XAPP1292 (v1.0) October 5, 2016

Loading Partial Bitstreams using TFTP

Author: David Robinson

Summary

This application note presents a software library (PR TFTP) written in C, that can be used to fetch partial bitstreams over Ethernet from a Trivial File Transfer Protocol (TFTP) server. A partial bitstream discovery mechanism is provided for applications that expect their available partial bitstreams to change over time. Fetching partial bitstreams from a TFTP server can be useful in designs where partial bitstreams cannot be stored in local read-only memory, or where centralized control over the availability of partial bitstreams is required.

Three examples are included which make use of a MicroBlaze™ Processor to fetch bitstreams that are used to partially reconfigure the design. The ARM® processor on a Zynq® device could be used instead of a MicroBlaze.

More information on Partial Reconfiguration can be found in the *Vivado Design Suite: Partial Reconfiguration User Guide (UG909)* [Ref 1]. More information on the TFTP protocol can be found in RFC 1350 ([Ref 3]).

Introduction

Some partially reconfigurable applications are self-controlling, making their own decisions about which Reconfigurable Modules to load, and when to load them. These designs commonly fetch their bitstreams on demand from on-board read-only memory. This is a simple way of creating a partially reconfigurable design, but can be limited by the relatively small size of read-only memories. If the combined size of a design's static and partial bitstreams is close to the size of the local read-only memory, then late changes to the design such as increasing the size of a Reconfigurable Partition or adding new Reconfigurable Modules could cause the memory's capacity to be exceeded.

One way to mitigate this risk, yet keep the advantages of a self controlling design, is to store the bitstreams off-board in some kind of centralized bulk storage and have the application fetch the bitstreams from there rather than from local read-only memory. The application can fetch these on demand and pass them straight to the configuration port, or buffer them in local dynamic memory at startup.

Using a centralized bulk storage scheme for partial bitstreams can offer other advantages, such as easier management of in-field updates or lower overall storage costs for a system consisting of multiple identical FPGAs.

System Requirements

Fetching Partial bitstreams from a TFTP server requires the following system components:

1. A TFTP server and associated bitstream storage that can be reached by the application using an Ethernet connection.
2. An on-board CPU and memory to run the C software.
3. An Ethernet interface.
4. An interrupt controller.
5. An interface to the configuration port, such as the Partial Reconfiguration Controller IP.
6. The LightWeight IP (LwIP) software library (provided with SDK).

The following blocks may be required depending on the desired behavior:

1. A timer block to allow the TFTP client to detect transmission timeouts. This block can be avoided if the timeout support is not required.
2. An external memory interface, such as the Memory Interface Generator (MIG) IP, if bitstreams are to be stored in a local memory before being used.

