or LPT3 depending upon the configuration of your PC. After selecting the parallel port, you select from the pulldown list the type of XS Board you have connected to the PC parallel port.

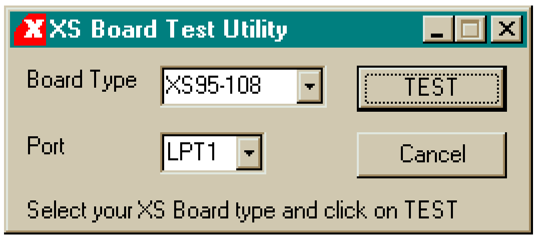
Next you must enter a divisor between 1 and 2052 into the text box. Once programmed, the oscillator will output a clock signal generated by dividing its 100 MHz master frequency by the divisor. The divisor is stored in non-volatile storage in the oscillator chip so you only need to use GXSSETCLK when you want to change the frequency.

An external clock signal can be substituted for the internal master frequency of the programmable oscillator. Checking the external clock checkbox will enable this feature in the programmable oscillator chip. Of course, you are then responsible for providing the external clock to the XS Board.

Clicking on the SET button will start the oscillator programming procedure. Status messages will be printed at the bottom of the GXSSETCLK window as the programming proceeds. You will also receive instructions on how to set the shunts on the XS Board jumpers to place the oscillator into its programming mode. At the end of the programming, you will receive a message informing you that your XS Board clock has been set.

Using GXSTEST

You start GXSTEST by clicking on the GXSTEST icon placed on the desktop during the GXSTOOLs installation. This brings up the screen shown below.



Select your XS Board type and click on TEST

Your next step is to select the parallel port that your XS Board is connected to from the port pulldown list. GXSTEST starts with parallel port LPT1 as the default, but you can also select LPT2 or LPT3 depending upon the configuration of your PC.

After selecting the parallel port, you select the type of XS Board you are testing from the associated pulldown list. Then click on the TEST button to start the testing procedure. GXSTEST will program the microcontroller and the FPGA or CPLD to perform a test procedure. Status messages will be printed at the bottom of the GXSTEST window as the testing proceeds. At the end of the test, you will receive a message informing you whether your XS Board passed the test or not.