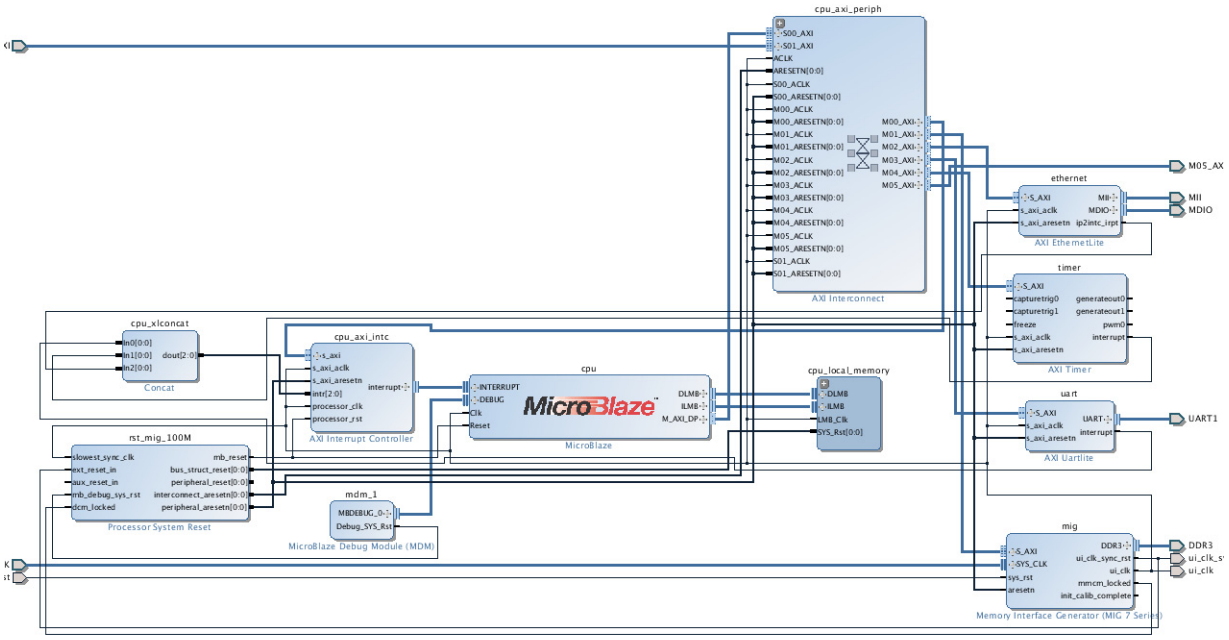

Figure 1: **Hardware blocks required for TFTP bitstream fetch**

About TFTP

TFTP is a lockstep file transfer protocol implemented on top of the UDP/IP protocols, and is commonly used for network booting and firmware transfers to simple network appliances due to its simplicity and small code size. The file to be transferred is sent in blocks with a maximum size of 512 bytes, and each block must be acknowledged before another is sent. Blocks are sent in-order which means they can be passed directly to the configuration port if desired. Optional extensions to the TFTP protocol have been defined to improve performance (for example, to increase the maximum block size and to increase the number of consecutive blocks that can be sent before an acknowledge is required).

The library described in this application note implements a limited TFTP client that can read files from a server. It does not support any of the optional TFTP extensions. The source code is made available for this library, enabling the user to add write operations and optional extensions if required. Refer to RFC 1350 ([Ref 3]) for further details on how TFTP operates.

Using the PR TFTP Library

Downloading the Library

Download the source code for the library and example design files for this application note from the Xilinx Website. Extract the ZIP file contents to any write-accessible location. This location is referred to as the `<Extract_Dir>` in this application note. The source code for the library is in `<Extract_Dir>/common/Sources/sw/pr_tftp_lib`.

Including the Library in SDK

To include the PR TFTP library in an SDK project, the following steps are required:

1. Enable the LwIP Library in the Board Support Package.
2. Add the PR TFTP source file (`<Extract_Dir>/common/Sources/sw/pr_tftp_lib/pr_tftp.c`) to the project.
3. Add the `<Extract_Dir>/common/Sources/sw/pr_tftp_lib` directory as an include path.

Initializing the Library in an Application

The best way to use the PR TFTP library is to start with an LwIP example design that is generated by SDK. This ensures the CPU, Ethernet interface, interrupt controller, LwIP, etc., are all initialized correctly for your platform's configuration.

No special integration is required for the PR TFTP library if TFTP timeouts are not required. The following steps are needed if timeouts are required:

1. Create a variable that holds a timer count value. For example:

```
volatile int unsigned PR_TFTP_TimerCount = 0;
```

The name of this variable is not important, but it needs to be visible from the code that will request TFTP transfers.

2. Update the timer callback function to increment this variable when the timer interrupt occurs:

```
void timer_callback(){
        PR_TFTP_TimerCount++;
}
```

The exact duration of the timer duration is not important. The file fetch functions in the PR TFTP library compare the value of the accumulated count value (PR_TFTP_TimerCount in this case) against a threshold value passed as a parameter to the function call, so any timer duration can be used as long as the threshold is adjusted accordingly.

For example, if a timeout of 20 seconds is required, a timer duration of 1 second with a threshold of 20 can be used. However, if the design requires a timer with a duration of 1 ms for other reasons, the threshold can be set to 20,000 to achieve the same 20 second timeout value.

Using the Library

Fetching a Partial Bitstream

The library has two functions to fetch a Partial Bitstream from a TFTP server. One stores the bitstream in a memory buffer and the other passes it in chunks to a user defined callback function. Both of these functions fetch data in network byte order (Big Endian). Depending on how the partial bitstreams are generated, and the endianness of the system, the received data may need to be manually converted to host byte order using functions such as `ntohl()`.

For example:

- A Little Endian system that uses BIN files created by write_bitstream will have to convert the data (see example 3) because the TFTP process converts the Little Endian BIN file to Big Endian
- A Little Endian system using BIN files created by write_cfgmem -interface SMAPx32 will not have to convert the data (see examples 1 and 2) because the bitstreams are converted to Big Endian by write_cfgmem and then reversed by the TFTP process which means they arrive in Little Endian format

Note: If you encounter difficulties getting partial bitstreams to load correctly then check the endianness of the received data to ensure it matches the endianness of the file on disk.

The following function is used to fetch a partial bitstream from the TFTP server to local memory:

```
pr_tftp_err_t PR_TFTP_FetchPartialToMem(
     struct netif        *  LocalNetif,
     struct ip_addr         ServerIPAddr,
     char                *  Filepath,
     char                ** ppDataBuffer,
     u32                 *  DataBufferSize,
     u32                 * DataBufferUsed,
```

```
        volatile int unsigned * pTimeoutTickCount,
        int unsigned            TimoutThreshold,
        int unsigned            TimeoutRetryAttempts,
        pr_tftp_options_s     *  pOptions);
```

Table 1: **PR_TFTP_FetchPartialToMem Parameters**

Parameter	Description
LocalNetif	The network interface to use for the TFTP transfer. This needs to be initialised before use.
ServerIPAddr	The IP address of the TFTP server stored in an "ip_addr" structure.
*Filepath	The path to the file on the TFTP server. This can be a pointer to a string containing the file name, or just the file name directly. For example, `"example_1/rp1_rm0.bin"` could be passed directly to the function.
**ppDataBuffer	A pointer to a pointer to the data buffer that will contain the partial bitstream. A pointer to a pointer is used because this function can be configured to reallocate the buffer if it is not big enough for the received file.
*DataBufferSize	A pointer to a variable holding the size (in bytes) of the buffer. The function will update this if the buffer is reallocated.
*DataBufferUsed	A pointer to a variable that returns the number of bytes stored. This value can be used to program the configuration controller.
*pTimeoutTickCount	A pointer to a counter incremented in a timer callback. The function will set this to zero when a timeout window is started. ***Note:*** There is no protection on this variable, so there may be a race between the timer callback incrementing it and the TFTP fetch function setting it to zero. This is unimportant as the TFTP timeout period does not have to be very accurate. It is only needed to handle the cases where there is no TFTP server, or no path to the TFTP server. If timeout detection is not required, then this variable does not need to be incremented in a timer callback.
TimeoutThreshold	The number of timer timeouts to wait for before declaring a TFTP timeout.
TimeoutRetryAttempts	The number times to retry a packet before abandoning the transfer.
*pOptions	A pointer to a struct containing some options to control the transfer. For example, if the function is allowed to reallocate the buffer memory, and if so, how much to add every time it needs to reallocate it. If `pOptions->ReallocateMemoryIfRequired` is set to 1, the function will allocate memory for the bitstream. If `pOptions->ReallocateMemoryIfRequired` is set to 0, then the buffer passed to the function must be pre-allocated to be large enough to hold the bitstream being fetched. See `<Extract_Dir>/common/Sources/sw/pr_tftp_lib/pr_tftp.h` for more information. The function does not modify the contents of this struct.

If this function completes successfully, then the partial bitstream will exist in the memory pointed to by ppDataBuffer.

The following function is used to fetch a partial bitstream from the TFTP server and pass the data (one packet at a time) to a user supplied callback function:

```
pr_tftp_err_t PR_TFTP_FetchPartialToFunction(
     struct netif          *    LocalNetif,
     struct ip_addr             ServerIPAddr,
```

```
char                    *    Filepath,
pr_tftp_recv_data_fn         RecvDataCallback,
void                    *    RecvDataCallbackArg,
volatile int unsigned   *    pTimeoutTickCount,
int unsigned                 TimeoutThreshold,
int unsigned                 TimeoutRetryAttempts,
pr_tftp_options_s       *    pOptions);
```

Most of these parameters are identical to those described in Table 1. The different parameters are listed below:

Table 2: **PR_TFTP_FetchPartialToFunction parameters**

Parameter	Description
RecvDataCallback	The function to call when a data packet is received. The prototype is: `void <function name> (void *Arg, char *Data, int unsigned NumberOfBytes);` Arg is a user defined variable that is passed to the callback function. Data is a character array of received data stored in network byte order.
RecvDataCallbackArg	A pointer to a variable to pass to the RecvDataCallback function.

Fetching Information about Available Reconfigurable Modules

Fetching partial bitstreams over Ethernet rather than storing them in read-only memory when the system is deployed introduces a great deal of flexibility into the design. For example:

- Reconfigurable Modules can be changed during in-field updates.
- Reconfigurable Modules can be added to, or removed from, the design after deployment.
- The Reconfigurable Modules returned by the server can be varied depending on the environmental or licensing conditions.

This flexibility can introduce some challenges to the application, namely:

1. What Reconfigurable Modules exist for this design?
2. How big are their bitstreams?
3. How should the configuration controller manage their loading and unloading?

The PR TFTP library offers a solution to this by providing a function to fetch a comma separated value (CSV) file that contains a list of Reconfigurable Modules, and information about each one. It also provides a function to convert this file into a hierarchical data structure for easy access.

The fields in the CSV file are listed below in order:

1. Reconfigurable Partition ID (16 bits)
2. Reconfigurable Module ID (16 bits)
3. Reset Duration (8 bits)
4. Reset Required (8 bits)
5. Startup Required (8 bits)

6. Shutdown required (8 bits)
7. Bitstream Size (32 bits)
8. File Name

For example:

Table 3: **Example of a CSV file**

#RP_ID	RM_ID	Reset Duration	Reset Required	Startup Required	Shutdown Required	BS_SIZE	FILE_NAME
0,	0,	0,	3,	0,	0,	375300,	example_2/shift_left.bin
0,	1,	1,	3,	0,	0,	375300,	example_2/shift_right.bin

Lines beginning with '#' are comments and are ignored by the parser. Empty fields such as ',,' and ', ,', will cause the parser to generate an error.

The CSV file requires two extra fields for the Ultrascale™ devices:

9. Clearing Bitstream Size (32 bits)
10. Clearing File Name

Support for these is enabled if the PR_TFTP_REQUIRES_CLEARING_BITSTREAM pre-processor symbol is defined. This should be done in **applications properties** > **C/C++ Build** > **Settings** > **MicroBlaze gcc compiler** > **Symbols**.

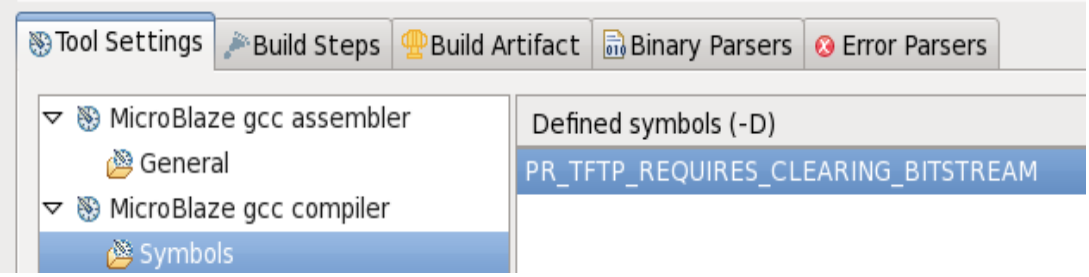

Figure 2: **Enabling support for UltraScale devices**

The information stored in the CSV is specific to the *Partial Reconfiguration Controller IP (PG193)* [Ref 6], but it will be generally useful for many applications. The code can be modified to change the fields in the file if required.

The following function is used to fetch a Reconfigurable Module Information File from the TFTP server to local memory:

```
pr_tftp_err_t PR_TFTP_FetchRmInfoToMem (
     struct netif          *  LocalNetif,
     struct ip_addr           ServerIPAddr,
     char                  *  Filepath,
     char                  ** ppDataBuffer,
     u32                   *  DataBufferSize,
     u32                   *  DataBufferUsed,
     volatile int unsigned * pTimeoutTickCount,
     int unsigned             TimoutThreshold,
```

```
        int unsigned              TimeoutRetryAttempts,
        pr_tftp_options_s     *   pOptions);
```

The parameters are identical to the ones used to fetch a partial bitstream, as shown in Table 1. If this function completes successfully then the information file will exist in the memory pointed to by **ppDataBuffer**.

Note: This function does not parse the file, so it can be used to fetch any ASCII file required by the application.

Accessing Information about Available Reconfigurable Modules

The following function populates a data structure with the contents of this CSV file:

```
pr_tftp_err_t PR_TFTP_InitialiseDataStructureFromRmInfoFile(
     char                    *  pRmInfoFile,
     u32                        FileSize,
     u16                        NumRPs,
     pr_tftp_rp_s            ** pRPInfoArray);
```

Table 4: **PR_TFTP_InitializeDataStructureFromRmInfoFile parameters**

Parameter	Description
*pRmInfoFile	A pointer to the memory holding file.
FileSize	The size of the file in bytes.
numRPs	The number of Reconfigurable Partitions in the design.
**pRPInfoArray	A pointer to a pointer to the data structure that will contain information about the Reconfigurable Modules. This structure will be allocated and initialized by the function. Ownership of the memory is passed to the calling function, which needs to deallocate it when it is no longer needed. A function is provided to deallocate the data structure correctly. The structure should be declared as: `pr_tftp_rp_s *<variable name>;` where, `<variable name>` should be replaced by the name you want to give the variable.

IMPORTANT: *This function will only work if the file is in the format described above. If it is used with different fields then the function and data structure will have to be changed to match, or replaced with an alternative.*

The following function is provided to free the memory used by this data structure:

```
pr_tftp_err_t PR_TFTP_FreeDataStructure(
     u16              numRPs,
     pr_tftp_rp_s ** pRPInfoArray);
```

Table 5: **PR_TFTP_FreeDataStructure Parameters**

Parameter	Description
NumRPs	The number of Reconfigurable Partitions in the design.
**pRPInfoArray	A pointer to a pointer to the data structure containing information about the Reconfigurable Modules.

The following code fragment shows the Reconfigurable Module Information File being fetched, turned into a data structure, and freed:

```
// ------------------------------------------------------------
// Fetch the Reconfigurable Module Information File
// ------------------------------------------------------------
//
// Setup some transfer options
TransferOptions.ReallocateMemoryIfRequired = 1;
TransferOptions.IncrementAmount            = 1024;
RmInfoFileBufferSize                       = 0;

Err = PR_TFTP_FetchRmInfoToMem(
    &LocalNetif                          , // The network interface to use.
    ServerIpAddr                         , // The TFTP server IP address.
    "example_3/rm_info.csv"              , // The path & name of the file
                                           // to fetch.
    &pRmInfoFile                         , // A pointer to a memory buffer
                                           // to store the file. This will
                                           // be updated if more memory is
                                           // allocated.
    &RmInfoFileBufferSize                , // A pointer to the size of the
                                           // buffer. This will be updated
                                           // if more memory is allocated.
    &RmInfoFileBufferUsed                , // A pointer to a variable to
                                           // return the number of bytes
                                           // stored.
    &PR_TFTP_TimerCount                  , // A pointer to a counter
                                           // incrementedin the timer
                                           // callback.
    PR_TFTP_TIMEOUT_THRESHOLD            , // How long to wait on a TFTP
                                           // packet before timing out.
    PR_TFTP_TIMEOUT_RETRY                , // How many times to retry a
                                           // failed packet.
    &TransferOptions
                                    );
if (Err != PR_TFTP_ERR_OK) {
  return XST_FAILURE;
}

// ------------------------------------------------------------
// Turn the File into a Data Structure
// ------------------------------------------------------------
//
if (PR_TFTP_InitialiseDataStructureFromRmInfoFile(
         pRmInfoFile, RmInfoFileBufferUsed,
         NumRPs     , &pRPInfoArray) != PR_TFTP_ERR_OK) {
  return XST_FAILURE;
}

// Free the buffer that holds the RM Information CSV file
free(pRmInfoFile);

// MAIN APPLICATION GOES HERE

// ------------------------------------------------------------
// Free the data structure containing the RM information
// ------------------------------------------------------------
//
PR_TFTP_FreeDataStructure(NumRPs, &pRPInfoArray);
```

The information retrieved from the Reconfigurable Module Information File is stored in an array of pr_tftp_rp_s structs, each of which contain information about that Reconfigurable Partition, and an array of pr_tftp_rm_s structs. These hold information about a Reconfigurable Module.

```
// A struct to hold information about a single Reconfigurable
// Partition
//
typedef struct pr_tftp_rp_s {
  u16          Id;
  u16          NumberOfRMs;
  s32          ActiveRM;
  pr_tftp_rm_s ** pRMInfos;
} pr_tftp_rp_s;
```

Table 6: **pr_tftp_rp_s struct fields**

Field	Description
Id	The Reconfigurable Partition Identifier
NumberOfRMs	The number of Reconfigurable Modules in this Reconfigurable Partition
ActiveRM	The ID of the currently loaded Reconfigurable Module. Set to -1 if nothing is loaded
pRMInfos	An array of Reconfigurable Module structs

```
// A struct to hold information about a single Reconfigurable
// Module
//
typedef struct pr_tftp_rm_s {
 u16          Id;
 u8           ResetDuration;
 u8           ResetRequired;
 u8           StartupRequired;
 u8           ShutdownRequired;
 u16          BsIndex;
 u32          BsSize;
 char *       pFileName;

 #ifdef PR_TFTP_REQUIRES_CLEARING_BITSTREAM
    u16       ClearingBsIndex;
    u32       ClearingBsSize;
    char *    pClearingFileName;
 #endif

} pr_tftp_rm_s;
```

Table 7: **pr_tftp_rm_s struct fields**

Field	Description
Id	The Reconfigurable Module Identifier
ResetDuration	The length of reset that this Reconfigurable Module needs. If the Partial Reconfiguration Controller IP is used, then this is in clock cycles
ResetRequired	The type of reset this Reconfigurable Module needs. See the *Partial Reconfiguration Controller Product Guide (PG193)* [Ref 6] for encodings, if the Partial Reconfiguration Controller is being used.

Table 7: **pr_tftp_rm_s struct fields *(Cont'd)***

Field	Description
StartupRequired	The type of startup this Reconfigurable Module needs. ee the *Partial Reconfiguration Controller Product Guide (PG193)* [Ref 6] for encodings, if the Partial Reconfiguration Controller is being used.
ShutdownRequired	The type of shutdown this Reconfigurable Module needs. ee the *Partial Reconfiguration Controller Product Guide (PG193)* [Ref 6] for encodings, if the Partial Reconfiguration Controller is being used.
BsIndex	The index of the bitstream in the Partial Reconfiguration Controller's Bitstream Information Register Bank that holds information about the partial bitstream for this Reconfigurable Module. See the *Partial Reconfiguration Controller Product Guide (PG193)* [Ref 6] for more information.
BsSize	The size of the partial bitstream (in bytes)
pFileName	The full name (including the path) of the partial bitstream on the TFTP server
ClearingBsIndex	The index of the bitstream in the Partial Reconfiguration Controller's Bitstream Information Register Bank that holds information about the clearing bitstream for this Reconfigurable Module. See the *Partial Reconfiguration Controller Product Guide (PG193)* [Ref 6] for more information. This field is only enabled if the PR_TFTP_REQUIRES_CLEARING_BITSTREAM pre-processor symbol is defined
ClearingBsSize	The size of the clearing bitstream (in bytes). This field is only enabled if the PR_TFTP_REQUIRES_CLEARING_BITSTREAM pre-processor symbol is defined
pClearingFileName	The full name (including the path) of the clearing bitstream on the TFTP server. This field is only enabled if the PR_TFTP_REQUIRES_CLEARING_BITSTREAM pre-processor symbol is defined

Note: The ActiveRM fields in the pr_tftp_rp_s structs are set to -1, when the data structure is allocated. This information is not carried in the Reconfigurable Module Information File. The application software will have to initialise it based on its knowledge of the system.

For example, the following code can be used with the Partial Reconfiguration Controller IP:

```
for (Id = 0; Id < XPrc_GetNumberOfVsms(pPrcConfig); Id++){
  // Set the default active RM for each VS
  //
  if (XPrc_GetHasPorRm (pPrcConfig, Id)) {
    pRPInfoArray[Id].ActiveRM = XPrc_GetPorRm(pPrcConfig, Id);
  }
}
```

The PR TFTP library contains the following functions to access Reconfigurable Partition and Reconfigurable Module information from the data structure:

```
pr_tftp_rp_s * PR_TFTP_GetRPInfoByIndex()
pr_tftp_rp_s * PR_TFTP_GetRPInfoByID()
pr_tftp_rm_s * PR_TFTP_GetRMInfoByIndex()
pr_tftp_rm_s * PR_TFTP_GetRMInfoByID()
```

If the Reconfigurable Partition information structures and the Reconfigurable Module information structures are stored in order in the data structure (i.e., if their array indices are the same as their IDs) then use the "ByIndex" functions as these are faster. If the Reconfigurable Partition information structures and the Reconfigurable Module information structures are stored out of order (i.e., if their array indices are not the same as their IDs) then use the "By ID" functions. These are slower to execute because the appropriate arrays have to be searched to find the structure with the correct identifier.

If the data structure is created using PR_TFTP_CreateDataStructure then the Reconfigurable Partition information structures will always be stored in-order. Reconfigurable Module information structures will be stored in the order they appear in the Reconfigurable Module Information File.

See `<Extract_Dir>/common/Sources/sw/pr_tftp_lib/pr_tftp.c` for more information about these functions.

An example of these functions being used is:

```
pr_tftp_rp_s *pRPInfoArray;
pr_tftp_rp_s *pRP;
pr_tftp_rm_s *pRM;
// Loop through all Reconfigurable Partitions
//
for (RpId = 0; RpId < NumRPs; RpId++){

  // Get information about the RP
  pRP = PR_TFTP_GetRPInfoByIndex(pRPInfoArray, NumRPs, RpId);

  if (pRP == NULL) {
    return XST_FAILURE;
  }

  // Loop through all Reconfigurable Modules in this RP
  //
  for (RmId = 0; RmId < pRP->NumberOfRMs; RmId++){
    pRM = PR_TFTP_GetRMInfoByID(pRP, RmId);

    if (pRM == NULL) {
      return XST_FAILURE;
    }

    // Access information about the Reconfigurable Module
    //
    pBuffer = malloc (pRM->BsSize);

    // Fetch the bitstream from the server, program the
    // configuration controller, etc.
  }
}
```

Example Designs

The PR TFTP library ships with three example designs that are targeted to the KC705 board:

1. In example 1, all the bitstreams are fetched on startup to DDR memory. The Partial Reconfiguration Controller IP knows how to process each RM, and the software application knows the names of the bitstream files at compilation time, so no information has to be fetched describing the Reconfigurable Modules. The received bitstream data does not have to be converted from the Network Byte Order.
2. In example 2, all the bitstreams are fetched on startup to DDR memory. However, the Partial Reconfiguration Controller IP does not know how to process each RM, and the software application does not know the names of the bitstream files, so information has to be fetched describing the Reconfigurable Modules. The received bitstream data does not have to be converted from the Network Byte Order.
3. In example 3, bitstreams are only fetched when required and are passed directly to the ICAP (through the AXI HWICAP IP) using a callback function. The partial bitstreams are not stored in DDR memory. The received bitstream data has to be converted from Network Byte Order.

Each example has a detailed description in the comments at the start of its `main.c` file. The following table gives a summary of each example for comparison:

Table 8: **The three example designs compared**

Example	Uses RM Information File?	Partial Bitstreams are fetched...	Partial Bitstreams are fetched to...	Configuration Mechanism	Data needs to be converted from Network Byte Order?
1	No	At startup	DDR Memory	Partial Reconfiguration Controller IP	No
2	Yes	At startup	DDR Memory	Partial Reconfiguration Controller IP	No
3	Yes	When needed	A callback function	AXI HWICAP IP	Yes

These designs are based on the *Vivado Design Suite Tutorial: Partial Reconfiguration (UG947)* [Ref 9] which has two partitions, each with two Reconfigurable Modules:

Table 9: **Reconfiguration Modules**

Partition	Reconfiguration Modules
Count	Count Up, Count Down
Shift	Shift Left, Shift Right

In addition to these, the examples use the following major blocks:

- A MicroBlaze
- AXI Ethernet Lite Controller

- Interrupt Controller
- Timer
- Memory Interface Generator (MIG)
- Local Memory for CPU
- UART (used for debug)
- The Partial Reconfiguration Controller IP (examples 1 and 2) and the AXI HWICAP IP (example 3)

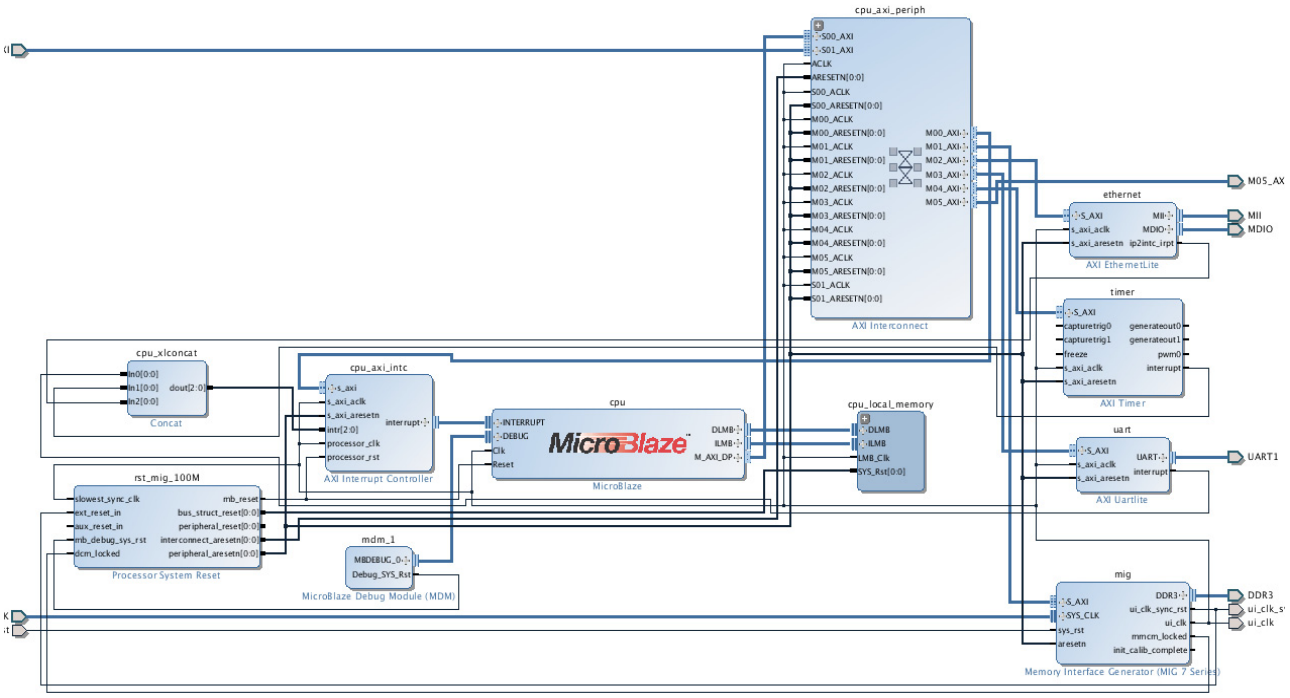

Figure 3: **Major Hardware Platform Blocks**

Requirements

To run the examples with minimal modifications, the requirements are as follows:

Hardware Requirements:

1. A KC705 board
2. An Ethernet cable
3. A computer running a TFTP server on the same LAN as the KC705 board

Software Requirements:

1. Vivado Design Suite 2016.3 or greater
2. Software Development Kit (SDK)

3. A TFTP Server. Free TFTP server software is available online if your system does not already have one.

Experienced users can modify the examples to support other boards and Xilinx CPUs.

Installation

1. Decompress the ZIP file to an empty directory of your choice. This will be referred to as `<Extract_Dir>`.
2. Install a TFTP server on the same network as the KC705 board.
3. Set a TFTP root directory which is referred to as `<TFTP_Root>`.
4. In `<TFTP_Root>`, create folders called `example_1`, `example_2`, and `example_3`.
5. Ensure that your firewall is configured to allow TFTP traffic.

Note: The project is split into common files and files specific to each example design. <N> will be used to show that the example number is needed. For example, if you are working on example 1, and you see a `<Extract_Dir>/example_<N>/Sources` path, this resolves to `<Extract_Dir>/example_1/Sources`.

Generating the Hardware Platform

Change the working directory to `<Extract_Dir>/example_<N>` and launch Vivado Design Suite. Execute the following command at the Vivado command-line:

```
source run.tcl -notrace
```

When complete, the partial bitstreams and the Reconfigurable Module Information File need to be copied to the TFTP server.

For example 1:

- copy `<Extract_Dir>/example_1/Partials/*.bin` to `<TFTP_Root>/example_1`

For example 2:

- copy `<Extract_Dir>/example_2/Sources/tftp/rm_info.csv` to `<TFTP_Root>/example_2`
- copy `<Extract_Dir>/example_2/Partials/*.bin` to `<TFTP_Root>/example_2`

For example 3:

- copy `<Extract_Dir>/example_3/Sources/tftp/rm_info.csv` to `<TFTP_Root>/example_3`
- copy `<Extract_Dir>/example_3/Bitstreams/*.bin` to `<TFTP_Root>/example_3`

Note: Copy these bin files the hardware is re-implemented.

A script called `copy_files_to_tftp_server.tcl` is provided with each example that will copy the appropriate files to the TFTP server, but this will only work if a TFTP client is installed

on your development system and your TFTP server has write access. From Vivado Design Suite, execute `source copy_files_to_tftp_server.tcl`

Note: This script has to edited to set the address of the TFTP server.

Generating the Software

Use the following steps to generate the example software:

1. Change to the sw directory in the example
2. Open SDK
3. Create a Hardware Platform Specification using `<Extract_Dir>/<example_N>/SDK/static_bd_wrapper.hdf` as the Target Hardware Specification
4. Create a Board Support Package (BSP) for this platform. Enable the LwIP library in the BSP settings
5. Create an empty application project. This is called `pr_app` in the following text, but any legal name can be used
6. Copy all files from `<Extract_Dir>/common/Sources/sw` to the `pr_app/src directory`
7. Copy all files from `<Extract_Dir>/<example_N>/Sources/sw` to the `pr_app/src directory`. The following Linux commands can be used:

   ```
   cp -R ../../../../common/Sources/sw/* .
   cp ../../../Sources/sw/* .
   ```

8. Refresh the project in SDK to see the files in the project
9. Add the following include paths as workspace paths:

   ```
   /pr_app/src
   /pr_app/src/pr_tftp_lib
   ```

   For example:

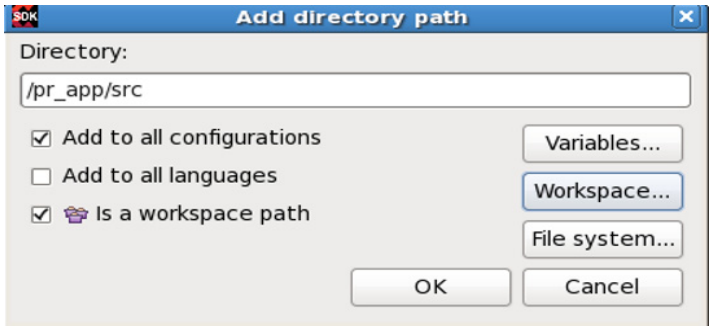

Figure 4: **Add Directory Path**

The software is ready to be compiled.

Configuring the Software

For each example, edit `<Extract_Dir>/example_<N>/sw/pr_app/src/main.c` and set the following values correctly:

```
unsigned char MacEthernetAddress[] = { 0x00, 0x0a, 0x35, 0x02, 0xAE, 0xE3 };
IP4_ADDR(&BoardIpAddr   , 149, 199, 131,  16);  // The board's IP address
IP4_ADDR(&Netmask       , 255, 255, 255,   0);  // The Netmask
IP4_ADDR(&GatewayIpAddr , 149, 199, 131, 254);  // The Gateway IP address
IP4_ADDR(&ServerIpAddr  , 149, 199, 131, 173);  // The TFTP server's IP address
```

Note:
The MAC address should be printed on a sticker on your KC705 board.
The BoardIpAddress needs to be assigned to you by your local IT team.
The Netmask and GatewayIpAddr will be given to you by your local IT team.
The ServerIpAddr is the address where your TFTP server is running.

Running the Design

To run the application:

1. Ensure your board is connected to the same network as your TFTP server using an Ethernet cable.
2. Ensure your board is connected to an SDK development machine using a JTAG to USB cable. This will allow SDK to download the static bitstream and debug the application.
3. The example designs use the UART output to send debug information. To view this on Windows based hardware you need a serial to USB cable and the Silicon Labs drivers that are available on the Silicon Labs website for CP210x USB-to-UART Bridge VCP Drivers [Ref 8]. A terminal emulation application is also required. Tera Term is freely available for this [Ref 7]. For Linux system, consult your local IT department.
4. Apply power to your board.
5. Start a Vivado HW Server instance on the machine with the board attached to allow SDK to communicate with the board.
6. Use SDK to program the FPGA. Select the `pr_app.elf` file as the Software Configuration.
7. Start the TFTP server and make sure it has the latest bitstreams.
8. Run the application from SDK.

Reference Design

Download the reference design files for this application note from the Xilinx website.

References

1. *Vivado Design Suite User Guide: Partial Reconfiguration* (UG909)
2. Description of Trivial File Transfer Protocol - https://en.wikipedia.org/wiki/Trivial_File_Transfer_Protocol
3. RFC1350, The TFTP Protocol (Revision 2) https://tools.ietf.org/html/rfc1350
4. *LightWeight IP (LwIP) Application Examples v5.1 Application Note* (XAPP1026)
5. *Vivado Design Suite Tutorial: Embedded Processor Hardware Design* (UG940)
6. *Partial Reconfiguration Controller Product Guide* (PG193)
7. The Tera Term Home Page (English)
8. The Silicon Labs Website for CP210x USB-to-UART Bridge VCP Drivers
9. *Vivado Design Suite Tutorial: Partial Reconfiguration* (UG947)

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
10/05/2016	1.0	Initial Xilinx release.

Please Read: Important Legal Notices